МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»

Направление подготовки/специальность
09.03.01 Информатика и вычислительная техника

направленность (профиль)
«Технологии разработки программного обеспечения»

Выпускная квалификационная работа

«Разработка бэкенда веб-системы для театральной индустрии»

Обучающегося 4 курса
очной формы обучения
Букина Данилы Юрьевича

Руководитель выпускной квалификационной работы:
Доктор педагогических наук, профессор кафедры информационных технологий и электронного обучения
Власова Елена Зотиковна

Рецензент:
Ученая степень *(при наличии)*, ученое звание *(при наличии)*, должность
Ф. И. О. *(указывается в именительном падеже)*

Санкт-Петербург
2023

Оглавление

ВВЕДЕНИЕ

В настоящее время Интернет считается одним из самых используемых источников получения информации. Так, если рассматривать информацию из отчёта Global Digital Reports, в начале 2022 года в Российской Федерации 129,8 млн человек являлись пользователями сети Интернет. Это количество составляет 89% от всего населения страны. Действительно, Интернет является удобным и быстрым источником для публикации и чтения различного вида информации.

В свою очередь, театральная индустрия представляет интерес для определённого круга людей – как для профессионалов в этой среде, так и для любителей постановок, которые посещают театры и ищут информацию о новых произведениях. Если кратко обобщить анализ представления информации о театральной индустрии в сети Интернет, то практически все источники являются веб-системами для покупки билетов в театры. Конечно, информация о театральных произведениях присутствуют в электронной энциклопедии Википедия, но данный ресурс не является узкоспециализированным, вследствие чего материал там не отличается обширностью и точностью. Поэтому театральная индустрия нуждается в веб-ресурсе, на котором можно найти информацию о театральных пьесах и событиях, связанных с индустрией театра. Это показывает **актуальность** данной работы.

Исходя из вышесказанного, **целью** настоящей работы является разработка серверной части (бэкенда) веб-базированного сервиса для театральной индустрии с возможностью просмотра информации о пьесах и добавления пользователями отзывов. Современные приложения позволяют разграничить труд разработчиков на две составляющие – бэкенд и фронтенд. В данной работе будет рассмотрен процесс разработки именно первого сегмента.

Предметом анализа является процесс разработки веб-системы, а **объектом** – бэкенд веб-ресурса для театральной индустрии.

Для достижения цели, обозначенной ранее, следует определить задачи, выполнению которых также посвящена настоящая работа:

- анализ технологий, доступных для бэкенд разработки;

- выбор технологического стэка;
- разработка архитектуры базы данных;
- проектирование системы;
- разработка программных модулей;
- оформление документации.

Результатом работы является API, к которому возможно обращение по ссылке: https://theatrebel.ru/api. Документация для проекта расположена по данному адресу: https://theatrebel.ru/swagger-ui/index.html. Исходный код веб-системы можно увидеть в GitHub репозитории: https://github.com/bee-joo/theatrebel.

1. АНАЛИЗ ТЕХНОЛОГИЙ ДЛЯ РАЗРАБОТКИ БЭКЕНДА

1.1 О бэкенд разработке

Для того, чтобы приступить к разработке бэкенда веб-системы для театральной индустрии и грамотно выстроить этот процесс, следует рассмотреть базовые понятия, которые характеризируют саму бэкенд разработку.

Бэкенд (от англ. backend) – это область веб-разработки, которая направлена на создание основной бизнес-логики веб-приложений. Бэкенд-программа обычно выполняется на удалённом компьютере – сервере. Бэкенд разработчики отвечают за обработку запросов от клиентской части (фронтенд), взаимодействие с базами данных и другими сервисами, а также обеспечение безопасности, производительности и надежности приложений. Пользователь в большинстве случаев коммуницирует с бэкендом через фронтенд. Фронтенд (от англ. frontend) – это «фасад» веб-приложения, то есть внешняя часть, отвечающая за клиентское взаимодействие и дизайн. Получается, что пользователь не видит само бэкенд-приложение, а коммуницирует с ним через клиент. Таким образом, мы подходим к определению клиент-серверной архитектуры.

Клиент-серверная архитектура – это архитектура приложений, при которой серверное ПО является поставщиком услуг, а клиентское ПО – потребителем услуг. Клиент обращается к серверу с помощью сетевых протоколов, а серверная программа обрабатывает запрос клиента, используя бизнес-логику, и возвращает ответ клиенту. Серверной программой как раз может быть приложение, разработанное с использованием технологий для бэкенда. Клиентской программой может быть браузер, а также такое ПО, как Postman и curl, которое создано для различного рода взаимодействия с сервером.

Языки программирования — это инструменты, с помощью которых разработчики пишут код для своих приложений. Существует множество языков программирования, которые могут использоваться для бэкенд разработки, но некоторые из самых популярных и востребованных на рынке сейчас это Python, Java, JavaScript (Node.js), PHP, Ruby и C#. Каждый из этих языков имеет свои преимущества и недостатки, а также специфические области применения.

Фреймворки — это наборы инструментов и библиотек, которые упрощают и ускоряют разработку веб-приложений, предоставляя готовые решения для часто встречающихся задач и проблем. Фреймворки обычно следуют определенным принципам и паттернам проектирования, таким как MVC (Model-View-Controller), REST (Representational State Transfer) или GraphQL (Graph Query Language).

Например, для языка Java существуют такие фреймворки как Spring Boot, Struts и Play. Spring Boot — это один из самых популярных и мощных фреймворков для Java и Kotlin, который предоставляет обширную экосистему для разработки веб-приложений и микросервисов на основе Spring Framework. Spring Boot упрощает конфигурацию и запуск приложений с помощью автоматического определения зависимостей и настроек. Struts — это один из самых старых фреймворков для Java, который реализует паттерн MVC. Play — это современный и высокопроизводительный фреймворк для Java и Scala, который основан на асинхронном программировании и RESTful архитектуре. Play позволяет создавать веб-приложения с горячей перезагрузкой кода, автоматической компиляцией и тестированием.

Таким образом, нам удалось рассмотреть понятие бэкенд разработки. Мы выяснили, что для построения этого процесса обязательно следует выбрать язык программирования, фреймворк и СУБД. Для проекта веб-системы для театральной индустрии качестве языка программирования был выбран Kotlin, в качестве фреймворка – Spring Boot, а в качестве СУБД – PostgreSQL. Далее подробнее рассказывается об этих технологиях и их преимуществах для разработки.

1.2 Об экосистеме Spring Boot

Так как мы рассмотрели основные понятия бэкенд разработки, хотелось бы перейти к обсуждению фреймворка Spring Boot и его экосистемы.

Spring — это универсальный фреймворк с открытым исходным кодом для Java-платформы. Он был создан Родом Джонсоном в 2002 году и предлагает разработчикам больше свободы и гибкости в проектировании и создании приложений, особенно корпоративных. Он также вводит новые возможности,

такие как инверсия управления, аспектно-ориентированное программирование и MVC.

Spring был выпущен под лицензией Apache 2.0 в июне 2003 года. На момент написания работы последняя версия фреймворка — 6.0.7, выпущенная 20 марта 2023 года. Spring состоит из множества модулей, которые можно использовать по отдельности или вместе для решения разных задач. Например, Spring Boot — это модуль для быстрого создания и запуска приложений с минимальной конфигурацией; Spring Data — это модуль для упрощения доступа к данным из разных источников; Spring Security — это модуль для обеспечения безопасности приложений; Spring Cloud — это модуль для разработки микросервисов и облачных приложений.

Spring стал одним из самых популярных и востребованных фреймворков в Java-сообществе. Он используется многими крупными компаниями, такими как Netflix, Amazon, Google, PayPal и другими. Он также имеет большую поддержку от разработчиков и активное сообщество. Spring — это фреймворк, который помогает создавать качественные, надежные и современные приложения на Java.

Такое долгое развитие фреймворка сказалось на его сложности – конфигурации становились всё труднее, а производительность могла уменьшаться. Из-за таких изъянов был разработан Spring Boot – это модуль фреймворка Spring, который упрощает создание и запуск приложений на основе Spring. Он предоставляет готовые настройки и зависимости для разных сценариев разработки, таких как веб-приложения, микросервисы, облачные приложения и другие. Spring Boot также встраивает в себя веб-серверы, такие как Tomcat, Jetty или Undertow. Spring Boot позволяет создавать самодостаточные, готовые к production-среде приложения, которые можно просто запустить.

Spring Boot также предлагает различные инструменты для разработки и мониторинга приложений, такие как Spring Boot Actuator, Spring Boot DevTools, Spring Boot CLI и Spring Boot Admin. Spring Boot интегрируется с многими другими модулями Spring и сторонними библиотеками, такими как Spring Data, Spring Security, Spring Cloud, GraphQL, Apache Camel и другими.

Рассмотрим также некоторые модули экосистемы Spring Boot, которые часто используются в разработке.

Spring Data — это модуль, который предоставляет универсальную модель для доступа к данным, сохраняя при этом особенности конкретного хранилища данных. Подробнее о технологиях доступа к СУБД, в частности про Spring Data, рассказано в следующем параграфе.

Spring Security — это модуль, обеспечивающий аутентификацию, авторизацию и защиту от распространенных атак в приложениях. Он поддерживает как императивные, так и реактивные приложения, и является де-факто стандартом для обеспечения безопасности Spring-приложений.

Таким образом, мы рассмотрели основные положения экосистемы Spring Boot. Далее хотелось бы рассмотреть различные технологии доступа к данным, которые используются в бэкенд разработке с использованием Spring.

1.3 СУБД и технологии ORM

Для работы большинства приложений нужна база данных. Если рассматривать этот вопрос в контексте веб-системы для театральной индустрии, то в первую очередь нам хотелось бы хранить данные о произведениях, их авторах и жанрах. Если хранить эти данные в памяти приложения, то возникают проблемы с персистентностью – данные легко можно потерять при банальном перезапуске приложений, что является недопустимым. Для решения этой проблемы существуют базы данных и СУБД.

СУБД — это система управления базами данных. Это совокупность программных и лингвистических средств, которые обеспечивают создание, хранение, обработку и защиту данных в базах данных. База данных — это структурированное хранилище данных, которые могут быть реляционными или нереляционными. Реляционные базы данных представляют данные в виде таблиц, а нереляционные — в виде документов, графов, объектов и других форматов.

СУБД позволяют выполнять различные операции над данными, такие как:

- Создание, изменение и удаление баз данных и их объектов (таблиц, индексов, ограничений и т.д.);

- Вставка, обновление, удаление и выборка данных из баз данных с помощью языков запросов (например, SQL);

- Управление доступом к данным и их целостностью с помощью правил и ролей пользователей;

- Резервное копирование и восстановление данных в случае сбоев или потерь;

- Мониторинг и оптимизация производительности и нагрузки на базы данных.

СУБД используются в разных областях и приложениях, где требуется работа с разными объемами данных. Например:

- Разработка веб-сайтов и приложений для хранения и обработки данных о пользователях, контенте, заказах и т.д. – в том числе и данных о театральной индустрии;

- Анализ данных и наука о данных для извлечения знаний и прогнозирования из данных;

- BI и системы для поддержки принятия решений на основе данных;

- Образование и наука для обучения и исследования в области баз данных.

СУБД — это важный элемент информационных систем и технологий. Такая система позволяет эффективно управлять данными и использовать их для разных целей.

Реляционные СУБД — это системы управления базами данных, основанные на реляционной модели данных. Реляционная модель данных была сформулирована в 1970 году Эдгаром Коддом из IBM и предполагает, что данные представляются в виде таблиц (отношений), состоящих из строк (кортежей) и столбцов (атрибутов). Каждая строка в таблице имеет уникальный идентификатор (ключ), а каждый столбец имеет имя и тип данных. Таблицы могут быть связаны друг с другом посредством ключей, образуя реляционную схему базы данных.

Реляционные СУБД позволяют выполнять операции над данными, такие как выборка, вставка, обновление и удаление, с помощью специального языка запросов — SQL (Structured Query Language). SQL также поддерживает создание, изменение и удаление таблиц и других объектов базы данных, а также определение прав

доступа и ограничений целостности. SQL является стандартизированным языком для работы с реляционными базами данных и поддерживается большинством реляционных СУБД.

Реляционные СУБД имеют некоторые преимущества перед другими типами СУБД, такие как:

- Логическая и физическая независимость данных. Это означает, что структура и представление данных могут быть изменены без влияния на приложения, работающие с данными.

- Высокая надежность и производительность. Реляционные СУБД обеспечивают механизмы транзакций, резервного копирования, восстановления, индексирования и кэширования данных, а также оптимизацию запросов.

- Простота и гибкость. Реляционные СУБД позволяют легко создавать и модифицировать базы данных, а также выполнять сложные запросы к данным с помощью SQL.

- Поддержка различных типов данных. Реляционные СУБД могут хранить не только простые типы данных, такие как числа и строки, но и сложные типы данных, такие как изображения, документы, XML и JSON.

- Переносимость и совместимость. Реляционные СУБД могут работать на разных платформах и операционных системах, а также поддерживать различные стандарты и протоколы для обмена данными.

Реляционные СУБД являются доминирующим классом систем баз данных на современном рынке. Среди самых популярных и известных реляционных СУБД можно назвать Oracle Database, IBM DB2, Microsoft SQL Server, MySQL, PostgreSQL, Firebird и другие.

Для реализации веб-системы, разработке которой посвящена настоящая работа, была выбрана реляционная СУБД PostgreSQL. PostgreSQL — это открытая реляционная система управления базами данных, которая использует и расширяет язык SQL совместно с множеством функций, которые безопасно хранят и масштабируют самые сложные рабочие нагрузки с данными.

PostgreSQL поддерживает большое количество типов данных, таких как числа, строки, даты, времена, интервалы, массивы, JSON, XML и другие. Он также позволяет определять собственные типы данных и индексы. PostgreSQL поддерживает различные методы индексирования данных, такие как B-tree, hash, GiST, GIN и BRIN. PostgreSQL поддерживает полнотекстовый поиск, географические объекты (PostGIS), репликацию данных, партиционирование таблиц и другие возможности.

PostgreSQL предоставляет механизм транзакций для обеспечения атомарности, согласованности, изолированности и долговечности (ACID) операций над данными. PostgreSQL также поддерживает сохранение точек во времени (PITR), резервное копирование и восстановление данных. PostgreSQL обеспечивает защиту данных от несанкционированного доступа с помощью системы ролей и привилегий. PostgreSQL также поддерживает шифрование данных на уровне диска и на уровне протокола.

Для того, чтобы иметь доступ к данным из приложения, существуют различные подходы. Самый близкий к самой базе данных – использование SQL-запросов прямо в приложении. Но это не является удобным способом – дебаггинг запросов является весьма нетривиальной задачей, также приходится вручную парсить результаты запроса, чтобы получить нужные данные в представлении объектов приложения, что сильно увеличивает время разработки. Чтобы решить эту проблему, мы используем технологии ORM.

ORM — это технология программирования, которая позволяет работать с данными из реляционных баз данных с помощью объектно-ориентированного подхода. ORM создает «виртуальную объектную базу данных», которая может использоваться из языка программирования. ORM упрощает взаимодействие с базой данных, избавляя программистов от необходимости писать SQL-код и преобразовывать данные между разными формами. ORM также может обеспечить автоматическую синхронизацию объектов в памяти с данными в базе данных. Существуют разные реализации ORM для разных языков программирования и баз данных. ORM имеет свои преимущества, такие как удобство использования,

скорость разработки, абстракция от деталей базы данных, но также существуют недостатки в виде потери производительности и ограничения в выражении сложных запросов.

В качестве технологии доступа к данным мы используем библиотеку Spring Data JPA. В свою очередь, данная библиотека использует ORM Hibernate. Hibernate - это библиотека для языка программирования Java, которая реализует технологию ORM и позволяет работать с реляционными базами данных с помощью объектно-ориентированного подхода. Hibernate упрощает маппинг атрибутов классов на столбцы таблиц с помощью XML-файлов или аннотаций. Hibernate также поддерживает спецификацию JPA и предоставляет различные инструменты и расширения для работы с данными. Hibernate имеет открытый исходный код и распространяется под лицензией GNU LGPL. Hibernate используется во многих проектах и фреймворках, таких как Spring, Grails, Play и т.д.

JPA — это спецификация Java Persistence API, которая определяет стандартный способ работы с реляционными данными в Java-приложениях с помощью ORM. JPA предоставляет аннотации для метаданных, которые используются для описания отображения между объектами и таблицами базы данных. JPA также поддерживает язык JPQL для выполнения запросов к объектам, критерии API для динамического построения запросов и различные стратегии наследования и каскадирования. JPA не является реализацией ORM, а лишь спецификацией, которую реализуют разные библиотеки, такие как Hibernate, EclipseLink, OpenJPA и т.д.

Spring Data JPA - это модуль фреймворка Spring Data, который облегчает реализацию репозиториев на основе JPA. Spring Data JPA предоставляет расширенную поддержку для работы с данными JPA в приложениях Spring. Spring Data JPA позволяет сократить объем шаблонного кода, необходимого для выполнения простых запросов. Spring Data JPA стремится существенно улучшить реализацию слоя доступа к данным, уменьшив усилия до необходимого минимума. Разработчики могут писать интерфейсы репозиториев, включая пользовательские

методы поиска, а тем временем Spring автоматически создает реализацию для них. Spring Data JPA также поддерживает следующие функции:

- Продвинутая поддержка для создания репозиториев на основе Spring и JPA;
- Поддержка предикатов Querydsl и типобезопасных JPA-запросов;
- Прозрачный аудит доменных классов;
- Поддержка пагинации, динамического выполнения запросов, возможность интеграции пользовательского кода доступа к данным;
- Валидация запросов, аннотированных @Query, при запуске приложения;
- Поддержка XML-базированного маппинга сущностей;
- Конфигурация репозиториев на основе JavaConfig с помощью @EnableJpaRepositories.

Мы можем сделать вывод о том, что технологии СУБД позволяют удобно сохранять и извлекать данные, которые веб-приложение подготовит для пользователя. Также технологии ORM сильно упрощают использование базы данных в своих приложениях, позволяя структурировать код вне зависимости от используемой СУБД.

1.4 Язык программирования Kotlin

В описаниях технологий, которые описываются ранее, фигурирует язык программирования Java. Но в качестве языка программирования для веб-системы, разработка которой описывается в настоящей работе, был выбран Kotlin. Данный параграф в том числе направлен на объяснение, почему в языке Kotlin мы можем использовать эти библиотеки, а также на разъяснение преимуществ использования данного языка.

Язык программирования Kotlin — это статически типизированный, объектно-ориентированный язык, который работает поверх Java Virtual Machine и разрабатывается компанией JetBrains с 2010 года. Он создан как альтернатива Java, которая была слишком многословной и неудобной для разработки. Kotlin полностью совместим с Java и может использоваться для разных платформ, включая Android, серверную и веб-разработку. Kotlin имеет множество особенностей, которые делают его лаконичным, выразительным и безопасным

языком. Например, более хорошо поддерживает функциональное программирование, null-безопасность, функции-расширения, дата-классы, корутины и многое другое. Kotlin также имеет большое сообщество разработчиков и официальную поддержку от Google для Android.

Kotlin имеет ряд преимуществ по сравнению с Java, которые делают его более удобным и современным языком для разработки. Некоторые из них:

- Kotlin более лаконичен и выразителен, чем Java. Он позволяет писать меньше кода для достижения тех же результатов, что упрощает чтение и поддержку кода.

- Kotlin обеспечивает null-безопасность, то есть он не допускает присваивания или возврата null значениям, которые не могут быть null. Это помогает избежать NullPointerException, одной из самых распространенных ошибок в Java.

- Kotlin имеет мощный механизм расширения функций, который позволяет добавлять новые функции к существующим классам без наследования или декорирования. Это улучшает читаемость и модульность кода.

- Kotlin поддерживает корутины, которые являются легковесными потоками для асинхронного и параллельного программирования. Они позволяют писать простой и последовательный код для решения сложных задач с использованием многопоточности и неблокирующего ввода-вывода.

- Intellij IDEA имеет встроенный конвертер из Java в Kotlin, который позволяет легко переводить существующий код на Java в код на Kotlin. Это облегчает миграцию проектов и интеграцию с библиотеками и фреймворками на Java.

В общем и целом, язык программирования Java высоко ценится в среде бэкенд разработки за его сильно развитую экосистему, стабильность и бесперебойность. Но наш проект является новым. Хоть Java также используется в среде стартапов, было бы логично выбрать в качестве технологической платформы что-то одновременно и современное, и развитое. Язык программирования Kotlin отлично подходит под эту роль, так как он полагается на JVM и библиотеки

программирования Java, что является стабильной и развитой основой, но также предлагает элегантный, современный и лаконичный синтаксис, который упрощает разработку и чтение кода.

1.5 Об архитектуре приложений

Для грамотной разработки любого бэкенда следует обратить внимание на корректное проектирование системы. Чтобы спроектировать систему, нужно изучить архитектурные паттерны проектирования. В данном параграфе мы рассмотрим некоторые из них и выберем те, что будут использоваться для веб-системы для театральной индустрии.

Одним из самых популярных паттернов является шаблон MVC. Паттерн MVC — это архитектурный паттерн или паттерн проектирования, который делит модули на три группы: модель, представление и контроллер. Это позволяет отделить бизнес-логику (модель) от её визуализации (представление) и повысить возможность повторного использования кода.

Модель (Model) предоставляет данные и реагирует на команды контроллера, изменяя своё состояние. Представление (View) отвечает за отображение данных модели пользователю, реагируя на изменения модели. Контроллер (Controller) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Паттерн MVC широко используется в веб-разработке на разных языках программирования, в том числе на Java. Например, фреймворки Spring MVC и Struts основаны на паттерне MVC.

Несмотря на популярность паттерна MVC, мы не используем его в разработке веб-системы, так как данный шаблон предполагает, что фронтенд и бэкенд будут единым целым. В нашем случае бэкенд – это обязательно самостоятельная единица, которую могут использовать разные клиенты, в том числе и фронтенд. Поэтому в разработке используется стиль RESTful API.

RESTful API — это архитектурный стиль для интерфейса программирования приложений (API), который использует HTTP-запросы для доступа и использования данных. Эти данные могут быть использованы для операций чтения,

обновления, создания и удаления ресурсов, которые обозначаются как методы GET, PUT, POST и DELETE. RESTful API соответствует принципам архитектуры REST, которая была определена в 2000 году учёным Роем Филдингом в его докторской диссертации. Принципы архитектуры REST включают следующие ограничения: единый интерфейс, разделение клиента и сервера, отсутствие состояния, кеширование, система, состоящая из слоёв. RESTful API обеспечивает гибкость, независимость, производительность и масштабируемость при коммуникации между различными приложениями и компонентами.

Фреймворк Spring Boot отлично подходит для реализации RESTful API. А в качестве архитектуры кодовой базы Spring может предоставить Clean Architecture – фреймворк содержит средства, которые позволяют реализовать данный паттерн наиболее полно.

Clean Architecture – это шаблон проектирования программного обеспечения, который следует концепциям чистого кода и реализует принципы SOLID. Он состоит из коллекции лучших практик проектирования, которые помогают минимизировать зависимости внутри системы.

Clean Architecture можно изобразить диаграммой, которая показа на рисунке 1.

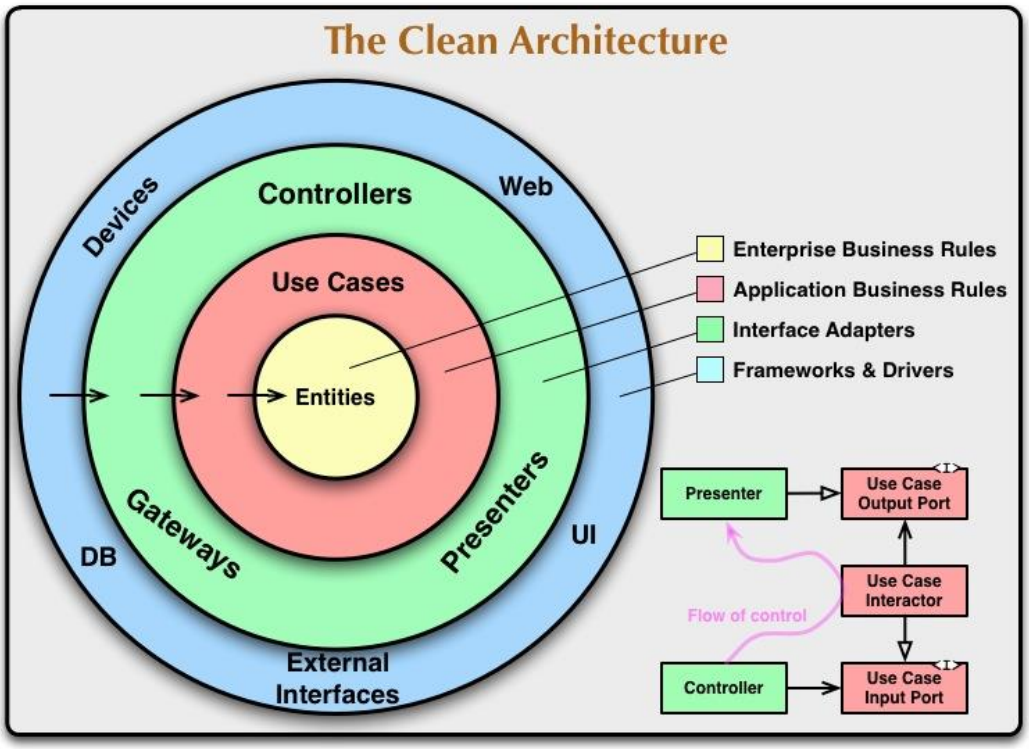


Рисунок 1. Диаграмма Clean Architecture

На вышеуказанной диаграмме видно, что система разделена на слои. Каждый слой имеет свою ответственность и зависит только от внутренних слоев. Самый внутренний слой — это Entities, которые представляют доменную и бизнес-логику. Они являются общими для всего приложения и наименее подвержены изменениям из-за внешних факторов. Следующий слой — это Use Cases, которые реализуют прикладную логику или сценарии использования приложения. Они координируют поток данных между сущностями и внешними интерфейсами. Слой Interface Adapters содержит адаптеры, которые преобразуют данные из формата, используемого во внешних интерфейсах, в формат, используемый внутри приложения. Это могут быть контроллеры, представления, шлюзы или презентеры. Самый внешний слой — это Frameworks and Drivers, который содержит инструменты и фреймворки, такие как базы данных, веб-серверы, UI-фреймворки и т.д. Они обеспечивают реализацию деталей, необходимых для работы приложения.

Преимущества Clean Architecture заключаются в том, что она делает приложение независимым от фреймворков, тестируемым, независимым от UI,

независимым от баз данных и независимым от любых внешних систем. Это повышает гибкость и расширяемость системы.

Так как мы говорим о Clean Architecture, нам следует рассмотреть принципы SOLID, на которые сильно полагается данный паттерн. Принципы SOLID — это пять принципов объектно-ориентированного программирования и проектирования, которые были сформулированы Робертом Мартином в начале 2000-х годов. SOLID — это акроним, который состоит из следующих букв:

- S - Single responsibility principle (Принцип единственной ответственности). Этот принцип гласит, что каждый класс должен иметь только одну причину для изменения и выполнять только одну задачу.
- O - Open-closed principle (Принцип открытости/закрытости). Этот принцип гласит, что классы должны быть открыты для расширения, но закрыты для модификации. Это означает, что мы можем добавлять новую функциональность к существующим классам, не изменяя их код.
- L - Liskov substitution principle (Принцип подстановки Лисков). Этот принцип гласит, что подклассы должны быть взаимозаменяемы с их суперклассами. Это означает, что мы можем использовать объекты подклассов вместо объектов суперклассов, не нарушая корректность программы.
- I - Interface segregation principle (Принцип разделения интерфейса). Этот принцип гласит, что клиенты не должны зависеть от методов интерфейсов, которые они не используют. Это означает, что мы должны разбивать большие интерфейсы на меньшие и более специфичные, чтобы уменьшить связанность между классами.
- D - Dependency inversion principle (Принцип инверсии зависимостей). Этот принцип гласит, что классы должны зависеть от абстракций, а не от конкретных реализаций. Это означает, что мы должны использовать интерфейсы или абстрактные классы в качестве параметров или полей классов, а не конкретные классы.

Цель принципов SOLID — это создание такого кода, который будет легко читать, понимать, тестировать, изменять и расширять. Принципы SOLID помогают избегать таких проблем, как дублирование кода, сильная связанность, сложность архитектуры и низкая гибкость.

Для реализации данных принципов средством Spring Boot можно использовать метод разделения интерфейсов и имплементаций, которые будут задействованы механизмами Dependency Injection и контейнером Inversion of Control.

Dependency Injection (DI) - это шаблон проектирования, который используется для реализации инверсии управления (IoC). Он позволяет создавать зависимые объекты вне класса и предоставлять их классу различными способами. Используя DI, мы перемещаем создание и связывание зависимых объектов за пределы класса, который от них зависит. Шаблон Dependency Injection включает в себя четыре роли: сервисы, клиенты, интерфейсы и инжекторы.

Сервис — это любой класс, который содержит полезную функциональность. Клиент — это любой класс, который использует сервисы. Любой объект может быть сервисом или клиентом; названия относятся только к роли, которую объекты играют во внедрении. Интерфейс — это абстракция, которая определяет контракт между сервисом и клиентом. Инжектор — это внешний код (например, контейнер), который создает сервисы и внедряет их в клиентов через конструктор, свойство или метод.

Цель Dependency Injection - разделить ответственность за создание и использование объектов, что приводит к слабой связанности программ. Это облегчает замену одной реализации сервиса другой, а также упрощает тестирование кода с использованием моков или заглушек.

IoC контейнер — это специальный инструмент, который позволяет реализовать принцип инверсии управления в объектно-ориентированном программировании. Инверсия управления означает, что объекты не создают и не управляют своими зависимостями (другими объектами), а получают их извне. IoC контейнер берет на себя задачу создания, связывания и предоставления объектов

по запросу. Это упрощает код, делает его более модульным, тестируемым и расширяемым. То есть, фреймворк берёт на себя работу по управлению жизненным циклом объектов и их зависимостями. Такой подход позволяет целиком сконцентрироваться на бизнес логике, не отвлекаясь на написание ручного внедрения зависимостей.

2 ПРОЕКТИРОВАНИЕ ВЕБ-СИСТЕМЫ

2.1 Проектирование базы данных

Для построения процесса разработки бэкенда веб-системы следует уделить должное внимание вопросу проектирования всех узлов системы, начиная от базы данных, заканчивая архитектурой кодовой базы. В данном параграфе рассматривается проектирование сущностей базы данных.

Процесс проектирования реляционной базы данных состоит из нескольких этапов:

- Концептуальное (инфологическое) проектирование — построение семантической модели предметной области, то есть информационной модели наиболее высокого уровня абстракции. Такая модель создаётся без ориентации на какую-либо конкретную СУБД и модель данных. Обычно используются графические нотации, подобные ER-диаграммам.
- Логическое (даталогическое) проектирование — создание схемы базы данных на основе конкретной модели данных, например, реляционной модели данных. Для реляционной модели данных даталогическая модель — набор схем отношений, обычно с указанием первичных ключей, а также «связей» между отношениями, представляющих собой внешние ключи.
- Физическое проектирование — определение физических характеристик базы данных, таких как способы хранения, индексирования и доступа к данным. На этом этапе учитывается специфика конкретной СУБД и аппаратного обеспечения.

Проектирование базы данных направлено на обеспечение хранения в БД всей необходимой информации, возможности получения данных по всем необходимым запросам, сокращения избыточности и дублирования данных и обеспечения целостности базы данных.

Начнём с этапа концептуального проектирования. Для совершения этого процесса нужно выделить абстрактные сущности, на которые опирается предметная область приложения, а также абстрактные связи между ними. Удалось выделить 4 сущности:

1. Пьеса: каждая пьеса имеет от одного и более авторов; каждая пьеса имеет название; некоторые пьесы могут иметь название на языке оригинала текста; пьесы могут иметь год написания, если он известен; пьеса может иметь текстовое описание; пьеса может иметь город, в котором она написана (если известен); пьеса может иметь ссылку на текст; пьеса имеет жанр; пьеса может иметь изображение обложки.

2. Автор: автор имеет имя; автор имеет пьесы; автор может иметь страну; автор может иметь город.

3. Жанр: жанр имеет название.

4. Отзыв на пьесу: отзыв имеет текст; отзыв ссылается на пьесу.

Можно также выделить пятую сущность в виде пользователя системы, но на данный момент рабочий прототип веб-системы не имеет возможности авторизации и аутентификации, а лишь имеет возможность просмотра данных о произведениях. Возможность регистрации пользователей и разделения их возможности в зависимости от роли планируется добавить в следующих итерациях разработки приложения.

Проанализировав сущности веб-системы, можно приступить к разработке концептуальной диаграммы. Данный процесс можно произвести с помощью онлайн-приложения draw.io. Draw.io — это мощный и гибкий инструмент для создания различных видов диаграмм. С его помощью можно:

- Создавать блок-схемы, организационные схемы, ER-диаграммы, логотипы, диаграммы процессов, сетевые диаграммы, UML и многое другое.

- Хранить свои данные в разных хранилищах, например в Google Drive, OneDrive, Dropbox, GitHub или GitLab.

- Импортировать и экспортировать файлы в разных форматах, таких как .vsdx, Gliff и Lucidchart.

- Использовать готовые шаблоны и примеры диаграмм для экономии времени.

Рассмотрев возможности данного программного обеспечения, можно прийти к выводу о том, что draw.io отлично подходит для создания моделей данных, в том

числе и концептуальной. Интуитивно понятный интерфейс позволяет создавать диаграммы достаточно быстро и комфортно.

С помощью вышеуказанного анализа сущностей была создана концептуальная диаграмма модели данных, изображённая на рисунке 2.

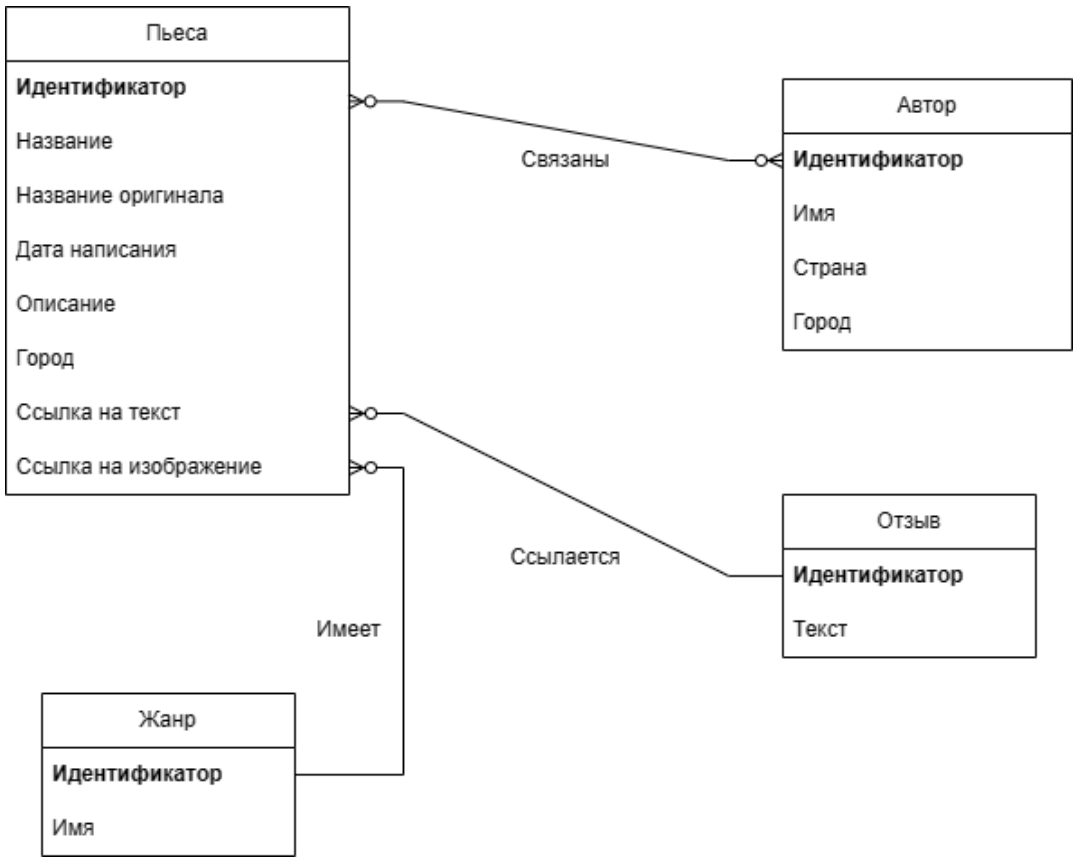


Рисунок 2. Концептуальная диаграмма модели данных

Хотелось бы отметить, что концептуальное проектирование базы данных является действительно полезным и важным этапом разработки любой системы, которая связана с хранением и обработкой данных. Концептуальное проектирование помогает уточнить цель и требования к базе данных. Оно облегчает логическое и физическое проектирование базы данных, так как предоставляет четкую и независимую от реализации структуру данных. Также этот тип проектирования помогает определить ограничения, которые накладываются на данные, тем самым повышая надежность и эффективность базы данных.

Следующим этапом после концептуального проектирования идёт логическое (даталогическое) проектирование. Для логического проектирования базы данных нужно выполнить следующие шаги:

- Выбрать модель данных, на которой будет основана схема базы данных, например, реляционную модель данных.
- Преобразовать концептуальную модель данных в логическую модель данных с учетом специфики выбранной модели данных. Для этого можно использовать формальные правила или специальные инструменты.
- Определить схемы отношений (таблиц) и их атрибуты (поля), а также указать первичные ключи и внешние ключи для обеспечения связей между отношениями.
- Провести нормализацию данных.

В качестве модели данных была выбрана реляционная модель. Также хотелось бы уделить должное внимание нормализации данных. Нормализация отношений — это процесс преобразования схемы отношений (таблиц) таким образом, чтобы избежать избыточности и дублирования данных, а также потенциальных аномалий при вставке, обновлении или удалении данных. Нормализация отношений основана на следующих принципах:

- Каждое отношение должно иметь первичный ключ, который однозначно идентифицирует каждую запись в отношении.

- Каждый атрибут в отношении должен быть атомарным, то есть не содержать несколько значений.

- Каждый атрибут в отношении должен зависеть от первичного ключа целиком, а не от его части.

- Каждый атрибут в отношении должен зависеть только от первичного ключа, а не от других атрибутов.

- Каждое отношение должно содержать только те атрибуты, которые связаны с темой этого отношения.

Для проверки соблюдения этих принципов используются понятия нормальных форм. Нормальная форма — это степень соответствия схемы отношений определенным правилам нормализации. Существует шесть нормальных форм: первая (1NF), вторая (2NF), третья (3NF), нормальная форма Бойса-Кодда (BCNF), четвертая (4NF) и пятая (5NF). Чем выше нормальная форма, тем меньше избыточность и аномалии в данных. Однако высокие нормальные формы могут приводить к увеличению числа таблиц и сложности запросов. Поэтому при проектировании базы данных нужно выбирать оптимальный уровень нормализации, учитывая требования к производительности и гибкости базы данных.

Чаще всего ограничиваются именно третьей нормальной формой. Третья нормальная форма (3NF) — это одна из возможных нормальных форм отношения в реляционной базе данных. 3NF была изначально сформулирована Э. Ф. Коддом в 1971 году. Третья нормальная форма предполагает, что каждый столбец, не являющийся ключом, должен зависеть только от столбца, который является ключом, то есть должна отсутствовать транзитивная функциональная зависимость. Транзитивная функциональная зависимость - это ситуация, когда атрибут C транзитивно зависит от атрибута A, если атрибут C зависит от атрибута B, а атрибут B зависит от атрибута A. Например, если в таблице есть столбцы CourseId, Course и TeacherId, и известно, что CourseId определяет Course и TeacherId, а TeacherId определяет Teacher, то можно сказать, что Teacher транзитивно зависит от CourseId. Для приведения таблицы к 3NF нужно вынести в отдельную таблицу те атрибуты, которые находятся в транзитивной зависимости от ключа. Например, в данном случае нужно создать таблицу Teachers с атрибутами TeacherId и Teacher и связать ее с таблицей Courses по атрибуту TeacherId. Чтобы грамотно привести модель данных к третьей нормальной форме, нужно использовать связи между таблицами.

Отношение один ко многим означает, что одна запись в одной таблице может быть связана с несколькими записями в другой таблице. Приведём пример на основе предметной области настоящей работы. В базе данных, которая хранит информацию о пьесах и отзывах, каждая пьеса может иметь много отзывов, но

каждый отзыв может иметь только одну пьесу. Чтобы создать такое отношение, нужно добавить поле с первичным ключом одной таблицы (например, id_пьесы) в другую таблицу (например, отзыва) в качестве внешнего ключа.

Отношение многие ко многим означает, что несколько записей в одной таблице могут быть связаны с несколькими записями в другой таблице. Например, в базе данных, которая хранит информацию о пьесах и авторах, каждая пьеса может иметь много авторов, и каждый автор мог работать над многими пьесами. Чтобы создать такое отношение, нужно создать третью таблицу (например, пьесы_авторы), которая будет хранить пары значений внешних ключей из двух других таблиц (например, id_пьесы и id_автора).

С помощью знаний о логическом проектировании и нормализации данных удалось разработать ER-диаграмму логической схемы данных. Было использовано ПО draw.io, как и в случае с концептуальным проектированием. Результат работы показан на рисунке 3.

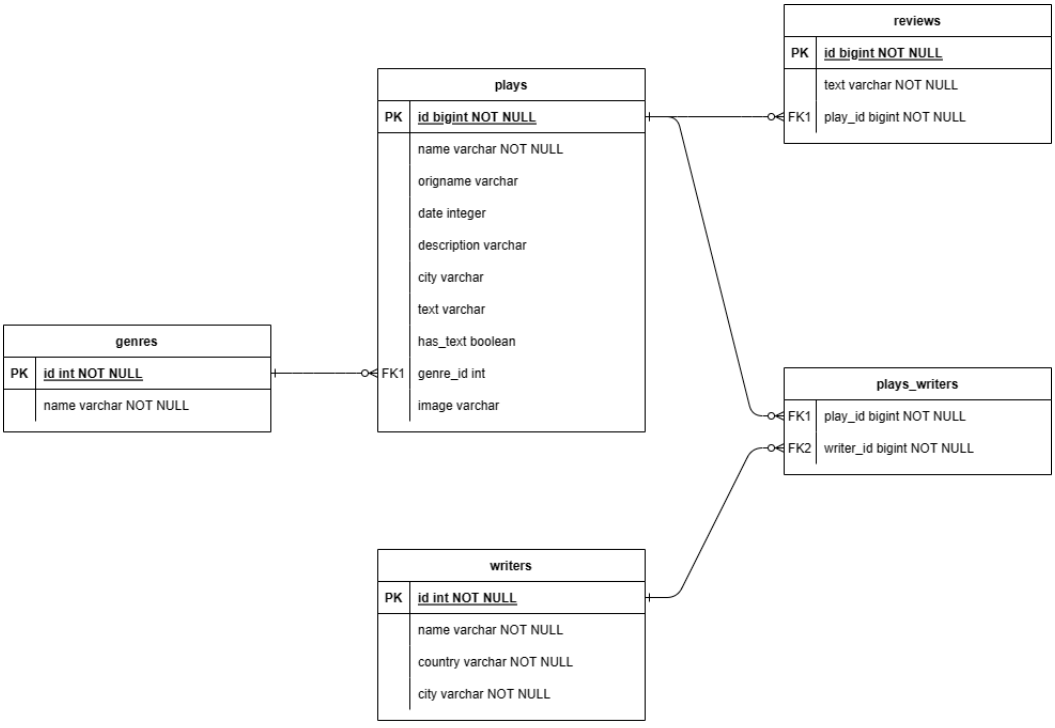


Рисунок 3. ER-диаграмма модели данных

То есть, нам удалось сделать нашу модель данных менее абстрактной и более близкой к физической форме хранения в базе данных. Нам удалось нормализовать данные, выделяя первичные ключи и связи между сущностями.

Как было рассмотрено в примерах ранее, между сущностями автор (writers) и пьесы (plays) существует связь многие-ко-многим, которая реализована с помощью промежуточной таблицы plays_writers. Также связь между пьесой и отзывом связана отношением один-ко-многим с помощью добавления поля play_id в таблицу reviews. Таким же образом существует связь между пьесой и жанром, но поле genre_id добавлено в таблицу plays.

Физическое проектирование базы данных обычно включает в себя следующие шаги:

- Создание набора реляционных таблиц и ограничений для них на основе информации, представленной в глобальной логической модели данных.
- Определение конкретных структур хранения данных и методов доступа к ним, обеспечивающих оптимальную производительность СУБД.
- Разработка средств защиты создаваемой системы.

Физическое проектирование базы данных веб-системы для театральной индустрии производится с помощью ПО Liquibase. Liquibase - это инструмент для управления изменениями базы данных, который позволяет разработчикам автоматизировать процесс изменения схемы базы данных и интегрировать его в процесс непрерывной поставки (CI/CD). Liquibase поддерживает более 50 баз данных и позволяет определять изменения в SQL, XML, JSON или YAML. Liquibase также предоставляет функции проверки качества для проверки кода базы данных на соответствие стандартам безопасности и соблюдения нормативных требований. Liquibase можно использовать как командную утилиту, работающую на macOS, Windows, Unix и Linux.

Мы используем Liquibase как зависимость нашего проекта. Также в качестве языка разметки был выбран XML из-за большей наглядности и удобства использования. Liquibase позволяет автоматически обновлять структуру базы данных после произведения изменений. Каждое изменение помещается в новый файл. При ошибках проектирования можно легко сделать откат изменений.

Файл изменений Liquibase имеет формат .xml и содержит корневой тэг databaseChangelog. Любое атомарное изменение структуры базы данных

помещается в тэг changeSet. Внутри него мы можем помещать тэги, связанные с проектированием физической структуры данных – создание таблиц, добавление атрибутов таблиц, обновление и удаление элементов структуры. Рассмотрим использование Liquibase на примере создания таблицы для пьес (plays). На рисунке 4 показан changeset, который используется для первоначального создания отношения.

```
<changeSet id="create-table-plays" author="bee-joo">
    <createTable tableName="plays">
        <column name="id" autoIncrement="true" type="bigint">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="name" type="varchar(255)">
            <constraints nullable="false"/>
        </column>
        <column name="origname" type="varchar(255)"/>
        <column name="date" type="int"/>
        <column name="description" type="varchar(2000)"/>
    </createTable>
</changeSet>
```

Рисунок 4. Changeset таблицы plays

Как мы можем увидеть, changeset может иметь свой идентификатор и автора. Это нужно для более прозрачного процесса добавления и отката изменений. Для создания таблицы используется тэг createTable, а имя таблицы добавляется в атрибут tableName. Все колонки таблицы добавляются путём использования тэга column, где в атрибутах указываются названия колонок и их типы данных. Также можно накладывать ограничения с помощью тэга constraints.

Но созданный таким образом xml-файл не считывается утилитой автоматически. Для миграции изменений следует отредактировать файл *db.changelog-master.xml*. С помощью тэга include можно включать файлы миграций в общий пул изменений, указывая в атрибуте file путь к файлу. Так, общий файл с изменениями, которые совершались во время разработки проекта, показан на рисунке 5.

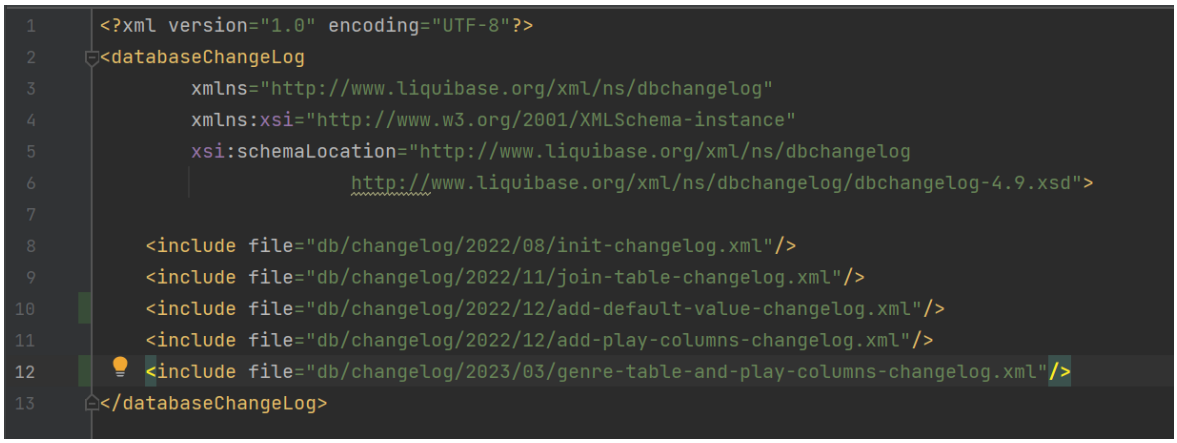


Рисунок 5. Общий файл changelog

Таким образом, Liquibase позволяет несколько абстрагироваться от вида СУБД в вопросе физического проектирования базы данных. С помощью различных языков разметки имеется возможность удобно контролировать все изменения, которые происходят в схеме данных. При добавлении изменений Liquibase автоматически инициализирует их при запуске приложения. С помощью данной утилиты нам удалось реализовать физическое проектирование базы данных на основе логической схемы. Все файлы миграций доступны в GitHub-репозитории и расположены в каталоге *src/main/resources/db/changelog*.

2.2 Проектирование архитектуры приложения

Рассмотрев проектирование базы данных, перейдём к процессу проектирование архитектуры кодовой базы приложения. Здравствуйте, это Bing. Архитектура приложения — это способ организации компонентов и функций приложения, чтобы обеспечить его эффективность, масштабируемость, безопасность и поддержку. Архитектура приложения важна, потому что она определяет, как приложение будет работать с данными и другими системами, а также определяет взаимодействие внутренних компонентов системы. Архитектура приложения также помогает контролировать сложность создаваемого приложения и получать преимущества, такие как:

- Масштабируемость – позволяет расширять систему и улучшать ее производительность благодаря добавлению новых модулей.

- Изменяемость – изменение одной части программы не требует вмешательства в другие.

- Тестируемость – модули программы можно отделить и отдельно протестировать.

Как уже было сказано ранее, при разработке используется паттерн Clean Architecture. Далее хотелось бы отобразить те слои, которые используются в приложении.

1. Слой Entity. В приложении Spring — это часть архитектуры, которая отвечает за представление данных из базы данных в виде объектов Java. Классы, которые помечены аннотацией @Entity, являются сущностями и могут быть сопоставлены с таблицами в реляционной базе данных.

2. Слой Repository. Он отвечает за взаимодействие с базой данных и предоставляет методы для работы с данными. Слой Repository обычно состоит из интерфейсов, которые наследуются от JpaRepository или других интерфейсов фреймворка Spring Data. Spring Data автоматически генерирует реализацию этих интерфейсов при запуске приложения. Слой Repository также помечается аннотацией @Repository. Этот слой является частью шаблона проектирования DAO (Data Access Object) или Repository, который разделяет бизнес-логику от логики доступа к данным.

3. Слой Service. Этот слой отвечает за бизнес-логику приложения и выполняет различные операции над данными, полученными из слоя Repository. Слой Service обычно состоит из классов, которые помечены аннотацией @Service, которая является специализацией аннотации @Component и позволяет легко интегрировать слой Service с другими компонентами Spring, такими как контроллеры или аспекты. Слой Service является частью шаблона проектирования Business Delegate, который разделяет бизнес-логику от логики представления.

4. Слой Controller. Это слой, который отвечает за обработку HTTP-запросов от клиентов и возвращает HTTP-ответы с данными. Слой Controller обычно состоит из классов, которые помечены аннотацией @Controller. В этом слое также можно использовать аннотации, такие как @RequestMapping, @RequestParam, @PathVariable, @ModelAttribute и другие, для определения маппингов URL, параметров запроса, переменных пути, моделей данных и других аспектов

обработки запросов. Слой Controller является частью шаблона проектирования Front Controller или MVC (Model-View-Controller), который разделяет логику представления от логики контроллера.

Итак, в рамках проектирования архитектуры нам удалось выделить четыре слоя приложения. Слой Entity представляет собой слой модели данных, слой предметной области приложения. Слой Repository позволяет нам абстрагироваться от конкретной реализации СУБД в вопросах обращения к данным. Слой Service отделяет бизнес логику от остальных компонентов системы. А слой Controller позволяет создавать контракт для взаимодействия с другими системами. Нам удалось выбрать масштабируемую и тестируемую архитектуру, в которой каждый слой выполняет свою конкретную задачу и может взаимодействовать с другими слоями.

Но также хотелось бы отметить, что зачастую слой Controller принимает и отдаёт данные не в том виде, в котором они существуют в слое Entity. Например, мы бы не хотели при запросе получения данных о пользователе отдавать клиенту все его данные, в том числе и пароль. Для решения этой проблемы можно использовать паттерн Data Transfer Object (DTO). DTO — это объект, который используется для передачи данных между слоями или подсистемами приложения. DTO обычно не содержит бизнес-логики, а только хранит и предоставляет доступ к данным. DTO может отличаться от модели по структуре или содержанию, чтобы оптимизировать передачу данных по сети или адаптировать данные к нуждам клиента. DTO также может инкапсулировать логику сериализации или десериализации данных в определенный формат, например JSON или XML.

То есть, в разработке с использованием Spring является хорошей практикой разделять сущность и представление. В нашем приложении каждый объект модели имеет два класса для коммуникации с другими системами. Классы View – это то, что видит пользователь системы, а классы DTO – это ожидаемый запрос от пользователя.

Таким образом, нам удалось спроектировать архитектуру приложения, разделив его на четыре самостоятельных слоя. Такой подход упрощает

тестирование и независимость компонентов системы. Также нам удалось решить проблему отображение модели данных для клиента и создания контракта для получения данных от пользователя.

3 РАЗРАБОТКА БЭКЕНДА

3.1 Инициализация проекта и управление зависимостями

В процессе разработки бэкенда веб-системы первым шагом всегда является инициализация проекта. Для каждого стэка технологий этот процесс является индивидуальным, поэтому рассмотрим создание проекта на фреймворке Spring Boot.

Для создания проекта можно использовать удобную утилиту Spring Initializr. Spring Initializr — это сервис, который позволяет генерировать проекты на основе Spring Boot с различными настройками и зависимостями. Вы можете использовать Spring Initializr через веб-интерфейс или через интеграцию с IDE, такой как IntelliJ IDEA. Spring Initializr поддерживает разные языки программирования (Java, Kotlin, Groovy), системы сборки (Maven, Gradle) и версии Spring Boot. Графический интерфейс облегчает процесс инициализации проекта и делает его более наглядным. На рисунке 6 показано главное окно утилиты.

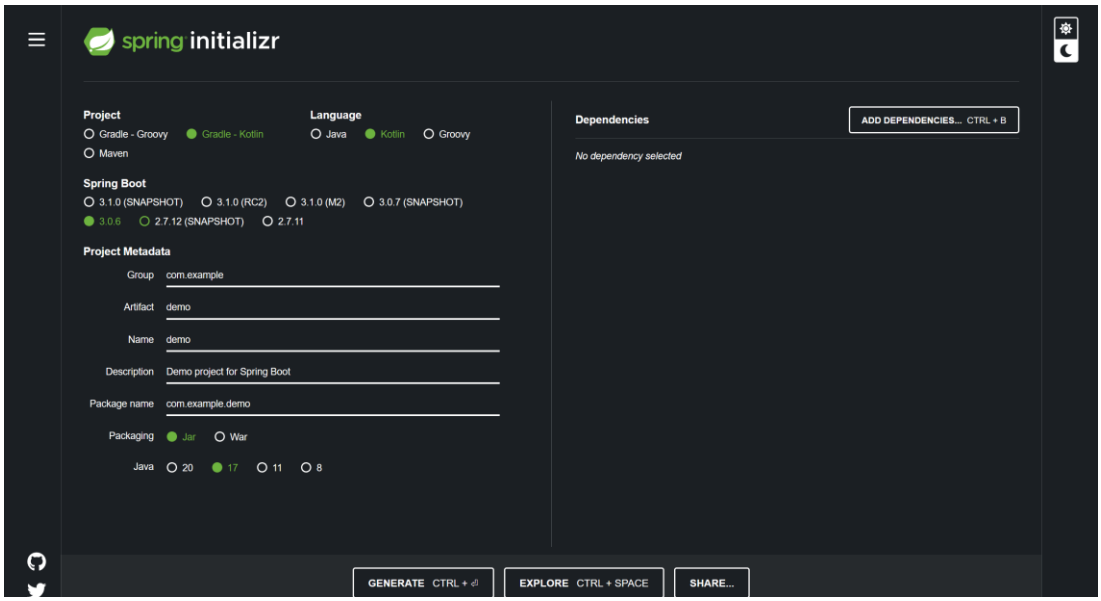


Рисунок 6. Главное окно Spring Initializr

На главном окне мы можем выбрать язык программирования, систему сборки и версию фреймворка. Также можно заполнить метаданные, например имя проекта и описание. Зависимости можно добавлять с помощью кнопки Add Dependencies. Окно добавления зависимостей проекта показано на рисунке 7.

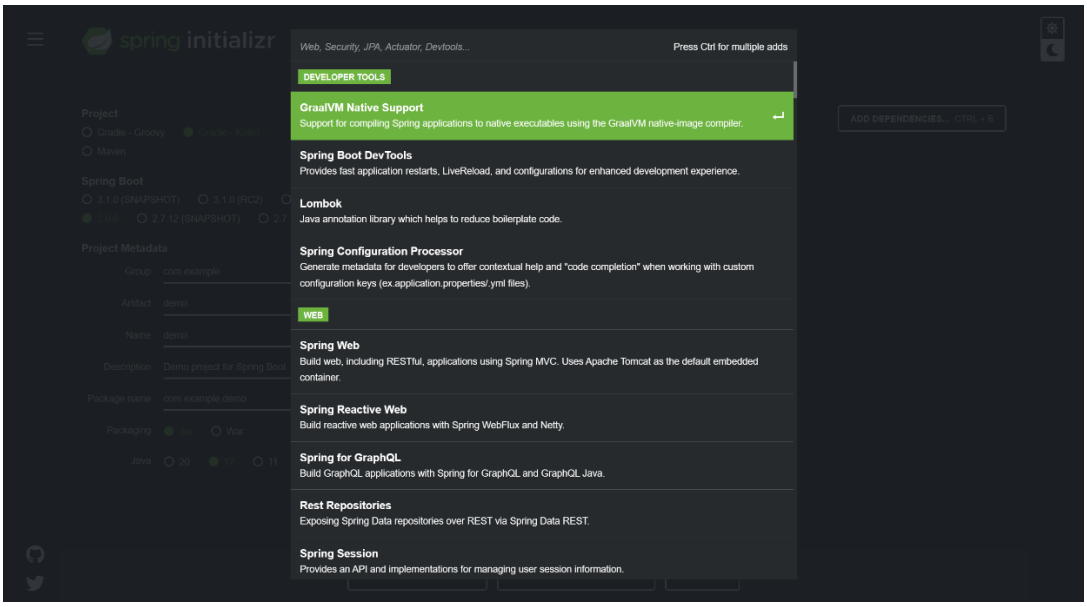


Рисунок 7. Добавление зависимостей

Spring Initializr предоставляет внушительный список библиотек, которые можно добавить в свой проект. Есть библиотеки для работы с данными, драйверы для различных баз данных, библиотеки для тестирования, безопасности и многие другие. Из зависимостей главной является Spring Web – это опора проекта, которая предоставляет функции для создания веб приложений, например встроенный контейнер сервлетов (по умолчанию Apache Tomcat), аннотации для контроллеров. Также для доступа к данным мы выбрали библиотеку Spring Data JPA, о которой было рассказано ранее. Так как в качестве СУБД была выбрана PostgreSQL, нам нужно добавить соответствующий драйвер. Также ранее был описан процесс физического моделирования структуры базы данных с помощью библиотеки Liquibase. Эту зависимость также можно добавить с помощью Spring Initializr. Окно с выбранными зависимостями изображено на рисунке 8.

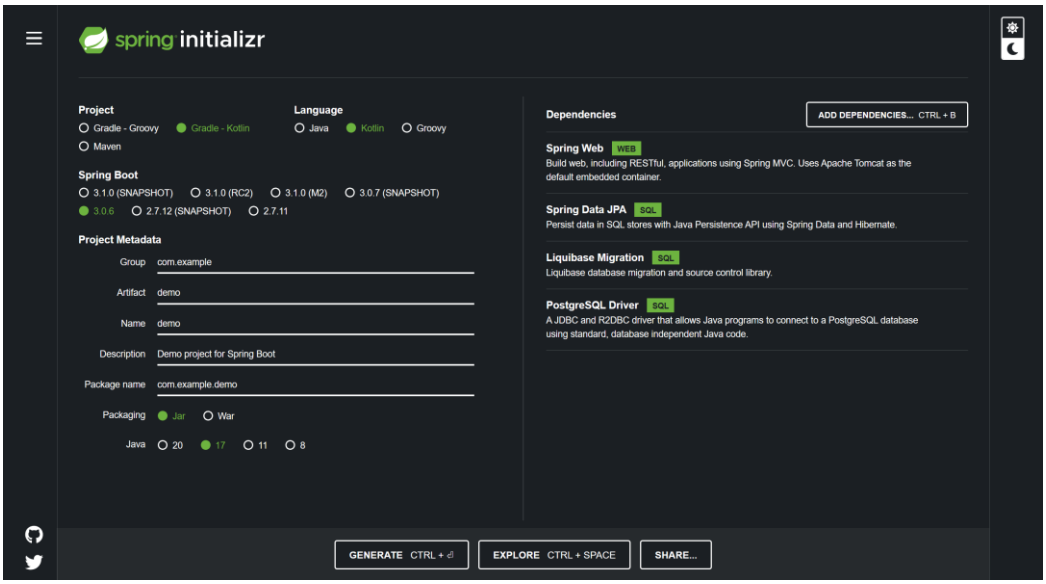


Рисунок 8. Зависимости проекта

С помощью кнопки Generate полученный проект можно загрузить в виде архива и открыть в любом редакторе кода. Такой же пользовательский интерфейс для генерации проекта предлагает и интегрированная среда разработки Intellij IDEA, в которой происходит написание программного кода проекта. Структура сгенерированного проекта показана на рисунке 9.

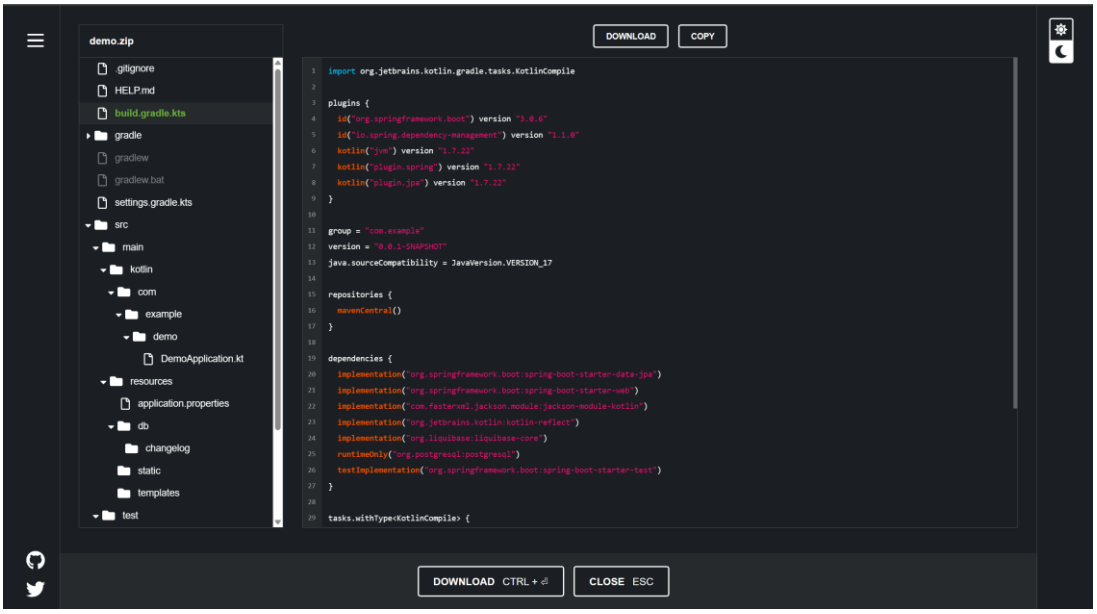


Рисунок 9. Проект и файл с зависимостями

Но Spring Initializr предоставляет далеко не все зависимости, которые могут пригодиться для проекта. Например, он не предоставляет библиотеки для генерации документации по стандарту Open API. Рассмотрим процесс добавления

собственных зависимостей с помощью системы сборки Gradle, которая была выбрана в качестве системы сборки проекта для театральной индустрии.

Gradle – это система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но предоставляющая DSL на языках Groovy и Kotlin вместо традиционного XML-представления конфигурации проекта. Gradle использует направленный ациклический граф для определения порядка выполнения задач и поддерживает каскадную модель разработки, управление зависимостями и веб-визуализацию сборки. Gradle широко используется для разработки и развертывания Java, Groovy, Scala и Android приложений, а также других языков программирования.

Процесс использования системы сборки Gradle мы рассмотрим на примере добавления библиотеки SpringDoc. SpringDoc – это библиотека для автоматизации генерации документации API, подробнее будет рассмотрено позднее. Зависимости проекта помещаются в файл *build.gradle.kts*. Именно там мы добавим нужные строки для того, чтобы мы могли собрать наш проект со всеми полезными библиотеками. В этом файле есть секция plugins, в которой описаны нужные для системы сборки плагины – они помогают обрабатывать работу и сборку библиотек корректно. SpringDoc требует плагин для своей работы. На рисунке 10 показана секция плагинов. Плагин для SpringDoc находится на 7 строчке (нумерация отображена слева).

```
3   plugins { this: PluginDependenciesSpecScope
4       val kotlinVersion = "1.8.0"
5       id("org.springframework.boot") version "2.7.4"
6       id("io.spring.dependency-management") version "1.0.11.RELEASE"
7       id("org.springdoc.openapi-gradle-plugin") version "1.6.0"
8       kotlin("plugin.lombok") version "1.8.0"
9       id("io.freefair.lombok") version "5.3.0"
10      kotlin("jvm") version kotlinVersion
11      kotlin("plugin.spring") version kotlinVersion
12      kotlin("plugin.jpa") version kotlinVersion
13      kotlin("plugin.allopen") version kotlinVersion
14      kotlin("kapt") version kotlinVersion
15  }
```

Рисунок 10. Секция плагинов

Но плагины — это вспомогательные частицы библиотек. Сами зависимости добавляются в секцию dependencies. На рисунке 11 показаны все добавленные зависимости нашего проекта. Сами зависимости для библиотеки SpringDoc находится на 37 и 38 строчках.

```
30  dependencies { this: DependencyHandlerScope
31      implementation("org.springframework.boot:spring-boot-starter-data-jpa")
32      implementation("org.springframework.boot:spring-boot-starter-web")
33      implementation("com.fasterxml.jackson.module:jackson-module-kotlin")
34      implementation("org.jetbrains.kotlin:kotlin-reflect")
35      implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
36      implementation("org.liquibase:liquibase-core")
37      implementation("org.springdoc:springdoc-openapi-ui:1.6.13")
38      implementation("org.springdoc:springdoc-openapi-kotlin:1.6.13")
39      implementation("io.jsonwebtoken:jjwt-api:$JJWT_VERSION")
40      runtimeOnly("io.jsonwebtoken:jjwt-impl:$JJWT_VERSION")
41      runtimeOnly("io.jsonwebtoken:jjwt-jackson:$JJWT_VERSION")
42      testRuntimeOnly("com.h2database:h2")
43      runtimeOnly("org.postgresql:postgresql")
44      testImplementation("org.springframework.boot:spring-boot-starter-test")
45  }
```

Рисунок 11. Секция зависимостей

При запуске сборки проекта все зависимости будут загружены и собраны в единый .jar-файл для корректной работы веб-системы. Таким образом, нам удалось рассмотреть процесс инициализации проекта Spring Boot и управления зависимостями с помощью системы сборки Gradle. Хочется отметить, что Spring Initializr сильно упрощает процесс инициализации новых проектов, а с помощью Gradle процесс сборки становится более прозрачным и простым.

3.2 Разработка слоёв приложения

После инициализации проекта и подключения зависимостей следует перейти к дальнейшей разработке приложения. Ранее нам удалось выделить четыре слоя приложения – Entity, Repository, Service, Controller. Теперь можно рассмотреть разработку непосредственно названных слоёв.

Начнём со слоя Entity. Данный слой полностью определяет предметную область веб-системы, а также устанавливает правила хранения данных – наше приложение основано на хранении и обработке данных, поэтому этот слой является основополагающим. Ранее нам удалось выделить четыре основные сущности

нашего приложения – автор, пьеса, жанр, отзыв. Структура слоя данных показан на рисунке 12.

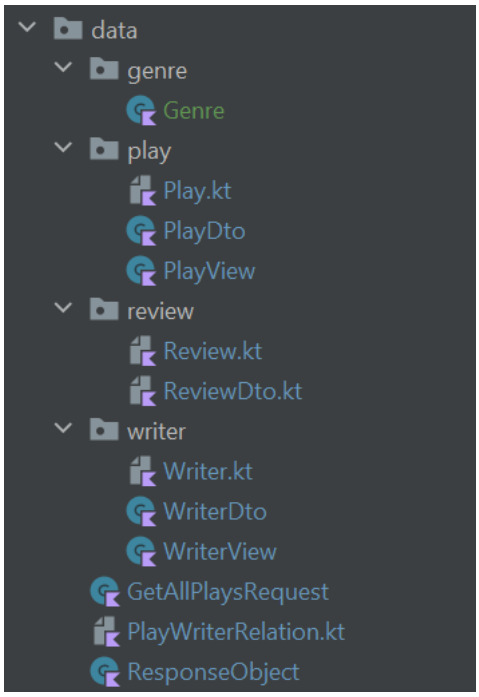


Рисунок 12. Структура слоя Entity

Так, каждый элемент предметной области имеет свой каталог. Также помимо собственно сущности имеются классы DTO и View, созданные для принятия и отправки данных. Рассмотрим разработку слоя данных на примере сущности автора. Код модели показан на рисунке 13.

```
11  @Entity
12  @Table(name = "writers")
13  @JsonInclude(JsonInclude.Include.NON_NULL)
14  @JsonIgnoreProperties(ignoreUnknown = true)
15  class Writer(
16      @Column(nullable = false)
17      var name: String,
18
19      var country: String? = null,
20      var city: String? = null,
21
22      @Id
23      @GeneratedValue(strategy = GenerationType.IDENTITY)
24      @Column(nullable = false)
25      var id: Long? = null
26  )
```

Рисунок 13. Сущность «автор»

Как и все модели, класс Writer помечен аннотацией @Entity, что позволяет использовать этот класс в качестве объекта базы данных. С помощью аннотации @Table указываем имя таблицы в базе данных – в нашем случае оно отличается от имени класса. Также каждая сущность должна иметь первичный ключ – это отображается с помощью аннотации @Id. Для первичного ключа указан способ генерации в аннотации @GeneratedValue. Стратегия Identity означает, что используется встроенный в БД тип данных столбца для генерации значения первичного ключа. Также над столбцами, которые не могут быть null, указано это правило с использованием аннотации @Column и аргументом nullable = false. Также по правилам языка Kotlin значения, которые могут быть null, помечаются типом с использованием знака ?. Этот знак означает, что данное свойство объекта может быть null. Также для них указывается значение по умолчанию. Свойство id имеет nullable-тип из-за особенностей реализации JPA в языке Kotlin. JPA генерирует id не на момент создания объекта, а на момент его сохранения в базе данных, из-за чего нам приходится помечать его как nullable для корректной работы с классом.

Рассмотрев пример написания программного кода для сущности, также обсудим разработку DTO. Для этого можно использовать такую структуру данных, как data class (класс данных). Data class в Kotlin — это класс, основная цель которого - хранить данные. Он отличается от обычного класса тем, что Kotlin автоматически генерирует для него некоторые стандартные функции, такие как equals(), hashCode(), toString(), componentN() и copy(). Мы можем сделать все поля класса иммутабельными с помощью ключевого слова val, что позволит просто принимать данные и обрабатывать их, копируя поля в сущность. Полученный пример DTO показан на рисунке 14.

```
3  data class WriterDto(
4      val name: String? = null,
5      val country: String? = null,
6      val city: String? = null
7  )
```

Рисунок 14. DTO сущности «автор»

Таким образом, нам удалось рассмотреть процесс разработки слоя Entity и DTO на примере сущности автора. Далее следует рассмотреть процесс разработки слоя Repository. Этот слой является следующим по важности – мы уже разработали объекты для маппинга данных из базы данных, но мы не можем с помощью них обращаться к самой базе.

Репозитории в Spring Data предоставляют возможность создать основные методы для запросов к базе данных всего лишь создав интерфейс, который имплементирует стандартный интерфейс, входящий в состав Spring Data. Так, на рисунке 15 показан пример репозитория для сущности автора.

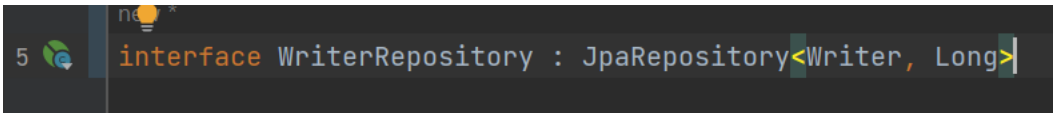


Рисунок 15. WriterRepository – репозиторий для автора

Мы получили достаточно большое количество методов для основных операций вставки, обновления, чтения и удаления просто создав пустой интерфейс. Наш интерфейс имплементирует JpaRepository, который помимо основных операций включает в себя методы для сортировки и пагинации данных. Этот интерфейс является обобщённым, в качестве первого типа он принимает класс сущности, а в качестве второго – тип данных первичного ключа. На данный момент функциональности JpaRepository полностью хватает, но также такие интерфейсы позволяют использовать SQL-запросы с помощью аннотации @Query. Существует возможность генерации нужной функциональности запроса через имя метода. На рисунке 16 показан репозиторий для отзывов.

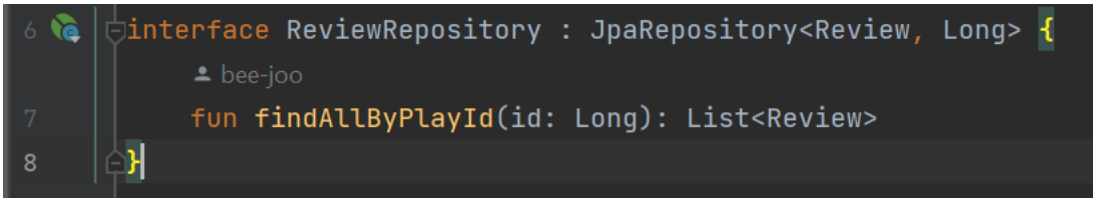


Рисунок 16. Репозиторий отзывов

Мы просто добавили пустой метод findAllByPlayId в наш интерфейс, а Spring Data автоматически сгенерирует запрос для получения этих данных. Так, нам удалось рассмотреть процесс разработки слоя репозитория. Библиотека Spring Data

JPA сильно упрощает этот процесс для типовых запросов, позволяя сконцентрироваться на бизнес-логике. Но также остаётся возможность использования собственных SQL-запросов, если этого требует задача.

Далее рассмотрим разработку слоя Service. Так как оба слоя доступа к данным уже являются разработанными, можно перейти к рассмотрению процесса разработки непосредственно бизнес-логики. Структура кодовой базы слоя изображена на рисунке 17.

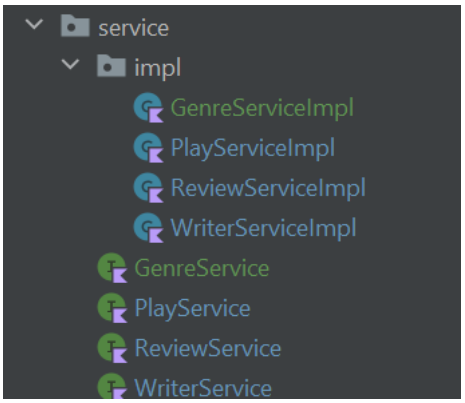


Рисунок 17. Слой Service

Соблюдая принципы SOLID, мы реализуем контракты для сервисов в виде интерфейсов. В каталоге impl содержится имплементация этих контрактов. Рассмотрим пример интерфейса сервиса на примере сервиса автора. Код показан на рисунке 18.

```
10  interface WriterService {
        bee-joo
11      fun addWriter(writerDto: WriterDto): Writer
        bee-joo
12      fun getAllWriters(orderBy: String, page: String, count: String): Page<WriterView>
        bee-joo
13      fun getPlaysByWriterId(id: Long): List<PlayView>
        bee-joo
14      fun getWriterById(id: Long): WriterView
        bee-joo
15      fun editWriter(id: Long, writerDto: WriterDto): Writer
        bee-joo
16      fun deleteWriter(id: Long): ResponseObject<String>
17  }
```

Рисунок 18. Контракт сервиса автора

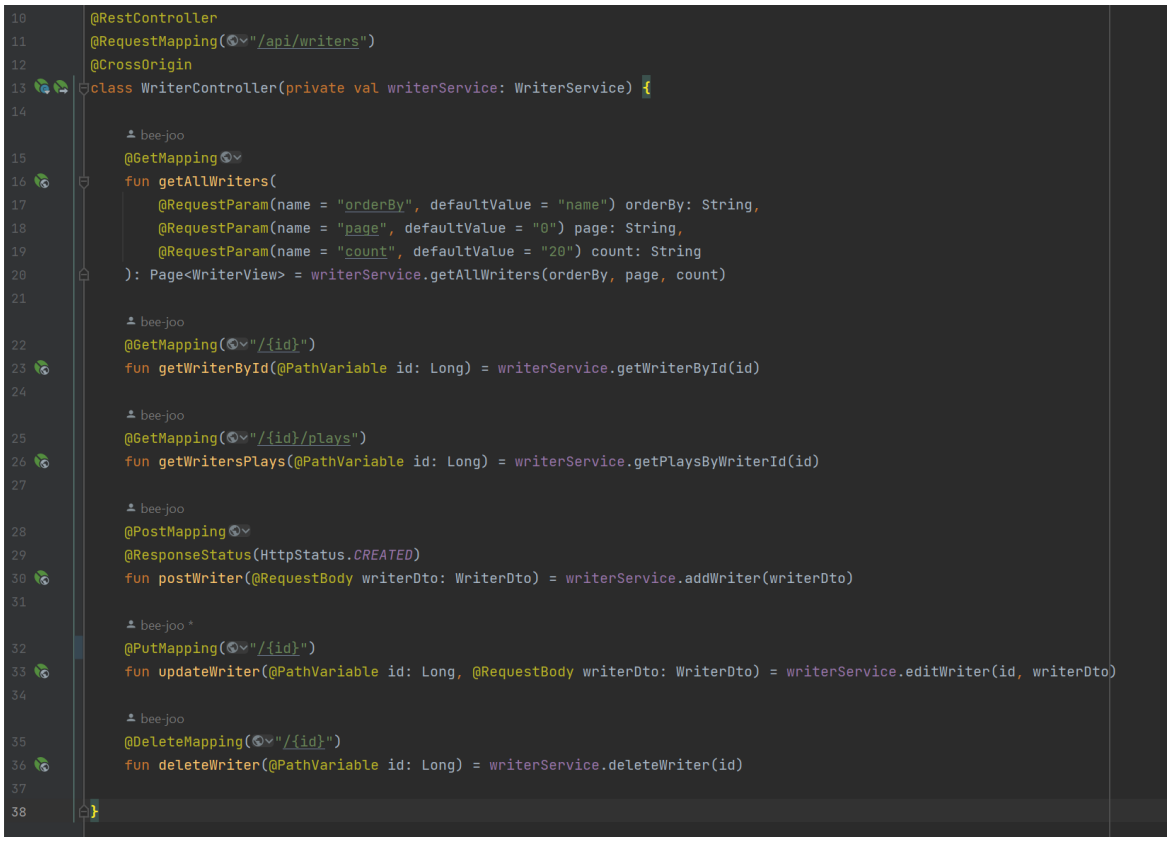
С помощью такого контракта мы определяем имена методов и их сигнатуру. Мы определяем, какая логика должна быть, какие данные принимают методы

сервисов и какие типы данных они возвращают. Пример имплементации интерфейса и одного метода показан на рисунке 19.

Рисунок 19. Имплементация сервиса

Мы создаём помеченный аннотацией @Service класс WriterServiceImpl, который имплементирует WriterService. Это означает, что наш класс с бизнес-логикой обязан полностью соблюдать описанный в интерфейсе контракт. Мы должны реализовать все методы, используя такие же сигнатуры. Так, на рисунке выше показан пример реализации метода addWriter. Реализации методов интерфейса помечаются ключевым словом override. В тело метода мы помещаем нужную для реализации задачи бизнес-логику. Здесь, например, мы конвертируем DTO в сущность и сохраняем её в базу данных. Также сигнализируем, если возникла ошибка валидации. И по такому алгоритму реализуются все методы контракта бизнес-логики – имплементируются методы, в которые помещается нужная логика.

Последним слоем после сервисного является слой Controller. Этот слой также является очень важным, так как именно он определяет фасад приложения. С помощью этого слоя веб-система открывает доступ к запросам от клиентов, и именно этот слой обрабатывает запросы и вызывает нужные методы сервисов. Также слой контроллеров возвращает ответы на запросы. Реализация слоя Controller на примере маршрутов для автора показана на рисунке 20.



```
@RestController
@RequestMapping("/api/writers")
@CrossOrigin
class WriterController(private val writerService: WriterService) {

    bee-joo
    @GetMapping
    fun getAllWriters(
        @RequestParam(name = "orderBy", defaultValue = "name") orderBy: String,
        @RequestParam(name = "page", defaultValue = "0") page: String,
        @RequestParam(name = "count", defaultValue = "20") count: String
    ): Page<WriterView> = writerService.getAllWriters(orderBy, page, count)

    bee-joo
    @GetMapping("/{id}")
    fun getWriterById(@PathVariable id: Long) = writerService.getWriterById(id)

    bee-joo
    @GetMapping("/{id}/plays")
    fun getWritersPlays(@PathVariable id: Long) = writerService.getPlaysByWriterId(id)

    bee-joo
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    fun postWriter(@RequestBody writerDto: WriterDto) = writerService.addWriter(writerDto)

    bee-joo *
    @PutMapping("/{id}")
    fun updateWriter(@PathVariable id: Long, @RequestBody writerDto: WriterDto) = writerService.editWriter(id, writerDto)

    bee-joo
    @DeleteMapping("/{id}")
    fun deleteWriter(@PathVariable id: Long) = writerService.deleteWriter(id)

}
```

Рисунок 20. WriterController

В контроллер средствами Dependency Injection внедряются сервисы, а затем при обращении к определённым маршрутам с определённым HTTP-глаголом вызываются методы этих сервисов. Аннотация @RestController говорит о том, что класс перед нами является контроллером и возвращает ответы в виде JSON. Аннотация @RequestMapping определяет базовый маршрут всех методов контроллера. HTTP-методы, которые будут использовать маршруты, определяются с помощью аннотаций @GetMapping, @PostMapping, @PutMapping, @DeleteMapping.

Также подробнее рассмотрим маршрут получения автора по его id. Если нужно получить данные, используется метод GET. В URL данного маршрута записывается строка “/{id}”, которая складывается с базовым URL – на выходе автора по id можно получить по маршруту */api/writers/{id}*. Выделить id из маршрута и использовать его как переменную в приложении позволяет аннотация @PathVariable. Затем эта переменная передаётся в соответствующий метод сервиса, а результат обработки возвращается клиенту в виде объекта JSON.

В качестве вспомогательного элемента для контроллеров можно использовать класс с аннотацией @RestControllerAdvice. Этот класс позволяет обрабатывать каждую ошибку, которая может возникнуть в приложении, и вернуть клиенту соответствующий ошибке ответ. Пример реализации показан на рисунке 21.

Рисунок 21. Обработчик ошибок

Таким образом, нам удалось рассмотреть процесс разработки всех слоёв кодовой базы веб-системы для театральной индустрии. Стоит отметить, что разделение приложения на разные уровни в зависимости от их задачи действительно упрощает поддержку приложения на следующих этапах разработки, хоть и может несколько замедлить процесс непосредственно написания кода. Готовый код рабочего прототипа приложения можно найти в репозитории. Все вышеназванные классы расположены в подкаталогах каталога *src/main/kotlin/ru/theatrebel*.

3.3 Тестирование и проверка работоспособности приложения

После разработки программных модулей следует уделить внимание их тестированию. Обсудим несколько методов тестирования приложений.

Unit-тестирование – это процесс проверки корректности работы отдельных модулей или функций программного кода. Это позволяет обнаруживать и исправлять ошибки на ранних стадиях разработки, а также упрощать интеграцию, рефакторинг и документирование кода. Для модульного тестирования используются специальные инструменты, которые автоматизируют создание и запуск тестов.

Mock-тестирование – это вид модульного тестирования, при котором используются специальные объекты-заглушки (mock-объекты), которые имитируют поведение реальных зависимостей тестируемого модуля. Это позволяет изолировать тестируемый модуль от внешних факторов и проверить его взаимодействие с другими компонентами.

Интеграционное тестирование – это вид тестирования, при котором проверяется взаимодействие и совместимость программных модулей или подсистем, объединенных в группы по логическому признаку. Обычно интеграционное тестирование проводится после модульного тестирования и до системного тестирования. Целью интеграционного тестирования является обнаружение ошибок в интерфейсах и потоках данных между компонентами программного продукта. Для автоматизации интеграционного тестирования могут использоваться системы непрерывной интеграции.

Тестировать наше приложение мы будем посредством mock-тестирования. При инициализации проекта была автоматически добавлена зависимость *spring-boot-starter-test*. Она включает в себя различные библиотеки и фреймворки для написания тестов. Нас интересуют библиотеки JUnit и Mockito. JUnit – это библиотека для автоматизированного модульного тестирования, а Mockito сочетается с этой библиотекой благодаря возможности создания mock-объектов. Пример mock-теста метода сохранения жанра из сервиса показан на рисунке 22.

```
@ExtendWith(MockitoExtension::class)
class GenreServiceTests {
    @Mock
    lateinit var genreRepository: GenreRepository

    @InjectMocks
    lateinit var genreService: GenreServiceImpl

    @Test
    fun saveGenre() {
        val name = getRandomGenreName()
        val id = Random.nextInt()
        val genre = Genre(name)

        `when`(genreRepository.save(any()))
            .thenReturn(setId(genre, id))

        val persistedGenre = genreService.addGenre(genre)
        assertEquals(genre.name, persistedGenre.name)
        assertEquals(id, persistedGenre.id)
    }
```

Рисунок 22. Mock-тест метода addGenre

Мы подключаем Mockito к классу теста с помощью аннотации @ExtendWith. Также мы создаём mock-объект репозитория жанров, используя аннотацию @Mock – репозиторий является зависимостью для сервиса. А саму заглушку сервиса мы создаём с помощью аннотации @InjectMocks, которая создаёт mock-объект и внедряет в него нужные зависимости-заглушки. Сам метод теста помечается аннотацией @Test, что позволяет фреймворку просканировать его и запустить. Результат работы метода мы определяем с помощью конструкции when…thenReturn. Работоспособные и грамотно разработанные тесты позволяют проверить работоспособность приложение до его запуска. В случае ошибки мы можем исправить её, так как нам всегда ясно, какой из тестов провалился.

Рассмотрев процесс модульного и mock-тестирования, можно проверить работоспособность приложения с помощью ручного теста. Для этого мы используем утилиту Postman. Postman – это платформа для тестирования и ручного использования API. Postman позволяет отправлять HTTP-запросы и проверять, правильно ли работает конечная точка. Проверим с помощью данного программного обеспечения маршрут получения всех пьес. Конечная точка

расположена по адресу https://theatrebel.ru/api/plays. Результат показан на рисунке 23.

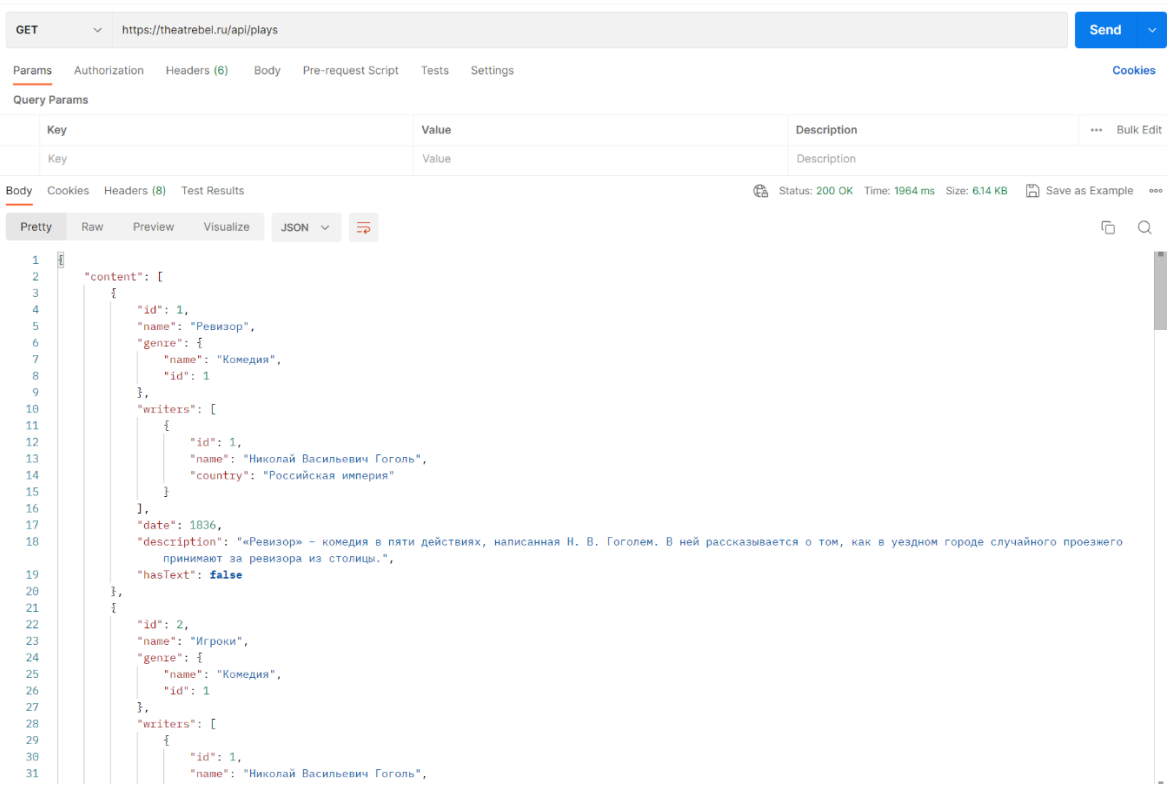


Рисунок 23. Окно программы Postman с ответом от приложения

На рисунке можно заметить, что работоспособное приложение в ответ на запрос отправило JSON с нужной информацией. Postman позволяет нам ввести маршрут, выбрать HTTP-метод и отправить соответствующий запрос, а затем отобразить ответ. Также в зависимости от задачи существует возможность добавить query-параметры, заголовки, тело запроса и многое другое.

Таким образом, нам удалось рассмотреть способы тестирования и проверки работоспособности приложения. Фреймворки для тестирования ускоряют процесс написания автоматизированных тестов, а такие HTTP-клиенты, как Postman, облегчают процесс ручной проверки приложения и отображают ответы в удобном формате.

3.4 Документация API

В процессе разработки всегда возникает необходимость документировать продукт. Это очень важный процесс, так как пользователи системы должны понимать аспекты продукта и успешно пользоваться его функционалом. Для бэкенд-систем актуальным стоит вопрос документации API. Этот вопрос решается

с использованием стандарта OpenAPI — это стандартный, независимый от языка интерфейс для HTTP API. OpenAPI использует специальный язык описания API, который может быть использован для генерации документации. Для отображения документации можно использовать Swagger – это набор инструментов для работы с OpenAPI. С помощью Swagger можно создавать, документировать и тестировать API на разных языках программирования. Swagger также предоставляет интерактивный интерфейс для просмотра и исполнения запросов к API. Интерфейс может отображать маршруты приложения вместе со структурой принимаемых и отправляемых данных.

Ранее был рассмотрен процесс добавления зависимости SpringDoc. Эта библиотека позволяет автоматически сканировать маршруты приложения и генерировать документацию по спецификации OpenAPI, а дополнение SpringDoc UI задействует Swagger для её отображения. В файле конфигурации application.yml, который расположен в каталоге */src/main/resources*, можно уточнить URL, по которому должна отображаться страница с документацией. Пример настройки показан на рисунке 24.

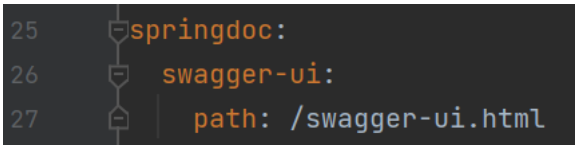


Рисунок 24. Конфигурация SpringDoc

Несмотря на то, что SpringDoc автоматически сканирует маршруты приложения, некоторые аспекты проанализировать он не может. Например, если в качестве параметра маршрута является объект, или если у полей такого объекта есть значения по умолчанию. На рисунке 25 показан пример объекта с ручной настройкой отображения в документации.

```
@ParameterObject
class GetAllPlaysRequest {
    @Schema(defaultValue = "0")
    val page: String = "0"
    @Schema(defaultValue = "20")
    val count: String = "20"
    val hasText: Boolean? = null
    val genreId: String? = null
    val year: String? = null
}
```

Рисунок 25. Объект с ручной настройкой SpringDoc

Так, аннотация @ParameterObject служит для отображения полей объекта как возможных query-параметров. А аннотация @Schema позволяет редактировать метаданные поля. В данном случае с помощью данной аннотации удалось предоставить в документации информацию о значениях полей по умолчанию.

Пример рабочей документации показан на рисунке 26.

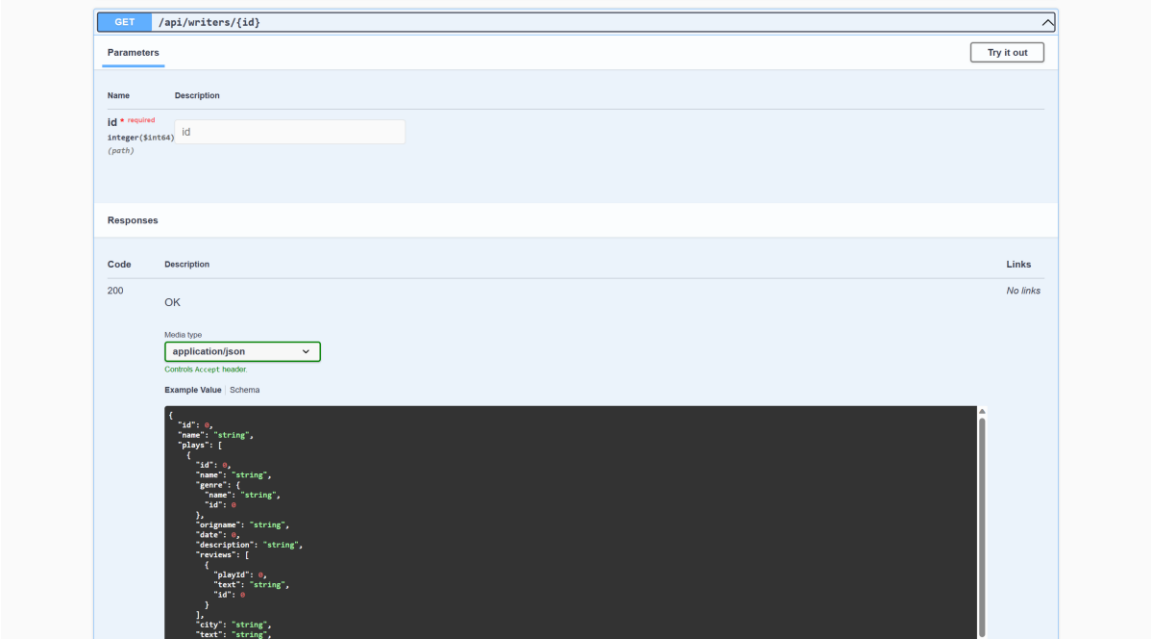


Рисунок 26. Документация маршрута получения автора

На рисунке видно, что при нажатии на маршрут раскрывается подробная документация. Помимо URL и HTTP-метода можно увидеть параметры, которые принимает маршрут, а также ту структуру данных, которую возвращает маршрут при запросе.

Таким образом, рассмотрев процесс разработки документации, можно сделать вывод о том, что спецификации OpenAPI и Swagger являются очень удобными для описания документации. Тот вид, которые они предоставляют, являются понятными для разработчиков внешних систем, таких как фронтенд-клиенты и мобильные приложения. Из-за унификации внешнего вида и условных обозначений OpenAPI де-факто является стандартом, и с его помощью разработка понятной документации упрощается.

3.5 Размещение приложения с помощью Docker Compose

Размещение приложения на сервере – весьма важный процесс, так как именно благодаря этим действиям к веб-системе смогут подключаться клиенты. Есть разные способы размещения веб-приложений – это может быть простая установка сервера и запуск приложения, также существует демонизация. В данном разделе будет рассмотрен процесс контейнеризации приложения и оркестровки с помощью Docker Compose.

Контейнеризация приложений — это процесс упаковки приложений и их зависимостей в единый компонент, который будет работать в различных средах. Контейнеры — это экземпляры исполняемого программного обеспечения, которые объединяют код приложения вместе со всеми связанными файлами конфигурации, библиотеками, зависимостями и средой выполнения и делают приложения переносимыми.

Docker — это платформа для разработки, сборки, развертывания и запуска приложений в контейнерах. Docker позволяет использовать готовые образы контейнеров с приложениями и их зависимостями, а также создавать свои собственные образы с помощью специальных файлов — Dockerfile. Docker предоставляет сервис Docker Hub для хранения и распространения образов контейнеров, который мы используем для дистрибуции нашего приложения. Docker Compose — это инструмент для запуска многоконтейнерных приложений в Docker, определенных с помощью формата файла Compose. Файл Compose используется для конфигурирования контейнеров, которые составляют приложение. С помощью файла Compose можно создавать и запускать все сервисы

из вашей конфигурации одной командой: docker compose up. Docker Compose позволяет использовать короткий и понятный синтаксис YAML для описания параметров контейнеров, таких как порты, тома, переменные окружения, сети и т.д.

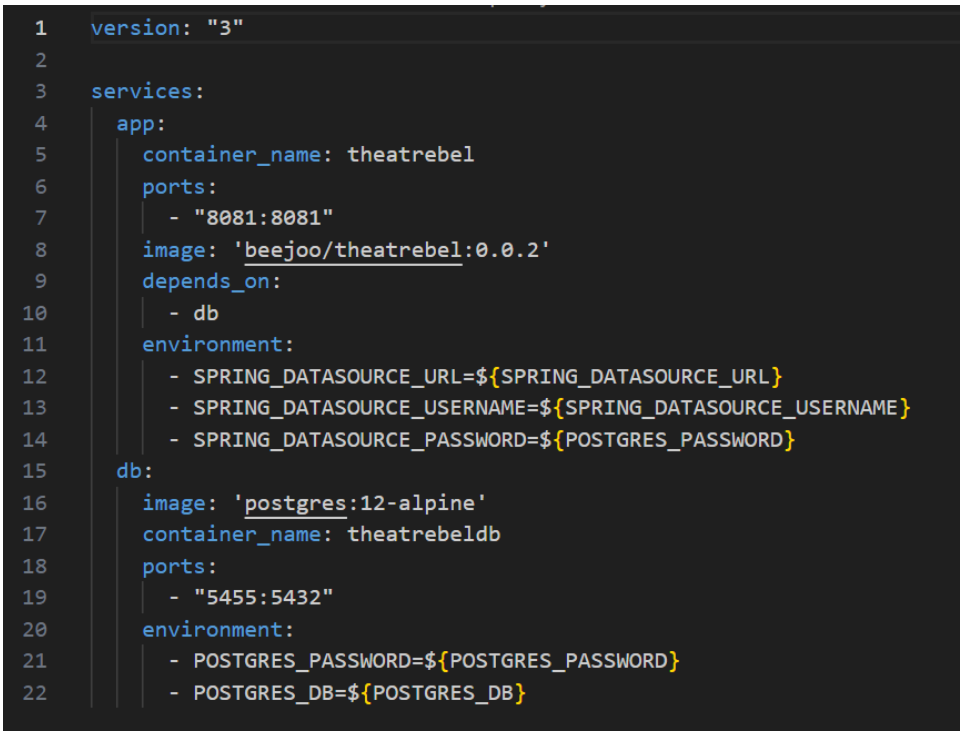
Для начала рассмотрим процесс создания Dockerfile и размещения полученного образа в Docker Hub. При создании контейнера следует выбрать образ, который будет определять среду приложения. Существует вариант использования образа Linux и самостоятельной установки в нём нужных зависимостей, но также можно выбрать готовый образ со всем нужным ПО. Есть также некоторое количество сборок JDK, с помощью которых можно запускать приложения в том числе и на языке Kotlin. В качестве сборки выберем Amazon Corretto, а в качестве дистрибутива Linux – Alpine, который является минималистичным и нетребовательным к ресурсам. Для соблюдения всех этих требований существует Docker-образ amazoncorretto-alpine. Он включает себя дистрибутив Linux Alpine и установленный Amazon Corretto. Для создания образа мы можем собрать наше приложения в единый файл .jar с помощью команды gradle build и использовать его в контейнере. Таким образом, сутью контейнера является запущенное приложение с подготовленной средой. Полученный Dockerfile показан на рисунке 27.

```
1  FROM amazoncorretto:17-alpine
2  ARG JAR_FILE=*.jar
3  COPY ${JAR_FILE} application.jar
4  ENTRYPOINT ["java", "-jar", "application.jar"]
```

Рисунок 27. Dockerfile

С помощью этого файла мы определяем образ системы, копируем в контейнер файл приложения и запускаем его. А для публикации контейнера в Docker Hub используется команда docker push. После завершения процесса создания образа и его публикации следует перейти к созданию файла для Docker Compose. В нашем случае используется два сервиса – контейнер с бэкендом веб-системы для театральной индустрии и контейнер с базой данных PostgreSQL, которая была выбрана ранее. При создании сервисов указываются их имена, а затем важные метаданные: например, образ контейнера – для БД указываем postgres:12-alpine, а для приложения опубликованный нами образ, который был назван

beejoo/theatrebel. Также указываются порты, имена контейнеров и переменные среды. Секретные данные указываются в файле .env. Полученный файл docker-compose.yml показан на рисунке 28.



```yaml
version: "3"

services:
  app:
    container_name: theatrebel
    ports:
      - "8081:8081"
    image: 'beejoo/theatrebel:0.0.2'
    depends_on:
      - db
    environment:
      - SPRING_DATASOURCE_URL=${SPRING_DATASOURCE_URL}
      - SPRING_DATASOURCE_USERNAME=${SPRING_DATASOURCE_USERNAME}
      - SPRING_DATASOURCE_PASSWORD=${POSTGRES_PASSWORD}
  db:
    image: 'postgres:12-alpine'
    container_name: theatrebeldb
    ports:
      - "5455:5432"
    environment:
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
```

Рисунок 28. Файл docker-compose.yml

Для запуска приложения с базой данных потребуются лишь установленный на машине Docker и файлы docker-compose.yml с конфигурацией и файл .env с секретными переменными. Нужно всего лишь ввести команду docker compose up, находясь в директории с файлами, а всю остальную работу по запуску сред проделает движок Docker. Таким образом, нам удалось рассмотреть процесс разворачивания приложения с помощью системы контейнеризации и оркестровки Docker Compose. Следует отметить, что использовать конфигурацию этого движка действительно удобно. Docker позволяет делать приложения переносимыми – не нужно копировать все файлы на сервер. Также Docker предоставляет для приложения изолированную от основной системы среду, что ведёт к улучшению безопасности и работоспособности систем.

ЗАКЛЮЧЕНИЕ

В ходе настоящей работы был создан рабочий прототип приложения – бэкенда веб-системы для театральной индустрии, что также является темой данной работы. Такая система позволяет пользователям находить данные о нужных театральных произведениях и постановках, что может увеличить осведомлённость населения о театре и повысить уровень образованности.

В ходе работы были рассмотрены основы бэкенд разработки, в том числе информация о фреймворках, базах данных, клиент-серверном взаимодействии. Были рассмотрены такие архитектурные шаблоны и концепции, как MVC, Clean Architecture, SOLID. Был произведён выбор технологического стэка и проведён обзор этих технологий – фреймворк Spring Boot, СУБД PostgreSQL, Docker Compose.

Также в ходе разработки удалось спроектировать структуру базы данных с помощью библиотеки Liquibase, реализовать программные модули с использованием фреймворка Spring Boot, структурировать код по концепциям паттерна Clean Architecture, использовать библиотеку доступа к данным Spring Data JPA и реализовать публикацию приложения с помощью системы оркестровки контейнеров Docker Compose. В конечном итоге существует рабочий прототип приложения, контроллеры которого могут принимать и отправлять данные.

Хотелось бы отметить, что в следующих итерациях развития приложения могут появиться новые модули, которые расширят и улучшат текущий функционал. Потенциально данная система может стать лидером в области просвещения о текущем и прошлом состоянии театральной индустрии.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Введение в Spring Boot: создание простого REST API на Java / [Электронный ресурс] // habr : [сайт]. — URL: https://habr.com/ru/articles/435144/ (дата обращения: 12.04.2023).
2. Заблуждения Clean Architecture / [Электронный ресурс] // habr : [сайт]. — URL: https://habr.com/ru/companies/mobileup/articles/335382/ (дата обращения: 12.04.2023).
3. Зачем забивать гвозди микроскопом, если есть Alpine Linux? / [Электронный ресурс] // habr : [сайт]. — URL: https://habr.com/ru/companies/digdes/articles/415279/ (дата обращения: 12.04.2023).
4. Изучаем Docker, часть 3: файлы Dockerfile / [Электронный ресурс] // habr : [сайт]. — URL: https://habr.com/ru/companies/ruvds/articles/439980/ (дата обращения: 12.04.2023).
5. Принципы SOLID в картинках / [Электронный ресурс] // habr : [сайт]. — URL: https://habr.com/ru/companies/productivity_inside/articles/505430/ (дата обращения: 12.04.2023).
6. Проблема с N+1 запросами в JPA и Hibernate / [Электронный ресурс] // habr : [сайт]. — URL: https://habr.com/ru/companies/otus/articles/529692/ (дата обращения: 12.04.2023).
7. Руководство по PostgreSQL / [Электронный ресурс] // Metanit : [сайт]. — URL: https://metanit.com/sql/postgresql/ (дата обращения: 12.04.2023).
8. Руководство по языку Kotlin / [Электронный ресурс] // Kotlinlang : [сайт]. — URL: https://kotlinlang.ru/ (дата обращения: 12.04.2023).
9. Что такое Java Spring Boot? / [Электронный ресурс] // Microsoft Azure : [сайт]. — URL: https://azure.microsoft.com/ru-ru/resources/cloud-computing-dictionary/what-is-java-spring-boot/ (дата обращения: 12.04.2023).
10. Что такое Внедрение зависимостей и как это использовать в разработке? / [Электронный ресурс] // AppTractor : [сайт]. — URL:

https://apptractor.ru/info/articles/dependency-injection.html (дата обращения: 12.04.2023).

11. Docker Compose overview / [Электронный ресурс] // Docker Docs : [сайт]. — URL: https://docs.docker.com/compose/ (дата обращения: 12.04.2023).
12. Dockerfile reference / [Электронный ресурс] // Docker docs : [сайт]. — URL: https://docs.docker.com/engine/reference/builder/ (дата обращения: 12.04.2023).
13. Introduction to Spring Data JPA / [Электронный ресурс] // Baeldung : [сайт]. — URL: https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa (дата обращения: 12.04.2023).
14. IoC, DI, IoC-контейнер — Просто о простом / [Электронный ресурс] // habr : [сайт]. — URL: https://habr.com/ru/articles/131993/ (дата обращения: 12.04.2023).
15. JSR 338: JavaTM Persistence API, Version 2.2 / [Электронный ресурс] // Oracle : [сайт]. — URL: https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf (дата обращения: 12.04.2023).
16. Kotlin Docs / [Электронный ресурс] // Kotlin : [сайт]. — URL: https://kotlinlang.org/docs/home.html (дата обращения: 14.05.2023).
17. PostgreSQL: Документация / [Электронный ресурс] // PostgresPro : [сайт]. — URL: https://postgrespro.ru/docs/postgresql (дата обращения: 12.04.2023).
18. SOLID — принципы объектно-ориентированного программирования / [Электронный ресурс] // Web Creator : [сайт]. — URL: https://web-creator.ru/articles/solid (дата обращения: 12.04.2023).
19. Spring Data JPA - Reference Documentation / [Электронный ресурс] // Spring : [сайт]. — URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (дата обращения: 12.04.2023).
20. The Clean Architecture / [Электронный ресурс] // The Clean Code Blog : [сайт]. — URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (дата обращения: 12.04.2023).