АННОТАЦИЯ

Отчет 40 с., 25 рис., 16 источн., 6 табл.

JAVA, SPRING FRAMEWORK, АРХИТЕКТУРА, ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ, КЛИЕНТ-СЕРВЕНРНОЕ ПРИЛОЖЕНИЕ, МЕРОПРИЯТИЕ

Объект исследования – клиент-серверное приложение «TicketBooking».

Цель работы – разработка архитектуры приложения для бронирования билетов на мероприятия.

В ходе работы был проведен анализ предметной области и обзор веб-приложений с аналогичной тематикой, а также проанализированы архитектурные стили и современные подходы к проектированию.

Результатом работы является веб-приложение «TicketBooking», учитывая постоянную потребность в посещении разных событий и концертов, предполагается, что в перспективе число пользователей веб-приложения будет расти.

ANNOTATION

Report 40 p., 25 fig., 16 sources, 6 tabl.

JAVA, SPRING FRAMEWORK, ARCHITECTURE, DESIGN PATTERNS, CLIENT-SERVER APPLICATION, INTERNET RESOURCE, EVENT

The object of the study is the client–server application «TicketBooking».

The purpose of the work is to develop an application architecture for booking tickets for events.

In the course of the work, a domain analysis and a review of web applications with similar topics were carried out, as well as architectural styles and modern design approaches were analyzed.

The result of the work is the «TicketBooking» web application, given the constant need to attend various events and concerts, it is assumed that in the future the number of users of the web application will grow.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящем отчете применяют следующие термины с соответствующими определениями:

Клиент – программное обеспечение или устройство, запрашивающее ресурсы или услуги у сервера в сети

Микросервис – независимый модуль системы, выполняющий определённую бизнес-логику

Монолит – архитектура, где вся система построена как единое приложение с общей кодовой базой

Фреймворк – программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта

Хостинг – услуга по предоставлению ресурсов для размещения информации на сервере, постоянно имеющем доступ к сети

Dockerfile – файл для предварительной работы, набор инструкций, который нужен для записи образа

Render – бесплатный хостинг для размещения приложений

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчете применяются следующие сокращения и обозначения:

СУБД	–	система управления базами данных
HTML	–	HyperText Markup Language, язык гипертекстовой разметки
JPA	–	Java Persistence API
JS	–	JavaScript, язык программирования
SQL	–	Structured Query Language
UML	–	Unified Modeling Language

ВВЕДЕНИЕ

Целью данной курсовой работы является разработка и обоснование архитектуры клиент-серверного приложения для записи пользователей на различные мероприятия. Для достижения этой цели необходимо провести анализ существующих решений в данной области, изучить современные подходы к проектированию клиент-серверных приложений и выбрать оптимальную архитектуру для реализации проекта. Приложение разработано с использованием языка программирования Java версии 21 [1], фреймворка Spring Boot [2] и реляционной базы данных PostgreSQL [3]. Основной функционал включает возможность бронировать билеты на мероприятия, проверку номера билетов по почте, видимость количества доступных билетов для бронирования. Для тестирования работы приложения применяется инструмент Postman [4]. Для достижения поставленной цели были сформулированы следующие задачи:

– провести анализ предметной области, связанной с разрабатываемым приложением,

– выбрать инструменты для реализации серверной части,

– изучить популярные паттерны проектирования и обосновать выбор технологий для реализации выбранной архитектуры,

– разработать архитектуру и приложение с применением выбранных технологий,

– протестировать разработанное приложение.

Для выполнения этих задач применяются методы анализа и сравнительного исследования. Информационной базой для этой работы являются знания, полученные в ходе практических занятий курса «Архитектура клиент-серверных приложений», а также данные из интернет-ресурсов.

В разделе с описанием предметной области расположена основная информация о проведенном анализе предметной области, функциональные и нефункциональные требования к системе, а также ограничения системы.

В разделе обоснования выбора технологий описано используемое программное обеспечение и технологии.

В разделе разработки приложения описана структура приложения, подключение к базе данных и реализация клиентской и серверной частей, а также тестирование.

В разделе заключения подводятся итоги курсовой работы.

1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Анализ предметной области

Анализ предметной области проводился среди веб-приложений на тематику «Бронирование билетов на мероприятия». Были выбраны три веб-приложения: Яндекс афиша [5], Ticketland [6], MTC live [7] представленные на рисунках 1.1-1.3.

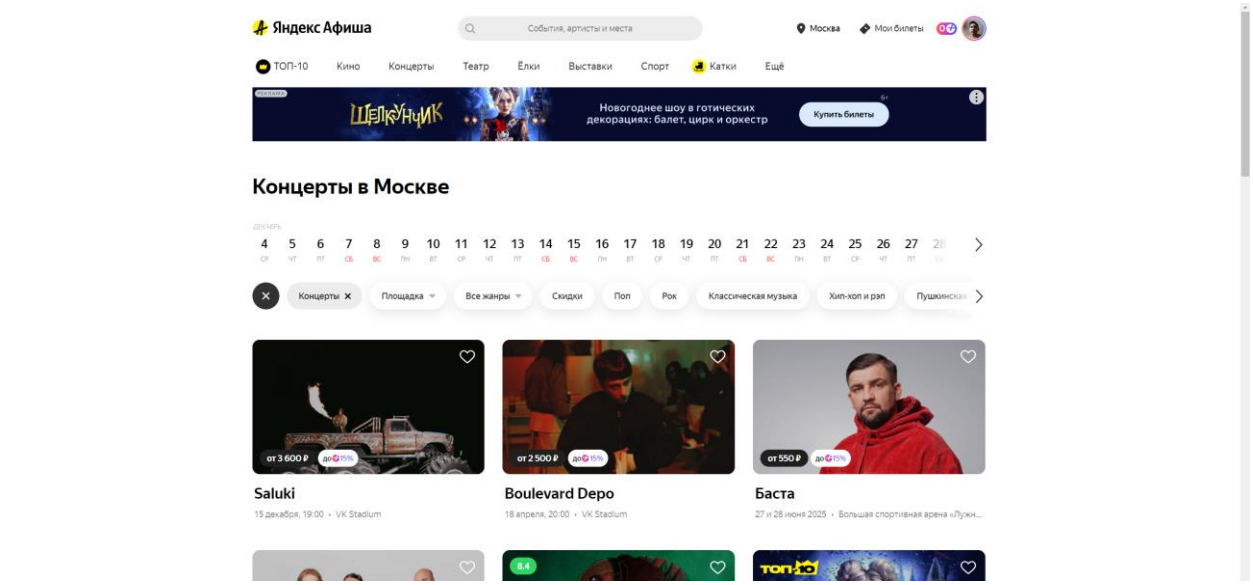


Рисунок 1.1 – Клиентская часть сайта Яндекс афиша

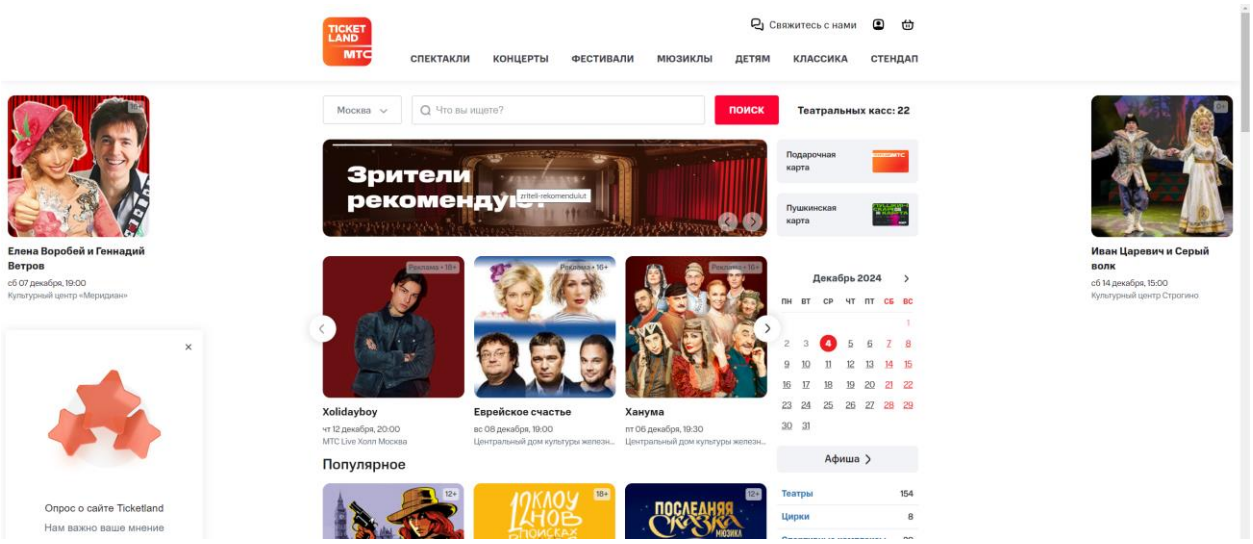


Рисунок 1.2 – Клиентская часть сайта Ticketland

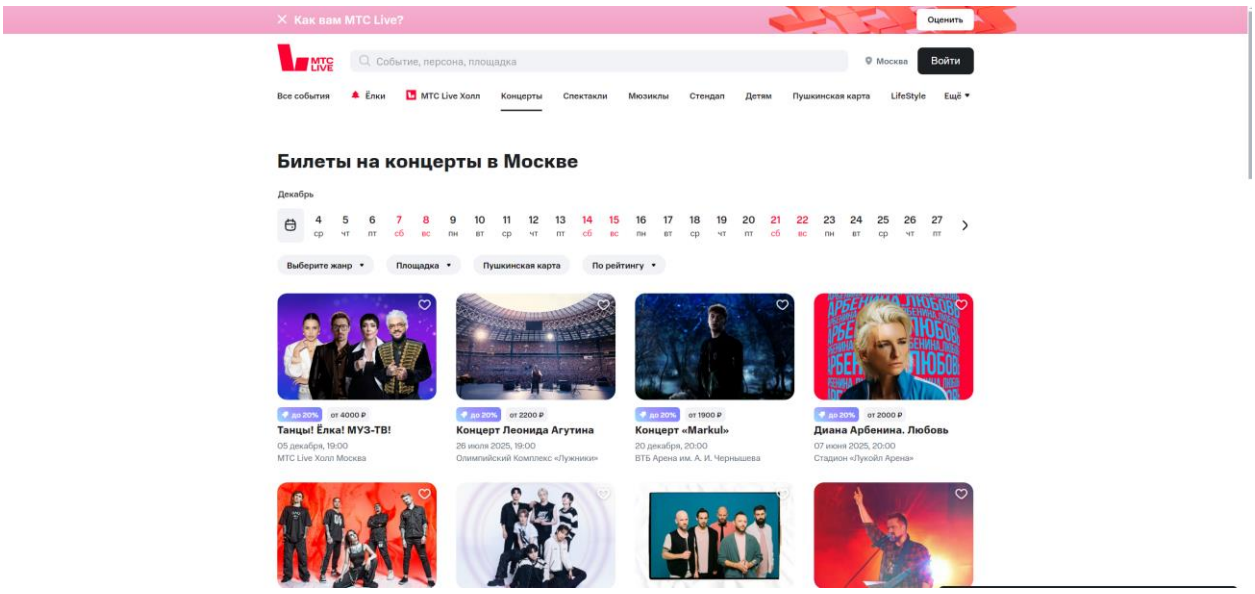


Рисунок 1.3 – Клиентская часть сайта МТС live

Все рассмотренные веб-приложения предоставляют пользователям возможность регистрации и входа в аккаунт, управления личными данными, записи на мероприятия, а также просмотра информации о доступных событиях.

Из анализа данных веб-приложений следует выделить ключевые возможности, которыми должен обладать разрабатываемый веб-ресурс данной тематики:

– регистрация на выбранное мероприятие,

– просмотр списка забронированных билетов,

– просмотр количества оставшихся билетов на мероприятия,

– просмотр списка доступных мероприятий.

Приложение должно быть разработано с учетом требований современного рынка веб-приложений в данной тематике, обеспечивая конкурентоспособность. Для этого необходимо внедрить актуальные функции, такие как предоставление информации о мероприятиях, а также возможность просмотра и бронирования доступных билетов. Кроме того, важно обеспечить удобство интерфейса, высокую производительность и адаптацию под различные устройства для привлечения широкой аудитории.

Уникальность приложения заключается в его простоте и удобстве для пользователей. Интуитивно понятный интерфейс позволяет легко разобраться в функционале, а отсутствие сложной системы авторизации делает процесс еще быстрее и проще. Обычно люди проверяют билеты уже непосредственно перед мероприятием, и возможность мгновенно увидеть свои билеты, не тратя время на авторизацию, является значительным преимуществом. Это не только экономит время, но и создает положительное впечатление от использования приложения, что способствует его привлекательности для пользователей.

1.2 Описание функционала приложения

В разрабатываемом приложении для регистрации на мероприятия необходимо реализовать следующие функции: пользователь должен иметь возможность зарегистрироваться на выбранное мероприятие, заполнив необходимые данные, такие как имя, фамилию и email; перед регистрацией система должна проверять доступность мест на мероприятие; пользователь должен видеть список всех доступных мероприятий, включая их названия, описание и количество оставшихся мест. Для удобства пользователь должен иметь возможность просматривать список мероприятий, на которые он уже зарегистрировался, с указанием их названий и уникальных номеров билетов. Система должна обрабатывать ошибки, возникающие в процессе регистрации или при работе с данными, и информировать пользователя о таких ситуациях.

Среди нефункциональных требований ключевое внимание уделяется удобству и производительности интерфейса, который должен быть интуитивно понятным и обеспечивать быстрый доступ к основным функциям. Создание и просмотр записей должны происходить без задержек. Взаимодействие клиента и сервера должно быть реализовано оптимальным образом для обеспечения стабильной работы приложения даже при большом количестве одновременных пользователей. Система должна быть легко масштабируемой, чтобы поддерживать рост нагрузки и объема данных. Это, например, включает в себя увеличение количества одновременных активных пользователей с 100 до 1000, а также расширение общей пользовательской

базы с 500 до 10 000 зарегистрированных пользователей, которые когда-либо взаимодействовали с сервисом. Кроме того, архитектура должна эффективно обрабатывать рост объема данных в базе данных от 100 мегабайт до 10 гигабайт, обеспечивая стабильность, производительность и отказоустойчивость на всех этапах масштабирования. Особое внимание необходимо уделить безопасности данных пользователей, включая шифрование конфиденциальной информации, такой как email.

Проект имеет ряд ограничений, которые могут повлиять на его работу. Система будет функционировать в условиях ограниченных серверных ресурсов, а именно – оперативная память от 2 ГБ, а процессорное время от 300 секунд CPU в сутки. Это потребует эффективной оптимизации алгоритмов и процессов для снижения нагрузки на серверы, поддержания стабильности работы и обеспечения высокой производительности при увеличении нагрузки. Возможны задержки при обработке запросов, если в базе данных хранится большое количество данных, например, при значительном числе зарегистрированных мероприятий или пользователей. Увеличение нагрузки на сервер может привести к замедлению работы приложения, особенно при большом количестве пользователей, одновременно выполняющих действия, такие как регистрация или просмотр мероприятий.

Разработка и тестирование ключевых компонентов системы должны быть завершены в срок не более 1-2 месяцев, что обеспечит своевременный запуск проекта и готовность к увеличению нагрузки.

1.3 Анализ архитектурных стилей и паттернов проектирования

Для выбора архитектурного стиля разрабатываемого приложения необходимо провести анализ наиболее популярных архитектур.

Одним из наиболее распространенных является монолитная архитектура. Монолитная архитектура – это подход, при котором все компоненты и функции приложения объединены в единую систему. Она выделяется своей простотой в разработке и развертывании: все элементы находятся в одном месте, что уменьшает затраты времени и ресурсов. Кроме

того, тестировать монолитное приложение проще, так как вся система проверяется как целое. Однако масштабируемость в монолите представляет собой вызов: для увеличения производительности необходимо масштабировать всю систему, а не отдельные её части. Это усложняет обновления, поскольку изменения одного компонента могут повлиять на другие части, увеличивая вероятность ошибок и сбоев. Кроме того, использование разных технологий внутри одной системы ограничено.

Клиент-серверная архитектура, другой популярный стиль, основывается на разделении приложения на две части: клиент, запрашивающий услуги, и сервер, предоставляющий их. Основное преимущество такого подхода – чёткое разделение ролей, что упрощает поддержку и развитие системы. Однако этот стиль имеет и слабые стороны, например, зависимость от сети. Если клиент генерирует слишком много запросов, сервер может испытывать перегрузку, что негативно влияет на производительность системы.

Сервисно-ориентированная архитектура (SOA) строится на идее создания приложения в виде набора взаимосвязанных сервисов, каждый из которых выполняет конкретную задачу. Такой подход позволяет повторно использовать компоненты, снижая затраты на разработку, и упрощает масштабирование отдельных частей системы. Однако взаимодействие между сервисами часто требует сложного управления. Для обеспечения совместимости и стабильной работы приложения требуется значительное внимание и усилия.

Микросервисная архитектура – это современный подход, развивающий идеи сервисно-ориентированной архитектуры (SOA). Она предполагает деление приложения на множество небольших и автономных сервисов, каждый из которых отвечает за выполнение своей задачи. Такой стиль дает высокую гибкость: для каждого микросервиса можно использовать свои технологии и языки программирования, что облегчает адаптацию системы к разным требованиям. Кроме того, микросервисы упрощают процесс

развертывания и обновления: отдельные сервисы можно обновлять независимо, без необходимости вмешательства в работу всего приложения.

Для разрабатываемого приложения по регистрации на мероприятия был выбран монолитный архитектурный стиль. Этот выбор обусловлен несколькими факторами, которые делают монолит оптимальным решением в данном контексте.

Во-первых, разрабатываемое приложение имеет относительно небольшую сложность. Его ключевые функции – регистрация на мероприятие, просмотр списка доступных мероприятий и просмотр списка зарегистрированных событий у пользователя, данные функциональные требования не нуждаются в масштабной декомпозиции на множество отдельных сервисов. Все эти функции логически связаны и взаимодействуют друг с другом, что позволяет эффективно реализовать их в рамках единой системы.

Во-вторых, монолитная архитектура обеспечивает более быстрое достижение целей разработки. Поскольку все модули приложения объединены в одном проекте, не требуется тратить дополнительные ресурсы на настройку взаимодействия между компонентами. Это особенно важно при ограниченных сроках или ресурсах.

Третьим фактором является упрощённое тестирование и развертывание приложения. Монолит позволяет тестировать приложение как единое целое, без необходимости учитывать сложные взаимодействия между независимыми модулями или сервисами. Развертывание также упрощается: нет необходимости настраивать несколько независимых сред, что уменьшает вероятность ошибок при внедрении изменений.

Кроме того, монолитная архитектура лучше подходит для небольших приложений с предсказуемой нагрузкой. В разрабатываемом приложении нагрузка будет зависеть от числа пользователей, но даже в случае увеличения числа регистраций на мероприятия она остаётся управляемой.

Масштабирование приложения при необходимости может быть выполнено путём увеличения вычислительных ресурсов сервера.

Наконец, монолит позволяет быстрее вносить изменения. В условиях, когда функциональные требования могут меняться в процессе разработки, важно, чтобы разработчики могли оперативно дорабатывать или исправлять функционал. В монолите изменения можно внедрять без необходимости координации между командами, отвечающими за разные микросервисы, что ускоряет цикл разработки. Это больше подходит к данному приложению, так как его сложность не должна вызывать трудностей в проектировке, что делает выбор монолита более логичным.

Таким образом, выбор монолитной архитектуры для приложения по регистрации на мероприятия полностью оправдан. Этот архитектурный стиль позволяет эффективно достигать целей разработки, сокращает временные и финансовые затраты, а также обеспечивает надежность и простоту дальнейшего обслуживания, основываясь на требованиях к приложению.

В разработке программных приложений широко используются различные паттерны проектирования, каждый из которых решает конкретные задачи и повышает гибкость, читаемость и тестируемость кода. В нашем проекте по регистрации на мероприятия используется несколько ключевых паттернов проектирования, которые помогают организовать структуру приложения и упростить решение задач. Одним из таких паттернов является паттерн слой (Layered Pattern), который применяется для разделения приложения на независимые слои с чётко определёнными обязанностями. Паттерн «Слой» является одним из наиболее часто используемых в разработке корпоративных приложений. Его основная цель – разделить приложение на несколько уровней, каждый из которых решает определённую задачу и взаимодействует с другими уровнями через чётко определённые интерфейсы. Это позволяет изолировать изменения, улучшить тестируемость, а также упростить поддержку и развитие приложения в будущем.

Помимо паттерна «Слой» используются и другие паттерны проектирования, каждый из которых решает конкретные задачи в процессе разработки. В этом проекте применяются такие паттерны как Репозиторий (Repository), Фабрика (Factory), Одиночка (Singleton) и Стратегия (Strategy).

Паттерн «Репозиторий» является одним из основополагающих паттернов для работы с данными в приложении. Он позволяет инкапсулировать логику взаимодействия с базой данных и обеспечивать простоту доступа к данным для остальных слоёв приложения, таких как слой бизнес-логики.

Паттерн «Фабрика» используется для создания объектов, особенно когда создание объекта требует сложной логики или когда тип создаваемого объекта может изменяться в зависимости от условий. В нашем проекте паттерн «Фабрика» можно применить для создания различных типов мероприятий в зависимости от их характеристик. Например, для регистрации на мероприятия могут быть различные виды мероприятий, такие как спортивные события, концерты, выставки и т.д. В таком случае, фабрика может быть использована для создания объектов разных типов мероприятий с различными параметрами.

Паттерн «Одиночка» используется для того, чтобы гарантировать существование единственного экземпляра объекта в приложении. Этот паттерн часто используется для объектов, которые должны быть созданы один раз и использоваться в течение всего жизненного цикла приложения. В данном проекте этот паттерн может быть применён для сервисов, которые должны быть единственными в рамках приложения. Используя данный паттерн, мы можем гарантировать, что каждый сервис или компонент будет иметь только один экземпляр, что позволит эффективно управлять его состоянием и обеспечит централизованное управление.

Паттерн «Стратегия» используется для изменения поведения объекта во время его работы, заменяя алгоритм, который он использует, без изменения самого объекта. В контексте приложения это может быть полезно при реализации различных алгоритмов обработки регистрации на мероприятия,

например, разные способы оплаты, различные критерии для фильтрации мероприятий и т.д.

Паттерн «Цепочка обязанностей» позволяет избежать жесткой связи между отправителями запросов и получателями, позволяя нескольким объектам обработать запрос. В нашем проекте этот паттерн может быть использован для обработки запросов на регистрацию на мероприятия, где каждый объект может обрабатывать запрос в зависимости от его типа или условий, прежде чем передать его дальше по цепочке. Для приложения цепочка обязанностей может быть полезна при реализации проверки данных на каждом этапе регистрации, например, проверка email, наличие мест на мероприятии и других условий.

В данном проекте паттерн «Слой» будет использоваться как ключевой паттерн для разделения логики работы с базой данных, бизнес-логики и представления пользователя. Применение такого паттерна способствует созданию гибкой и масштабируемой архитектуры, где каждый слой выполняет свою чётко определённую роль и может быть модифицирован без воздействия на остальные слои. Каждый слой взаимодействует с соседними слоями через чётко определённые интерфейсы, что позволяет избежать чрезмерной зависимости между компонентами и способствует изоляции изменений. Например, слой представления может обращаться к слою бизнес-логики через сервисы, но не имеет прямого доступа к базе данных. Это снижает связанность компонентов и упрощает поддержку приложения. Для разрабатываемого приложения, которое включает базовый функционал регистрации на мероприятия, просмотра доступных событий и регистрации на них, паттерн «Слой» является оптимальным решением. Этот проект не является сложным, и его задачи легко разделить на чёткие категории, такие как представление данных, обработка бизнес-логики и работа с данными.

Для выбора архитектурного паттерна разрабатываемого приложения необходимо разобрать наиболее популярные виды. Сравнительный анализ характеристик архитектурных паттернов приведён в таблице 1.1.

Таблица 1.1 – Характеристики архитектурных паттернов

Характеристика	MVC (Model-View-Controller)	MVP (Model-View-Presenter)	MVVM (Model-View-ViewModel)
Сложность реализации	Низкая. Простая реализация на начальных этапах.	Средняя. Требует больше усилий для реализации из-за добавления презентера.	Высокая. Реализация сложнее всего из-за необходимости создания ViewModel.
Использование слоев	Да	Да	Да
Удобство тестирования	Хорошая, но тестирование контроллера сложно.	Очень высокая, презентер легко тестировать.	Отличная, ViewModel полностью изолируется.
Гибкость	Средняя, сложно добавлять новые функции.	Высокая, благодаря слабой зависимости слоев.	Очень высокая, легко адаптируется под изменения.
Тестируемость	Высокая	Очень высокая	Высокая
Поддержка изменений	Затруднена из-за сильных зависимостей.	Удобная благодаря изоляции логики в презентере.	Лучшая, изменения легко изолируются.
Масштабируемость	Умеренная, часто требует переписывания кода.	Высокая, легко добавлять новые функции.	Отличная, минимальное вмешательство в код.
Поддержка изменений	Затруднена из-за сильных зависимостей.	Удобная благодаря изоляции логики в презентере.	Лучшая, изменения легко изолируются.

Для разработки приложения наиболее подходящим является архитектурный паттерн MVC (Model-View-Controller). Он разделяет логику на модель, представление и контроллер, что делает процесс разработки и поддержки более удобным. Данный паттерн идеально подходит для приложений с простым интерфейсом, не требующих сложных взаимодействий, характерных для MVP или MVVM. MVC обеспечивает эффективное управление данными и их отображением при сохранении низкой сложности реализации.

1.4 Описание функционала приложения в нотациях UML и DFD

Для создания UML-диаграмм [8] использован сервис draw.io [9].

Согласно функциональным требованиям создана диаграмма Use-case, показанная на рисунке 1.4.

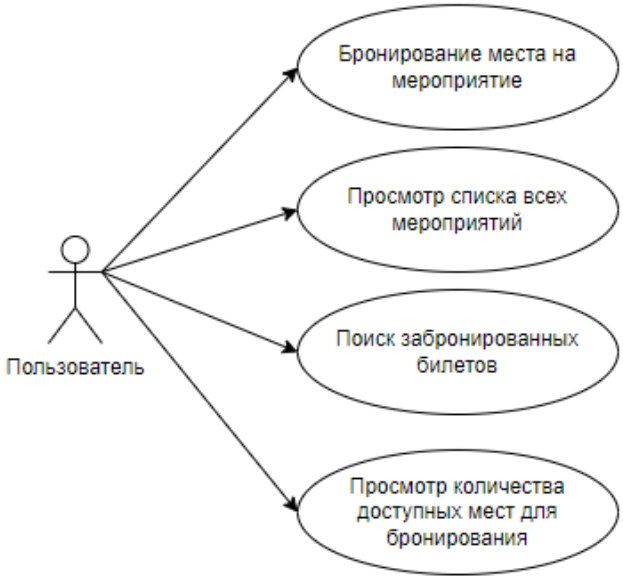


Рисунок 1.4 – Диаграмма Use-case

На рисунке 1.5 представлена диаграмма компонентов, обобщённо показывающая структуру разрабатываемого клиент-серверного приложения.

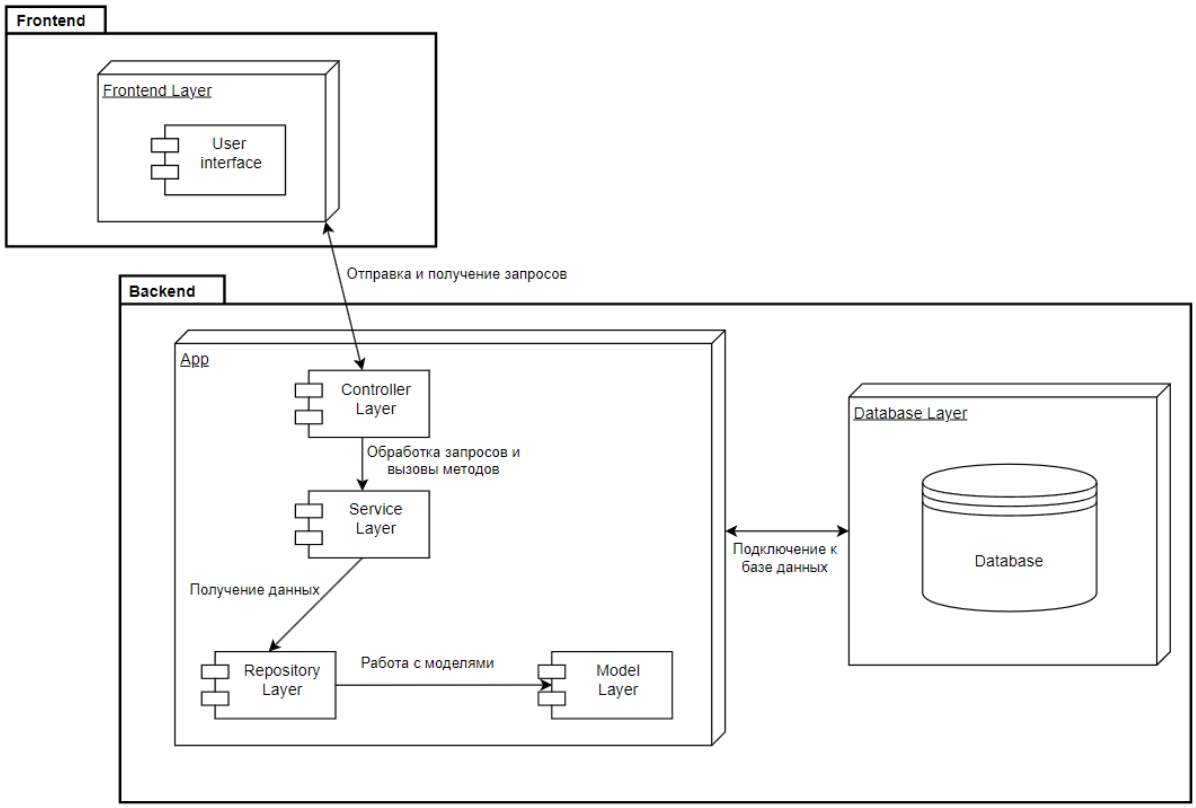


Рисунок 1.5 – Обобщённая диаграмма компонентов

Для пользования разрабатываемой системой, пользователь может различными способами взаимодействовать приложением. Создана диаграмма последовательности для API, которая отображает реализацию нескольких функций приложения, а именно вывод данных о доступных мероприятиях, бронирование билета на мероприятие (с успешной записью), поиск билета по email (в случае, когда билеты были найдены). Диаграмма последовательности представлена на рисунке 1.6.

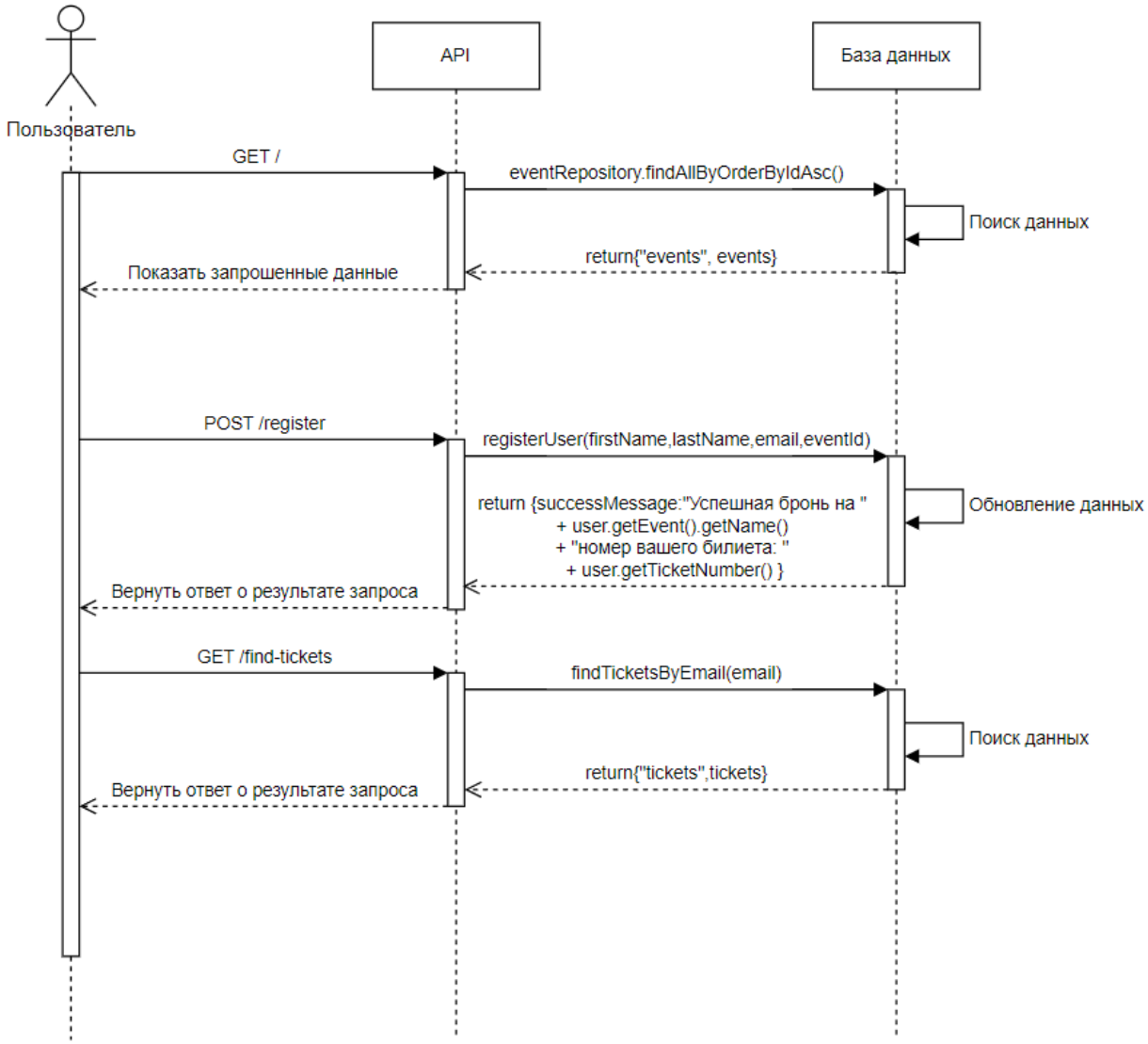


Рисунок 1.6 – Диаграмма последовательности для API

Далее представлена DFD-диаграмма, отображающая один из возможных процессов реализуемого приложения, а именно процесс поиска билетов по

email, как показано на рисунке 1.7, на рисунке 1.8 представлена декомпозиция процесса.

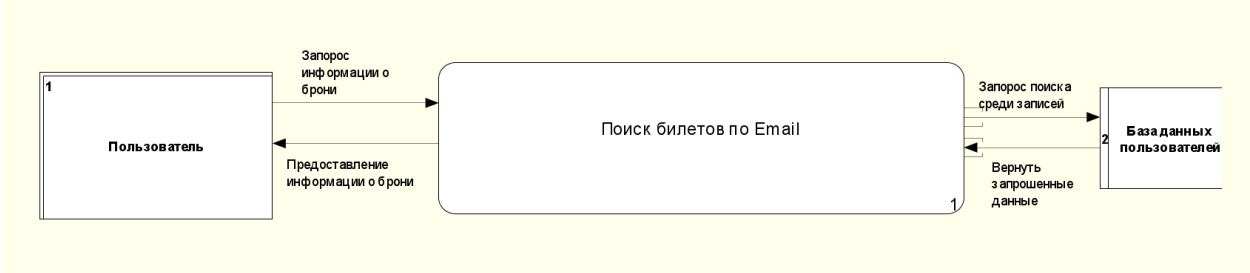


Рисунок 1.7 – DFD диаграмма

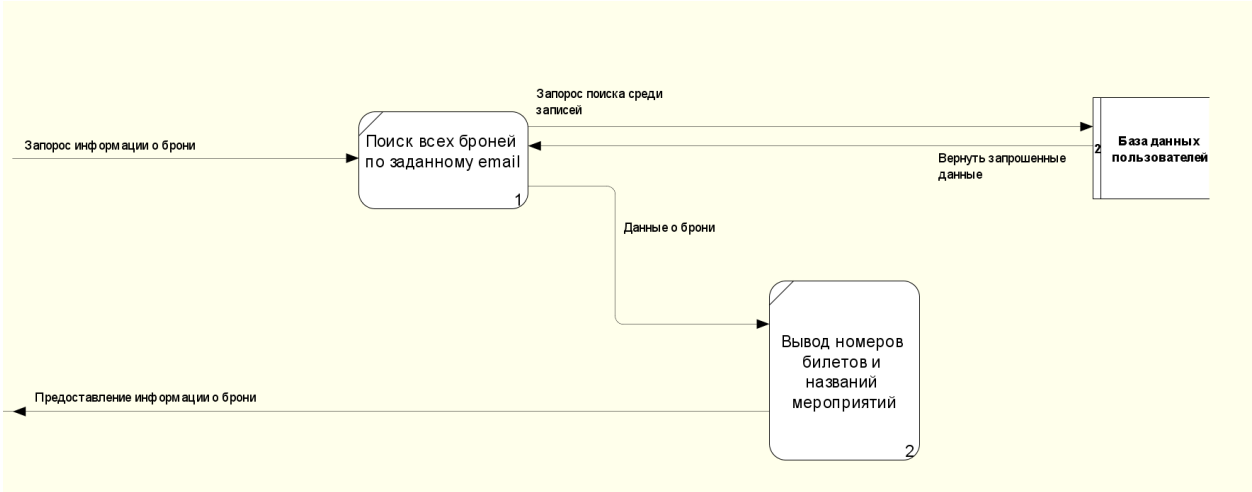
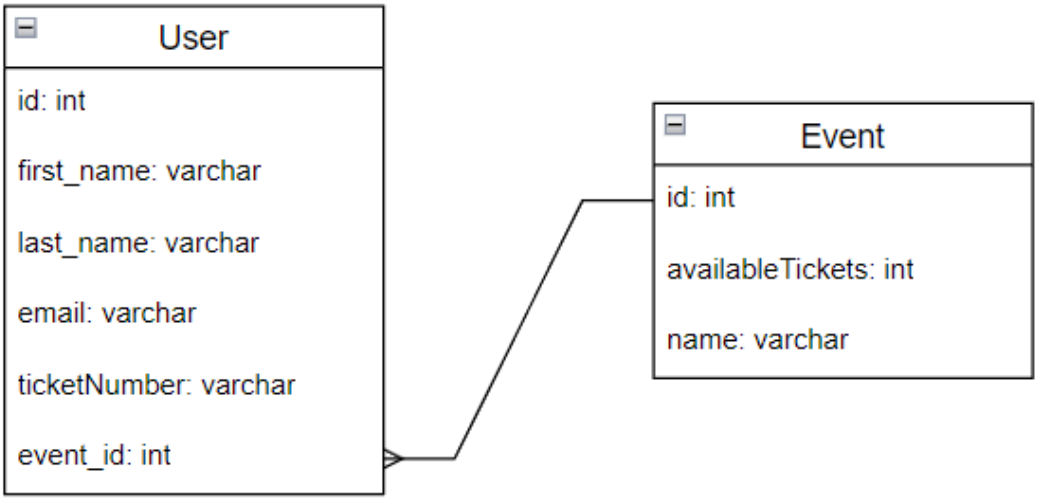


Рисунок 1.8 – Декомпозиция процесса

1.5 Структура базы данных

Для клиент-серверного приложения потребуется база данных для пользователей и событий. Схема базы данных показана на рисунке 1.9. Event (события) и Users (пользователя) связаны между собой через отношение «Многие к одному», где одно событие может иметь множество пользователей, зарегистрированных на него.



Рисуно 1.9 – Структура базы данных мероприятий и пользвателей

2 ОБОСНОВАНИЕ ВЫБОРА ТЕХНОЛОГИЙ

2.1 Используемое прикладное программное обеспечение

Для выбора среды разработки проведён сравнительный анализ IntelliJ IDEA [10] и VS Code [11], представленный в таблице 2.1.

Таблица 2.1 – Характеристики сред разработки

Характеристика	IntelliJ IDEA	VS Code
Поддержка Java	Встроенная поддержка, мощный инструмент для Java	Требуются расширения для полноценной поддержки
Интеграция с Spring Boot	Встроенная интеграция и шаблоны для Spring Boot	Настраивается через плагины (Spring Boot Tools)
Работа с проектами Maven/Gradle	Полная поддержка, встроенные инструменты	Требуются расширения (Maven, Gradle Extensions)
Производительность	Может быть медленным на больших проектах	Легковесный, работает быстрее
Отладка кода	Расширенные возможности отладки JVM	Базовая отладка через расширения
Поддержка других языков	Ограничена, но подходит для JVM-экосистемы	Многоязычная поддержка через плагины

Для разработки приложений на Spring будет использоваться IntelliJ IDEA, так как она обеспечивает встроенную поддержку фреймворка Spring, удобные инструменты для управления зависимостями, выполнения тестирования и отладки. Эти функции позволяют ускорить процесс разработки, повысить качество кода и упростить тестирование, делая работу разработчика более удобной и эффективной.

Для проверки работы клиент-серверного приложения был выбран браузер Google Chrome, поскольку он широко используется, обладает высокой производительностью и поддерживает современные веб-стандарты.

2.2 Используемые технологии

2.2.1 Языки программирования

Для выбора языка программирования для разработки клиент-серверного приложения был проведён сравнительный анализ нескольких популярных языков, учитывающий такие ключевые аспекты, как платформенная независимость, производительность, библиотеки и фреймворки, поддержка многозадачности, простота и удобство разработки, безопасность. В рамках анализа были рассмотрены Java, Python и C++, каждый из которых обладает

своими сильными сторонами и ограничениями, зависящими от требований проекта. Основные характеристики этих языков приведены в таблице 2.2.

Таблица 2.2 – Характеристики языков программирования

Характеристики	Java	Python	C++
Платформенная независимость	Высокая, благодаря JVM, приложения работают на разных ОС без изменения кода.	Высокая, через интерпретатор Python, работает на большинстве платформ.	Низкая, требует перекомпиляции для каждой целевой платформы.
Производительность	Хорошая, уступает C++, но достаточна для большинства приложений.	Умеренная, медленнее Java и C++ из-за интерпретируемого характера языка.	Высокая, особенно в задачах, требующих низкоуровневой оптимизации.
Библиотеки и фреймворки	Богатый набор, включая Spring, Hibernate и др., идеален для клиент-серверных приложений.	Широкий спектр библиотек, включая Django и Flask, удобен для быстрого прототипирования.	Ограниченный выбор по сравнению с Java и Python, хотя доступны Qt и Boost для разработки.
Поддержка многозадачности	Отличная, встроенные механизмы для потоков и параллельной обработки через библиотеки (например, Executors).	Умеренная, GIL ограничивает эффективность потоков, но asyncio хорошо подходит для I/O задач.	Высокая, предлагает низкоуровневый контроль над потоками, подходит для задач высокой нагрузки.
Простота и удобство разработки	Средняя, требуется понимание концепций ООП, но хорошее автодополнение и документация упрощают процесс.	Высокая, простой синтаксис, отлично подходит для начинающих разработчиков.	Низкая, сложный синтаксис, требующий внимательного управления памятью и компиляцией.
Безопасность	Высокая, благодаря JVM, сборке мусора и встроенным механизмам управления памятью.	Умеренная, зависит от внешних библиотек и качества написанного кода.	Низкая, высокий риск ошибок в управлении памятью, требует опыта для написания безопасного кода.

Для разработки выбран язык программирования Java и система сборки Gradle. Gradle был выбран из-за его высокой производительности, гибкости в

настройке и широкого распространения в сообществе разработчиков, что обеспечивает наличие качественной документации и активной поддержки.

2.2.2 Фреймворки и библиотеки

Для выбора фреймворка разрабатываемого клиент-серверного приложения проведён сравнительный анализ нескольких популярных фреймворков, среди которых были рассмотрены Spring, Django [12], а также Flask. Каждый из этих фреймворков имеет свои особенности, которые могут быть полезны в зависимости от требований проекта, таких как требуемая производительность и скорость разработки. Основные характеристики рассматриваемых фреймворков приведены в таблице 2.3.

Таблица 2.3 – Характеристики популярных фреймворков

Характеристика	Spring	Django	Flask
Язык программирования	Java	Python	Python
Производительность	Высокая, но зависит от конфигурации	Средняя, подходит для большинства веб-приложений	Высокая, особенно для небольших приложений
Масштабируемость	Отличная, поддержка микросервисов	Хорошая, но не идеально для очень крупных приложений	Хорошая для малых и средних приложений
Библиотеки	Большое количество встроенных решений для работы с базами данных, безопасности, кешированием и др.	Отличная интеграция с ORM, отличная поддержка админки	Минималистичный, позволяет гибко настраивать компоненты
Сообщество и поддержка	Широкое, много ресурсов, документации и примеров	Огромное сообщество, много документации	Большое сообщество, но меньше примеров по сравнению с Django
Безопасность	Встроенная поддержка защиты от CSRF, XSS, SQL-инъекций и др.	Высокая, включены защиты по умолчанию, как защита от CSRF	Требуется дополнительная настройка для обеспечения безопасности

Для разрабатываемого клиент-серверного приложения был выбран фреймворк Spring. Во-первых, Spring предоставляет высокую производительность и отличную масштабируемость, что позволяет эффективно разрабатывать приложения, которые могут легко адаптироваться

к изменяющимся требованиям. Одним из преимуществ Spring является интеграция с различными базами данных, включая реляционные и NoSQL базы. Использование Spring Data JPA позволяет разработчикам легко работать с базами данных через стандартный подход, который минимизирует количество кода для выполнения операций с данными. Это ускоряет разработку и делает код более читаемым и поддерживаемым. Кроме того, использование дополнительных инструментов, таких как Spring Boot, позволило значительно ускорить процесс разработки, предоставив возможность быстро настроить и запустить приложение с минимальными усилиями.

Также для выбора шаблонизатора, позволяющего динамически создавать элементы на HTML-страницах проведён сравнительный анализ популярных представителей, представленный в таблице 2.4.

Таблица 2.4 – Характеристики шаблонизаторов

Характеристика	Thymeleaf	Mustache
Синтаксис	XML-подобный, интуитивно понятный	Простая текстовая разметка
Интеграция с Java	Глубокая интеграция, поддержка Spring	Минимальная интеграция
Гибкость в форматировании	Высокая гибкость, поддержка сложных конструкций	Ограниченная гибкость, простые конструкции
Производительность	Высокая производительность, кэширование шаблонов	Легковесный, но без встроенного кэширования
Поддержка тестирования	Хорошая поддержка тестирования через Spring	Нет встроенной поддержки для тестирования

Применение Thymeleaf [13] для работы с HTML-страницами позволит эффективно разрабатывать динамичные и интерактивные пользовательские интерфейсы.

2.2.3 Базы данных

Для выбора подходящей базы данных для разрабатываемого клиент-серверного приложения был проведен сравнительный анализ нескольких популярных баз данных. В рамках анализа были рассмотрены такие варианты, как PostgreSQL, MongoDB [14] и MySQL [15]. Основные характеристики этих баз данных, а также их сильные и слабые стороны, приведены в таблице 2.5.

Таблица 2.5 – Характеристики баз данных

Характеристики	PostgreSQL	MongoDB	MySQL
Тип базы данных	Реляционная	Документо-ориентированная	Реляционная
Масштабируемость	Хорошая вертикальная и горизонтальная масштабируемость	Отличная горизонтальная масштабируемость (sharding)	Хорошая вертикальная масштабируемость
Производительность	Высокая производительность для сложных запросов	Высокая производительность для больших объемов неструктурированных данных	Хорошая производительность для стандартных SQL-запросов
Интеграция с Java	Отличная интеграция через JDBC и JPA	Хорошая интеграция через драйверы MongoDB для Java	Хорошая интеграция через JDBC
Гибкость схемы	Строгая схема (нужна схема таблицы)	Очень гибкая (без схемы)	Строгая схема (нужна схема таблицы)

Выбор PostgreSQL в качестве базы данных был обусловлен рядом факторов, которые делают её идеальным выбором для разрабатываемого клиент-серверного приложения. PostgreSQL обладает мощными возможностями для работы с реляционными данными, включая полную поддержку SQL, сложные запросы, транзакции и интеграцию с различными типами данных. Это обеспечило высокую гибкость в хранении и управлении данными приложения, что важно для обеспечения надежности и эффективности работы.

2.2.4 Инструменты DevOps

Для хостинга клиент-серверного приложения был выбран облачный сервис Render [16], который предоставляет автоматизированные инструменты для развертывания, масштабирования и управления инфраструктурой. Это решение позволяет разработчикам сосредоточиться на разработке функционала приложения, избавляя от необходимости вручную настраивать серверы и управлять ресурсами. Render поддерживает широкий спектр технологий, таких как Docker, Node.js и Java, что делает его гибким и универсальным решением для хостинга различных типов приложений. Интеграция с GitHub позволяет автоматизировать обновление приложения

при каждом изменении в репозитории. Также, благодаря функции автоматического масштабирования, Render эффективно управляет ресурсами в зависимости от текущей нагрузки, обеспечивая высокую доступность и стабильную производительность системы. На рисунке 2.1 представлена диаграмма компонентов. Стоит добавить, что обращение от репозитория к Model Layer – это не только запрос данных или передача информации, но и получение ответа, который репозиторий использует для выполнения своих операций. Model Layer возвращает репозиторию обработанные объекты, при обращении репозитория к нему, которые можно использовать для подтверждения успешного сохранения или выполнения бизнес-логики.

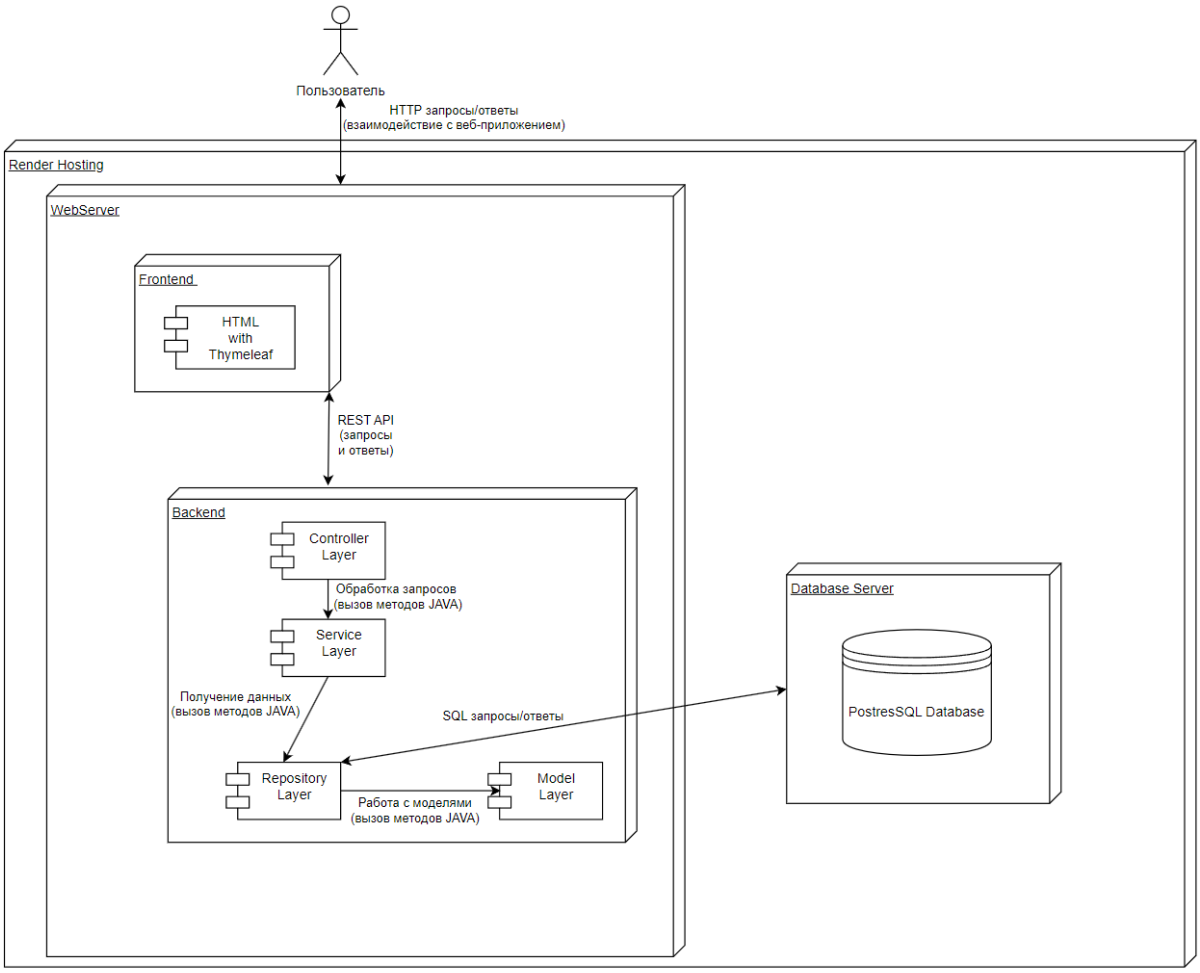


Рисунок 2.1 – Физическая диаграмма компонентов системы

3 РАЗРАБОТКА ПРИЛОЖЕНИЯ

3.1 Структура приложения

Рабочая директория приложения содержит в себе несколько уровней. Серверная часть находиться в директории java/afisha_shop, а клиентская часть, код для страниц, стили, скрипты и изображения, в директории resourses. Подробнее рабочая директория изображена на рисунке 3.1.

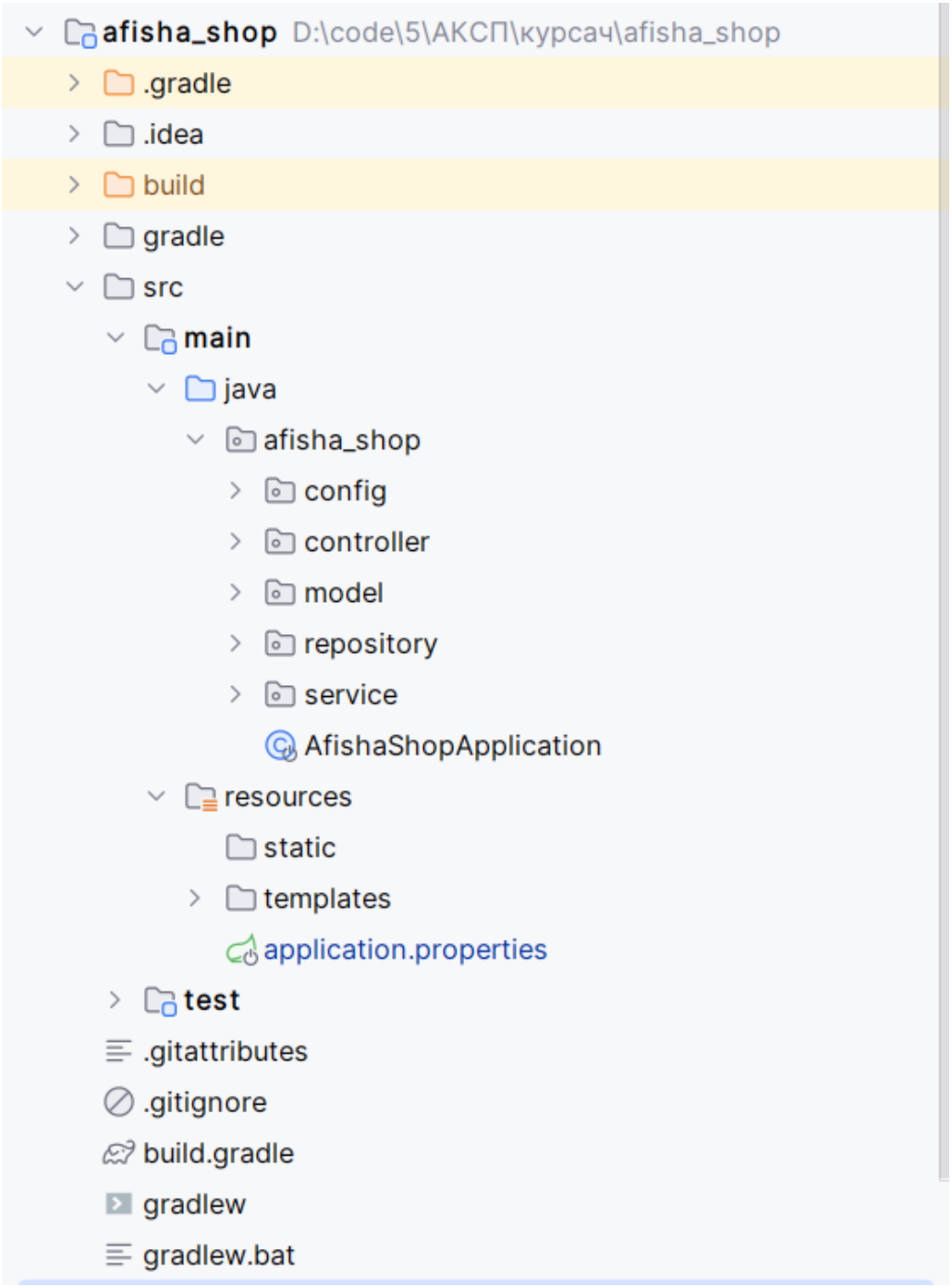


Рисунок 3.1 – Рабочая директория приложения

3.2 Подключение к базе данных

Для осуществления подключения к базе данных приложения требуется внести соответствующие настройки в файл application.properties, как показано на рисунке 3.2. Эти настройки включают строку подключения, а также учетные данные пользователя и пароль для доступа к конкретной базе данных.

```
spring.application.name=afisha_shop
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=password

server.port=8080

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.generate-ddl=true
```

Рисунок 3.2 – Файл application.properties

3.3 Разработка серверной части приложения

Процесс бронирования билетов реализован в приложении с использованием Spring Framework. Для обеспечения функциональности этих процессов в приложении настроены соответствующие контроллеры и сервисы. При регистрации пользователь передает данные (имя, фамилию, email) и выбирает событие, на которое хочет зарегистрироваться. Контроллер вызывает сервис для проверки доступных билетов и, если регистрация успешна, создается новый пользователь с уникальным номером билета. Далее происходит перенаправление на страницу с событиями, где выводится сообщение о успешной регистрации. Реализация показана на рисунках 3.3, 3.4.

```java
// Регистрация на событие
Mark *
@PostMapping("/register")
public String registerUser(
        @RequestParam String firstName,
        @RequestParam String lastName,
        @RequestParam String email,
        @RequestParam Long eventId,
        RedirectAttributes redirectAttributes) { // Используем RedirectAttributes для передачи сообщений
    try {
        Users user = bookingService.registerUser(firstName, lastName, email, eventId);
        redirectAttributes.addFlashAttribute( attributeName: "successMessage"
                , attributeValue: "Вы успешно забронировали билет на: " +
                user.getEvent().getName() +
                ". Уникальный номер вашего билиета: " + user.getTicketNumber() );
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute( attributeName: "errorMessage", e.getMessage());
    }

    return "redirect:/";  // Возвращаем на страницу с событиями
}
```

Рисунок 3.3 – Класс-контроллер бронирования билетов

```java
1 usage  Mark
public Users registerUser(String firstName, String lastName, String email, Long eventId) {
    if (userRepository.existsByEmailAndEventId(email, eventId)) {
        throw new RuntimeException("Пользователь с этой почтой уже зарегистрирован на данное событие.");
    }

    Event event = eventRepository.findById(eventId)
            .orElseThrow(() -> new RuntimeException("Событие не найдено."));

    if (event.getAvailableTickets() <= 0) {
        throw new RuntimeException("Билеты кончились.");
    }

    Users user = new Users();
    user.setFirstName(firstName);
    user.setLastName(lastName);
    user.setEmail(email);
    user.setTicketNumber(UUID.randomUUID().toString());
    user.setEvent(event);

    event.setAvailableTickets(event.getAvailableTickets() - 1);
    eventRepository.save(event);
    return userRepository.save(user);
}
```

Рисунок 3.4 – Сервис для бронирования билетов

Пользователь может найти свои билеты, введя email. Контроллер обрабатывает этот запрос и отображает все найденные билеты, или сообщение об ошибке, если билетов по данному email не существует, рисунок 3.5.

```
Mark *
@GetMapping("/find-tickets")
public String findTicketsByEmail(
        @RequestParam String email,
        RedirectAttributes redirectAttributes) {
    try {
        List<Users> tickets = bookingService.findTicketsByEmail(email);
        if (tickets.isEmpty()) {
            redirectAttributes.addFlashAttribute( attributeName: "errorMessage_search",
                    attributeValue: "No tickets found for email: " + email);
        } else {
            redirectAttributes.addFlashAttribute( attributeName: "tickets", tickets);
        }
    } catch (Exception e) {
        redirectAttributes.addFlashAttribute( attributeName: "errorMessage_search",
                attributeValue: "An error occurred: " + e.getMessage());
    }
    return "redirect:/";  // Остаемся на той же странице
}
```

Рисунок 3.5 – Класс-контроллер поиска билетов по email

Для работы с базой данных, где хранятся данные о событиях и пользователях, используются репозитории, такие как EventRepository для работы с событиями и UserRepository для работы с пользователями. Эти репозитории позволяют эффективно взаимодействовать с базой данных, обеспечивая доступ к необходимым данным, рисунок 3.6, 3.7.

```
2 usages  Mark
public interface UserRepository extends JpaRepository<Users, Long> {

    // Проверка, существует ли пользователь с указанным email и eventId
    1 usage  Mark
    @Query("SELECT CASE WHEN COUNT(u) > 0 THEN true ELSE false END FROM Users u WHERE u.email = :email AND u.event.id = :event_id")
    boolean existsByEmailAndEventId(@Param("email") String email, @Param("event_id") Long eventId);

    1 usage  Mark
    @Query("SELECT u FROM Users u WHERE u.email = :email")
    List<Users> findAllByEmail(@Param("email") String email);
}
```

Рисунок 3.6 – Класс-репозиторий UserRepository

```
6 usages  Mark
public interface EventRepository extends JpaRepository<Event, Long> {
    1 usage  Mark
    List<Event> findAllByOrderByIdAsc();
}
```

Рисунок 3.7– Класс-репозиторий EventRepository

3.4 Клиентская часть приложения

Клиентская часть приложения состоит из страницы, которая предоставляет пользователю возможность взаимодействовать с сервисом для бронирования билетов на различные события, рисунок 3.8.

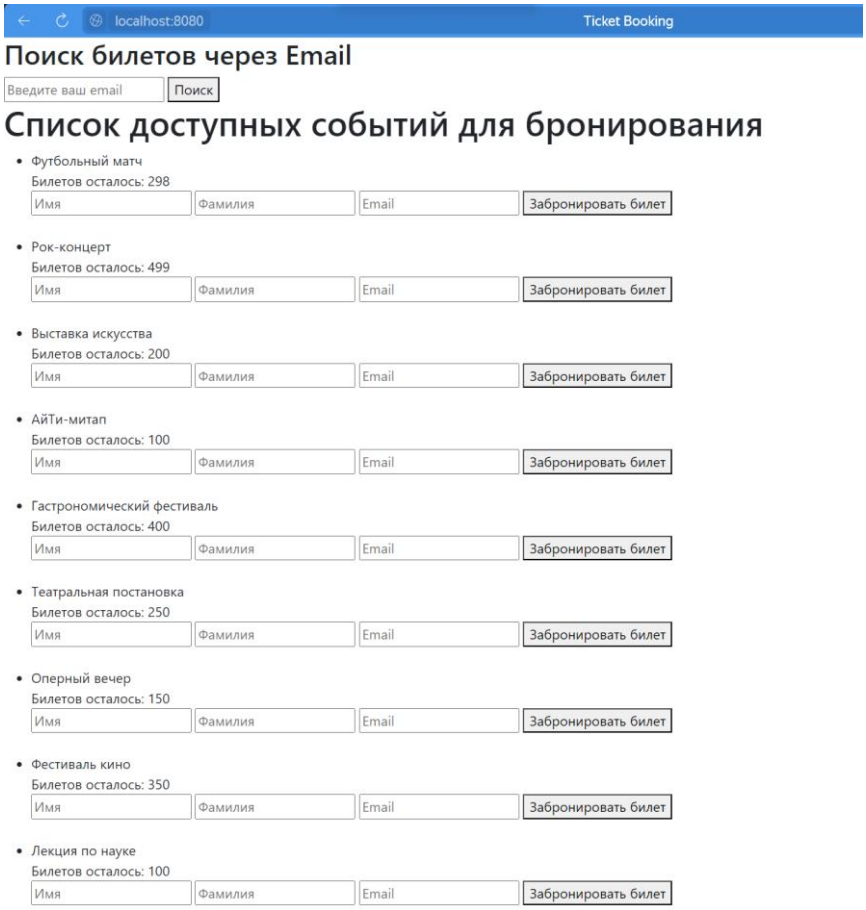


Рисунок 3.8 – Главная страница

Заполнив форму на бронь, появится сообщение об успешной регистрации на мероприятие, рисунок 3.9.

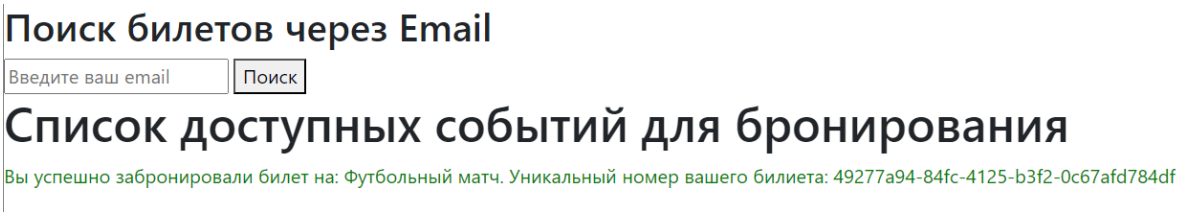


Рисунок 3.9 – Успешная регистрация

В верхней части веб-сервиса есть функция по поиску билетов через email, введя в нее почту, выйдет список мероприятий, на которые

зарегистрировался пользователь (или сообщение о том, что такого билетов у пользователя нет), рисунок 3.10.

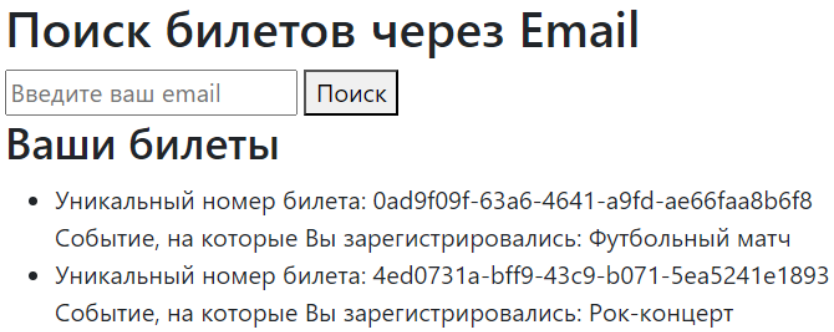


Рисунок 3.10 – Поиск билетов

3.5 Тестирование приложения

Используя программу Postman и браузер, были протестированы несколько конечных точек приложения. Для начала протестируем регистрацию на мероприятие, рисунок 3.11.

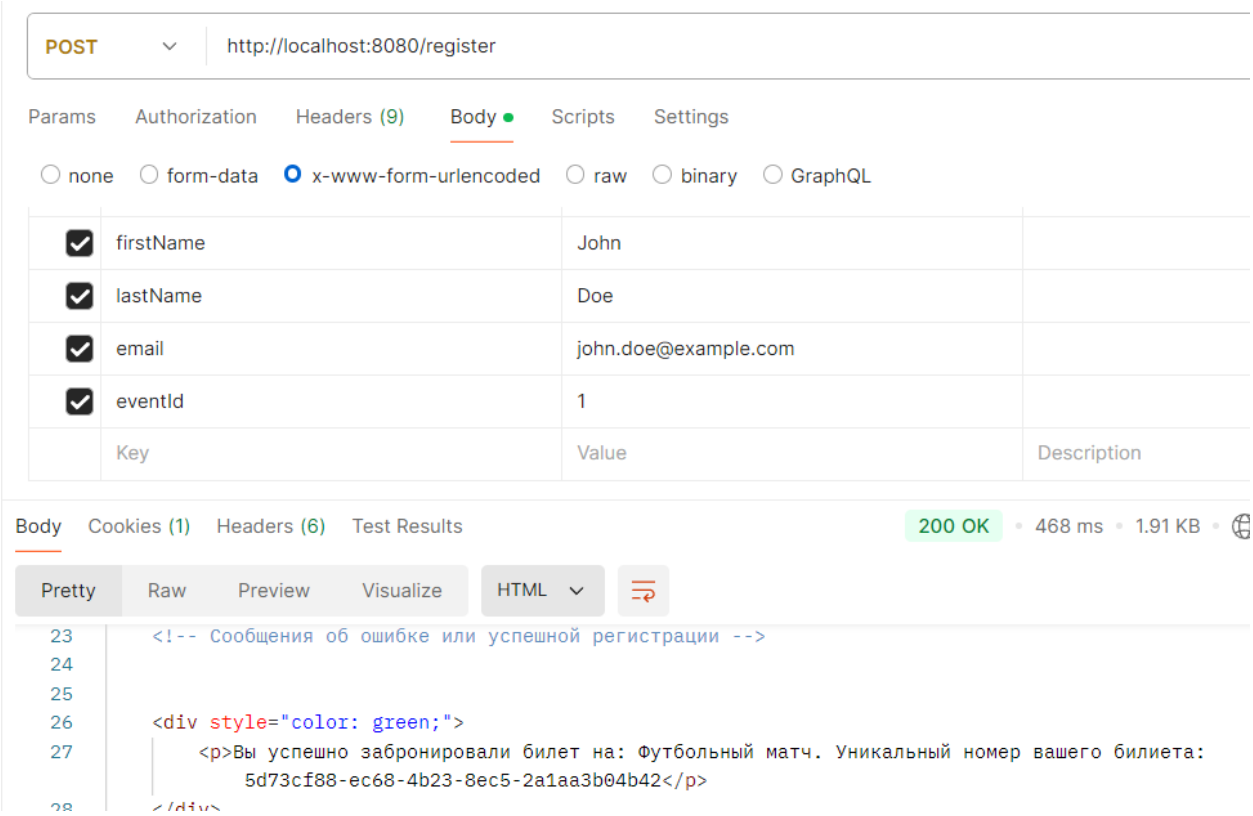


Рисунок 3.11 – POST запрос регистрации на мероприятие

Далее выполним тоже самое ничего не меняя, чтобы отловить ошибку о том, что на 1 почту нельзя дважды купить одно и то же мероприятие, рисунок 3.12.

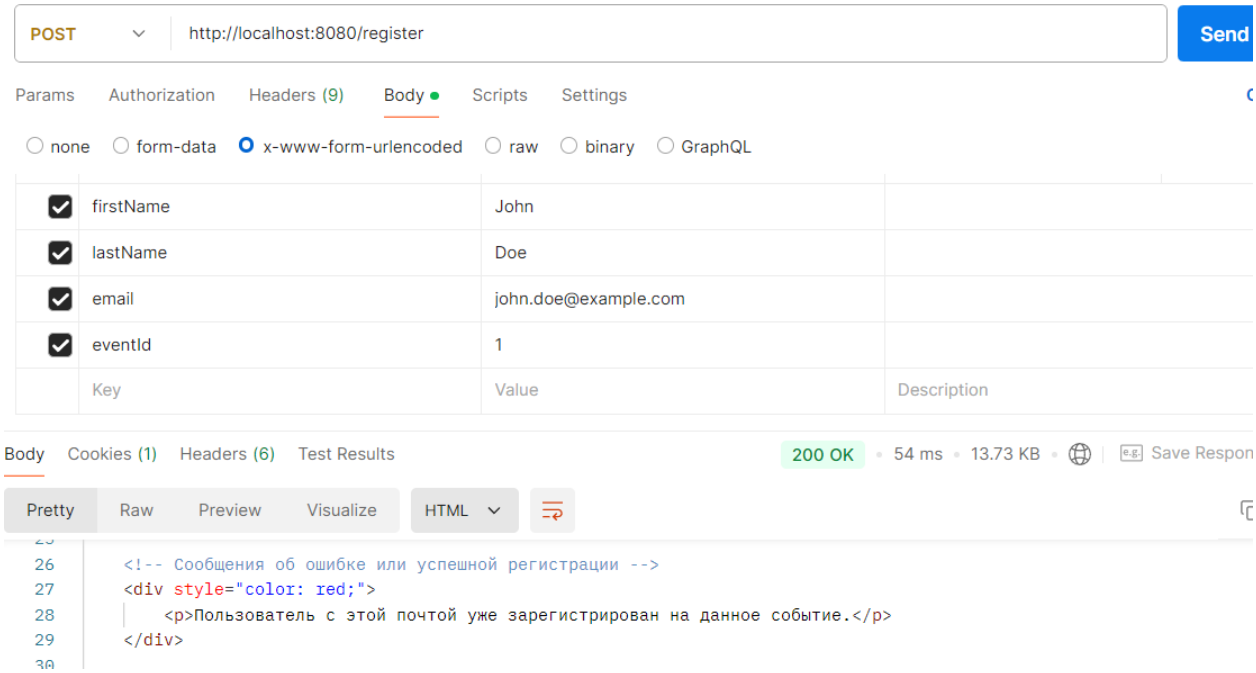


Рисунок 3.12 – POST запрос повторной регистрации на мероприятие

Как мы видим, мы не смогли повторно туда зарегистрироваться, значит все работает отлично. Далее, протестируем возможность забронировать билет, когда на сайте написано, что доступно 0 билетов, рисунок 3.13.

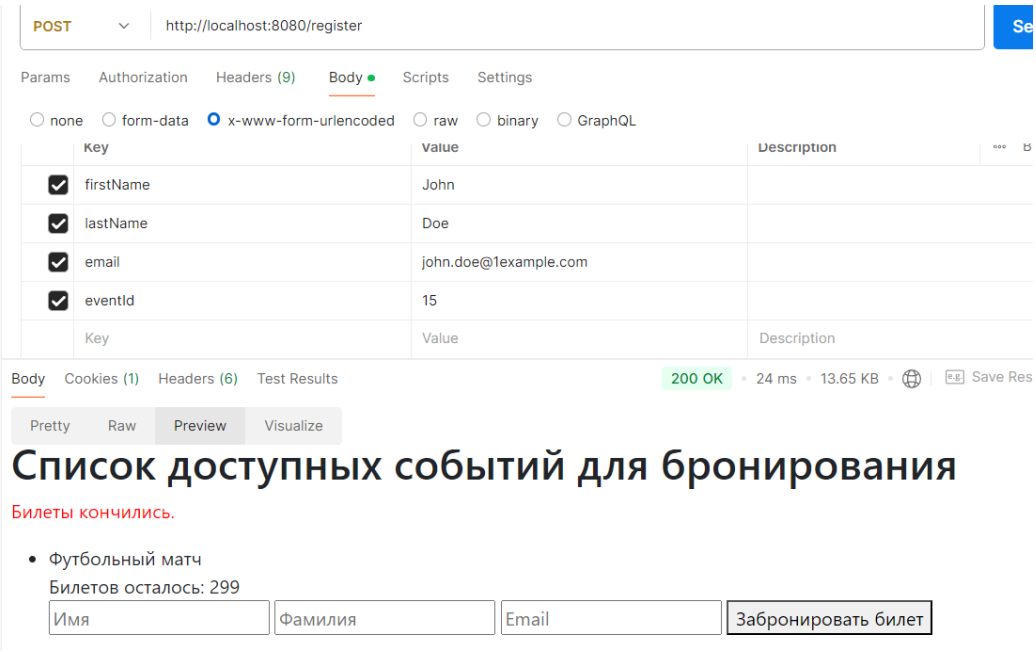


Рисунок 3.13– POST запрос регистрации на мероприятие без билетов

Как можно увидеть, тест прошел успешно. Протестируем функцию поиска билетов по email, рисунок 3.14.

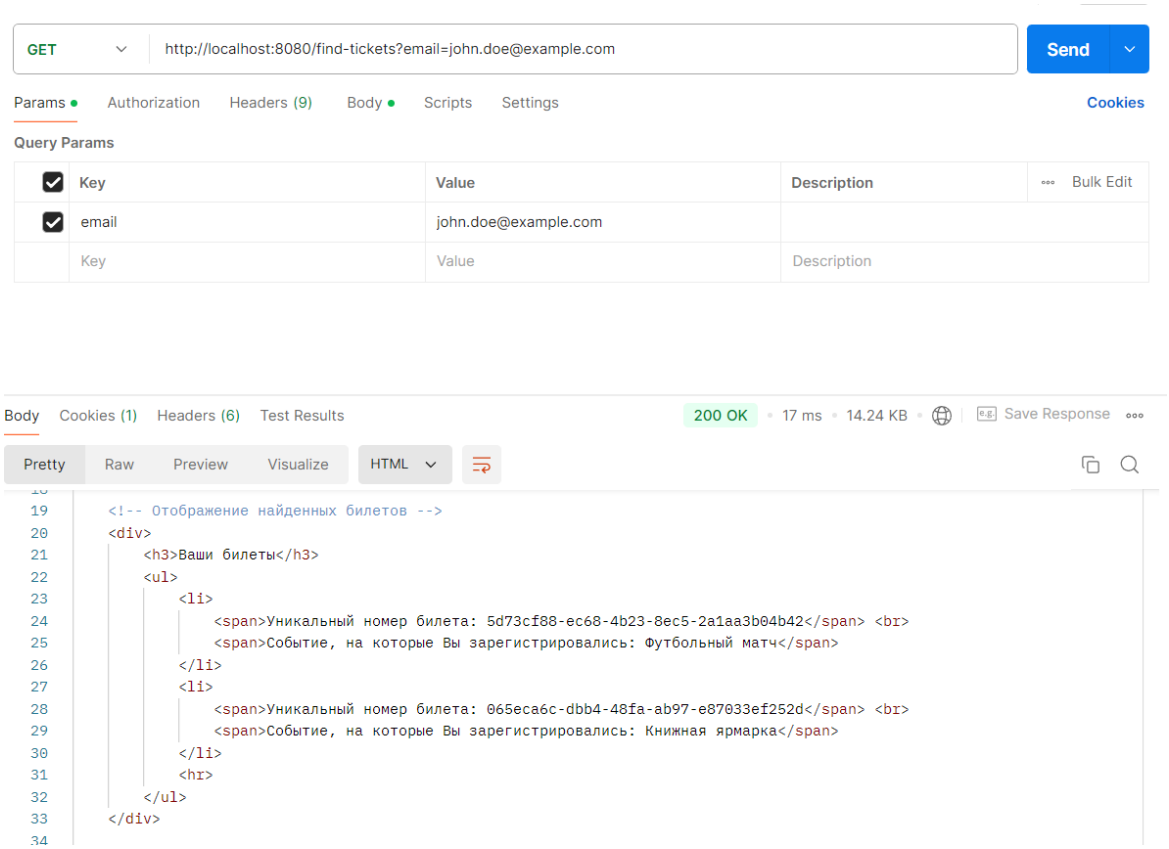


Рисунок 3.14– POST запрос регистрации на мероприятие без билетов

Все конечные точки протестированы и успешно пройдены.

3.6 Контейнеризация

Для развертывания на хостинге приложения, необходимо прописать Dockerfile, для создания контейнера, в котором будет работать разработанное приложения. В Dockerfile прописан основной образ, команда для копирования всех файлов в образ и запуск jar-файла. Содержимое Dockerfile показано на рисунке 3.15.

```dockerfile
# Используем официальный образ OpenJDK как базовый обра
FROM openjdk:23-jdk-slim

# Устанавливаем рабочую директорию в контейнере
WORKDIR /app

# Копируем jar файл вашего приложения в контейнер
COPY . /app/afisha-shop.jar

EXPOSE 8080

# Команда для запуска вашего приложения
ENTRYPOINT ["java", "-jar", "/app/afisha-shop.jar"]
```

Рисунок 3.15 – Файл Dockerfile

ЗАКЛЮЧЕНИЕ

В процессе разработки клиент-серверного приложения для записи пользователей на мероприятия был проведен анализ существующих решений, архитектурных стилей и паттернов проектирования, а также детальное изучение предметной области. На основе полученных данных было разработано приложение с монолитной архитектурой, что позволило ускорить реализацию и упростить разработку. Данный подход оказался наиболее эффективным для приложения с ограниченным функционалом и предполагаемой нагрузкой, обеспечив гибкость и минимизацию сроков разработки.

В результате работы были достигнуты все поставленные цели и задачи: определены ключевые требования и ограничения системы, проанализированы инструменты и средства разработки, создан набор детализированных диаграмм различных нотаций и обоснован выбор архитектуры. Итогом стало разработанное и протестированное приложение, которое демонстрирует стабильную работу, обладает интуитивно понятным интерфейсом и предоставляет пользователям необходимый функционал для записи на мероприятия.

Разработанное приложение было развернуто на локальном хостинге персонального компьютера, но также, при необходимости, может быть развернуто на открытом хостинге от Render.

Для ознакомления с разработанным приложением можно обратиться к исходному коду, который расположен на сайте GitHub: https://github.com/Shumila71/Afisha_shop.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Java Documentation [Электронный ресурс]. – URL: https://docs.oracle.com/en/java/ (дата обращения 10.10.2024).Spring Framework [Электронный ресурс]. – URL: https://spring-projects.ru/projects/spring-framework/ (дата обращения 12.09.2024).

2. Java Spring Boot [Электронный ресурс]. – URL: https://spring.io/projects/spring-boot (дата обращения 12.10.2024).

3. PostgreSQL [Электронный ресурс]. – URL: https://www.postgresql.org/ (дата обращения 14.10.2024).

4. Postman [Электронный ресурс]. – URL: https://www.postman.com/ (дата обращения 15.10.2024).

5. Яндекс афиша [Электронный ресурс]. – URL: https://afisha.yandex.ru/moscow/ (дата обращения 21.09.2024).

6. Ticketland [Электронный ресурс]. – URL: https://www.ticketland.ru/ (дата обращения 21.09.2024).

7. MTC live [Электронный ресурс]. – URL: https://live.mts.ru/moscow/collections/concerts (дата обращения 21.09.2024).

8. UML-диаграммы [Электронный ресурс]. – URL: https://www.uml-diagrams.org/ (дата обращения 22.10.2024).

9. Draw.io [Электронный ресурс]. – URL: https://app.diagrams.net/ (дата обращения 22.10.2024).

10. VS Code [Электронный ресурс]. – URL: https://code.visualstudio.com/ (дата обращения 28.10.2024).

11. IntelliJ IDEA [Электронный ресурс]. – URL: https://www.jetbrains.com/idea/ (дата обращения 27.09.2024).

12. Django [Электронный ресурс]. – URL: https://www.djangoproject.com/ (дата обращения 25.10.2024).

13. Thymeleaf [Электронный ресурс]. – URL: https://www.thymeleaf.org/ (дата обращения 27.10.2024).

14. MongoDB [Электронный ресурс]. – URL: https://www.mongodbmanager.com/ (дата обращения 13.11.2024).

15. MySQL [Электронный ресурс]. – URL: https://www.mysql.com/ (дата обращения 14.11.2024).

16. Render Quickstarts [Электронный ресурс]. – URL: https://docs.render.com/ (дата обращения 15.11.2024).