МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Институт (Филиал) № 8 «Компьютерные науки и прикладная математика»

Кафедра 806

Группа М8О-406Б-30

Направление подготовки 01.03.02 Прикладная математика и информатика

Профиль Информатика

Квалификация бакалавр

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

На тему: Web-приложение по оптимизации продовольственных процессов в МАИ

Автор ВКРБ Попов Илья Павлович (подпись)
(фамилия, имя, отчество полностью)

Руководитель Чернышов Лев Николаевич (подпись)
(фамилия, имя, отчество полностью)

Консультант _____ (_____)
(фамилия, имя, отчество полностью)

Консультант _____ (_____)
(фамилия, имя, отчество полностью)

Рецензент _____ (_____)
(фамилия, имя, отчество полностью)

К защите допустить

Заведующий кафедрой 806 Крылов Сергей Сергеевич (_____)
(№ каф) (фамилия, имя, отчество полностью)

_____ 2024г.

Москва 2024

РЕФЕРАТ

Выпускная квалификационная работа бакалавра содержит 45 страниц, 22 рисунка, 1 таблицу, 12 использованных источников.

JAVA, SPRING BOOT, SPRING CLOUD, EUREKA SERVER, API GATEWAY, POSTGRESQL, APACHE KAFKA, KEYCLOAK, JWT, ANGULAR, IONIC, WEB-ПРИЛОЖЕНИЕ

Разработанный проект направлен на решение актуальной проблемы, связанной с неспособностью студентов воспользоваться обеденным перерывом в полном объеме из-за длительных очередей, что влечет за собой быстрое употребление пищи и негативное воздействие на здоровье. Кроме того, проект предоставляет студентам возможность заработка в обеденное время путем выполнения доставки из столовых в различные корпуса института.

Целью данной работы звучит так – повысить качество сервиса заведений общественного питания в университетских кампусах за счет внедрения мер, направленных на сокращение времени ожидания в очереди и обеспечение доступности полноценного питания во время обеденного перерыва для поддержания здоровья и благополучия студентов.

В результате разработки создано приложение, позволяющее студентам заказывать блюда из столовых, минимизируя время ожидания, и наслаждаться обеденным перерывом, не тратя время на очереди.

Содержание работы разделено на три главы:

В первой освещены методы и технологии создания веб-приложений, проведен обзор и анализ существующих аналогичных решений.

Во второй освещается стек технологий, применяемых для реализации приложения.

В третьей осуществлено техническое проектирование, описание разработанной архитектуры системы и интерфейса.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Главная проблема, которую будет решать данный проект заключается в отсутствии возможности у студентов в полном объеме воспользоваться обеденным перерывом, в связи с продолжительными очередями, что вынуждает их прибегать к быстрому приему пищи, что, в свою очередь, может оказывать негативное воздействие на желудочно-кишечный тракт подростков.

Помимо этого, рассматриваемый проект даст возможность студентам, у которых есть время в обеденный перерыв (отсутствие пар и прочее) немного заработать, выполнив несложную доставку из столовых до одного из корпусов МАИ.

Цель, которая была поставлена звучит так – повысить качество сервиса заведений общественного питания в университетских кампусах за счет внедрения мер, направленных на сокращение времени ожидания в очереди и обеспечение доступности полноценного питания во время обеденного перерыва для поддержания здоровья и благополучия студентов.

Задачи, стоящие перед проектом:

– Определить текущую ситуацию на рынке доставки пищи, проанализировав деятельность конкурирующих компаний;

– Выбрать архитектурный подход написания приложения;

– Формализовать граф пользовательского взаимодействия с приложением;

– Реализовать микросервисную архитектуру приложения;

– Настроить окружение для микросервисной архитектуры (api gateway и load balancer);

– Реализовать взаимодействие с хранилищами данных – PostgreSql для статичных данных и Apache Kafka для поступающих заказов;

– Реализовать бизнес-логику микросервисов user-service, delivery-service, menu-service;

– Реализовать надежную защиту с помощью OAuth2 протокола авторизации на стороне backend;

– Реализовать backend for frontend сервис для обеспечения безопасности OAuth2 токенов на стороне браузера;

– Реализовать frontend приложения с использованием фреймворка Ionic, адаптированный для мобильных платформ.

Результатом разработки стало создание приложения, обладающего функционалом автоматизации и управления процессом доставки для студентов Московского авиационного института. Приложение позволяет студентам удобно заказывать и блюда из столовых, минимизируя время ожидания. Благодаря внедренной платформе доставки обеспечивается возможность приема пищи, не выходя из корпуса.

В результате использования приложения студенты получают возможность полноценно насладиться обеденным перерывом, избегая необходимости стоять в длинных очередях и спешить с приемом пищи. Благодаря реализованной микросервисной архитектуре и надежной защите данных, приложение обеспечивает высокий уровень безопасности и устойчивость к внешним воздействиям.

1 ГЛАВА

1.1 Оценка обстановки на рынке

1.1.1 Причины отсутствия аналогов

На рынке отсутствуют приложения с аналогичной концепцией. Их отсутствие может быть связано с юридическими сложностями, которые могут возникнуть вследствие некомпетентного или злоумышленного использования платформы для заказа и доставки товаров, в которой доставщиками могут быть люди, не подписавшие никаких юридических договоров.

Внедрение такого приложения подразумевает определенную степень ответственности за качество услуг и безопасность пользователей. Существует риск возникновения проблемных ситуаций, таких как потеря товара, неправильная доставка или возможные мошеннические действия.

Однако в данном случае есть ряд специфических особенностей, присущих только студенческой среде, таких как солидарность и поддержка внутри сообщества: среди студентов сильнее прослеживается атмосфера взаимопомощи и поддержки, нежели в других социальных группах, что может смягчить возможные риски.

Несмотря на отсутствие прямых конкурентов стоит отметить компании, занимающиеся доставкой, так как это половина функциональности рассматриваемого приложения.

1.1.2 Сравнение ценовой политики

К сожалению, многие компании указывают только наименьшие стоимости своих доставок, в реальности ситуация разительно отличается. Также отличается процент, который идет непосредственно курьеру от стоимости всей доставки.

Анализ трудозатрат и вознаграждения предоставляет убедительные данные о преимуществах работы студентов через данное приложение по сравнению с курьерами крупных компаний по доставке еды.

Таблица 1 – Анализ ценовой политики

Название компании-доставщика	Цена доставки, руб. (чистая цена доставки, с учетом среднего процента надбавки к продукту)	Среднее время заказа, мин. (зависит от площади покрытой компанией территории)
Яндекс.Еда	350 – 400	40 – 60
Утконос	300 – 400	20 – 40
IGoods	400 – 500	60 – 90
Самокат	250	30 – 60
Сбермаркет	200	от 60

При средней цене доставки в Москве в размере 350 рублей, курьеры, которые доставляют заказы за 40 минут – 1 час, в среднем получают на руки около 40 – 50% суммы, то есть за минуту работы до 4,5 рублей.

В данном приложении студент сможет получать вознаграждение в размере 10% от стоимости заказа, следовательно, за доставку среднего заказа на 600 рублей, потратив на это лишь 10 минут, то его заработок за минуту будет 6 рублей значительно выше, чем у курьеров крупных компаний.

Кроме того, экономический стимул для кафе внедрить рассматриваемое приложение заключается в том, что студенты могут тратить свои заработанные деньги исключительно внутри этой платформы. Это создает уникальную возможность привлечения новых клиентов для кафе и обеспечивает им стабильный поток заказов. Факт того, что студенты могут расходовать свои средства только в кафе, подключенных к приложению, выступает как дополнительный стимул для заведений присоединиться к платформе.

Также, конкуренция между кафе за предпочтение студентов создает дополнительную динамику на рынке и вынуждает заведения предлагать более выгодные условия и качественный сервис. Это приводит к улучшению обслуживания для конечных пользователей и расширению выбора кафе, что в конечном итоге приносит пользу клиентам.

1.2 Архитектурный подход

1.2.1 REST

REST (от англ. ***RE****presentational* ***S****tate* ***T****ransfer* — «передача репрезентативного состояния») — архитектурный стиль взаимодействия компонентов распределенного приложения в сети[1].

1.2.1.1 Свойства архитектуры REST

Архитектура REST подразумевает следующие свойства:

– Производительность: достигается благодаря эффективному взаимодействию компонентов системы, что обеспечивается единообразием и согласованностью запросов и ответов между клиентом и сервером

– Масштабируемость: благодаря ряду требований, наложенных на данную архитектуру, она обеспечивает возможность расширения и поддержки большого количества компонентов

– Гибкость: архитектура REST позволяет легко изменять и модифицировать компоненты системы благодаря своей децентрализованной и модульной структуре, что способствует адаптации к изменяющимся требованиям и условиям.

– Надежность: системы, построенные с использованием архитектуры REST, обладают высокой степенью надежности благодаря использованию стандартных протоколов и методов, а также возможности обработки ошибок и восстановления после сбоев.

1.2.2 Микросервисная архитектура

Микросервисная архитектура — вариант сервис-ориентированной архитектуры программного обеспечения, направленный на взаимодействие

насколько это возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов, получивший распространение в середине 2010-х годов в связи с развитием практик гибкой разработки и DevOps[2]. Сравнение микросервисной и монолитной архитектуры представлено на рисунке 1.

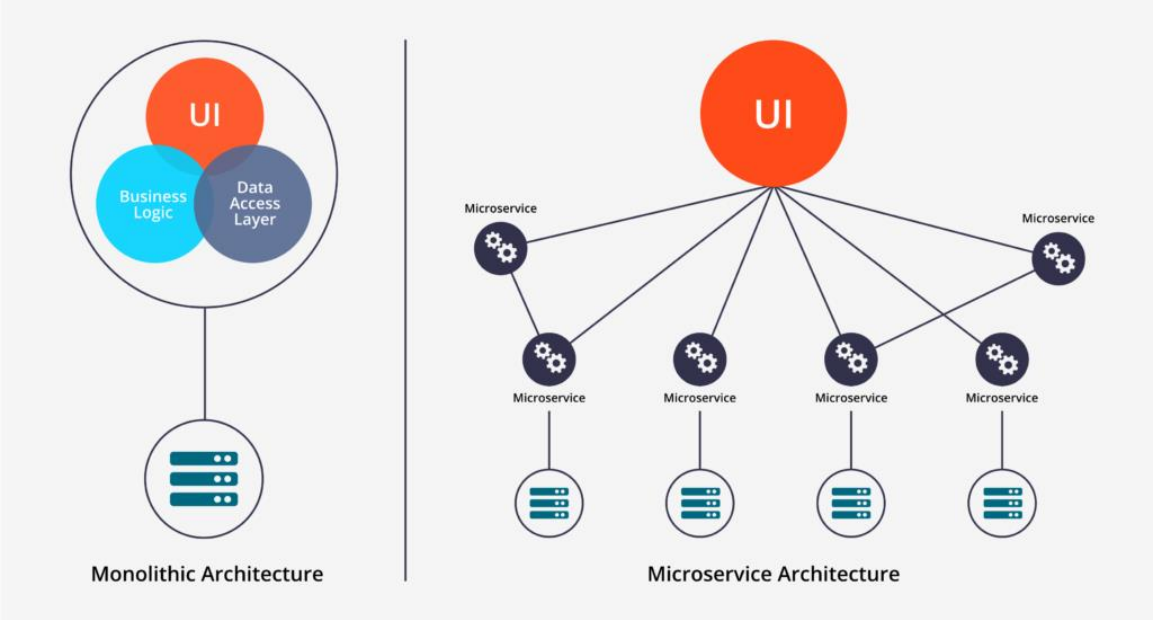


Рисунок 1 – Примеры микросервисной и монолитной архитектуры

1.2.2.1 Свойства микросервисной архитектуры

Свойства, характерные для микросервисной архитектуры:

– Легкая заменяемость: микросервисы можно легко заменить или переписать без значительного влияния на остальную систему. Это позволяет быстро адаптироваться к изменяющимся требованиям бизнеса или технологическим инновациям;

– Изоляция сбоев: поскольку каждый микросервис выполняется в собственном процессе или контейнере, сбои в одном сервисе не распространяются на остальные. Это повышает отказоустойчивость системы в целом и облегчает обнаружение и управление проблемами;

– Гибкое использование технологий: микросервисная архитектура позволяет использовать различные технологии и языки программирования для каждого сервиса в зависимости от их специфических требований. Это

позволяет оптимизировать производительность и эффективность каждого сервиса в рамках системы;

– Гибкость и легкость в развертывании: микросервисы обычно разрабатываются, управляются и развертываются независимо друг от друга. Это обеспечивает гибкость внесения изменений в систему, так как каждый сервис может быть обновлен и перезапущен отдельно от остальных;

– Самостоятельность и независимость: каждый микросервис может иметь собственную базу данных, свою собственную команду разработки и развертываться на отдельных серверах или контейнерах. Это позволяет избежать проблем совместного использования ресурсов и минимизировать влияние сбоев на другие компоненты системы;

– Простота масштабирования разработки: микросервисы позволяют командам разработчиков работать над различными компонентами системы независимо друг от друга, что способствует параллельной разработке и ускоряет процесс создания итераций продукта. Сравнение различий в подходе к масштабированию микросервисной и монолитной архитектуры представлено на рисунке 2.

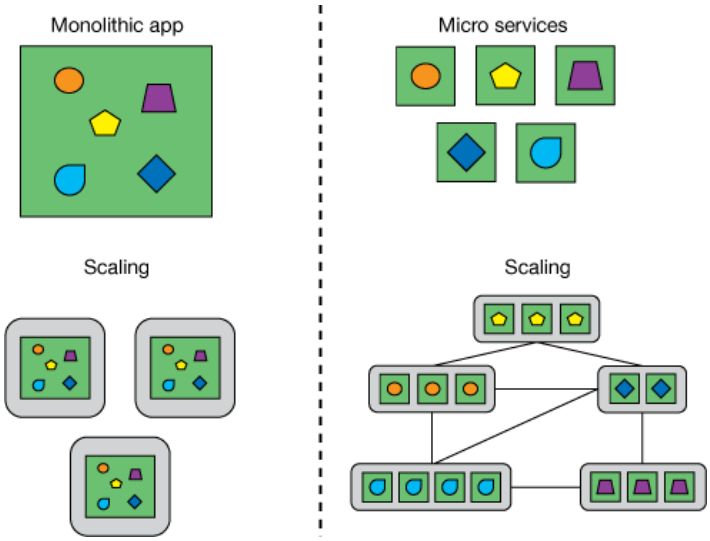


Рисунок 2 – Демонстрация различия в подходе к масштабированию монолитных и микросервисных приложений

1.2.2.2 Преимущества горизонтальной масштабируемости

Различают два типа масштабируемости: вертикальную и горизонтальную.

Вертикальная – увеличение ресурсов на одном сервере, но ограничена возможностями оборудования.

Горизонтальная – добавление новых серверов для распределения нагрузки, не имеет ограничений в количестве добавляемых серверов.

Сравнение перспектив горизонтального и вертикального масштабирования представлено на рисунке 3.

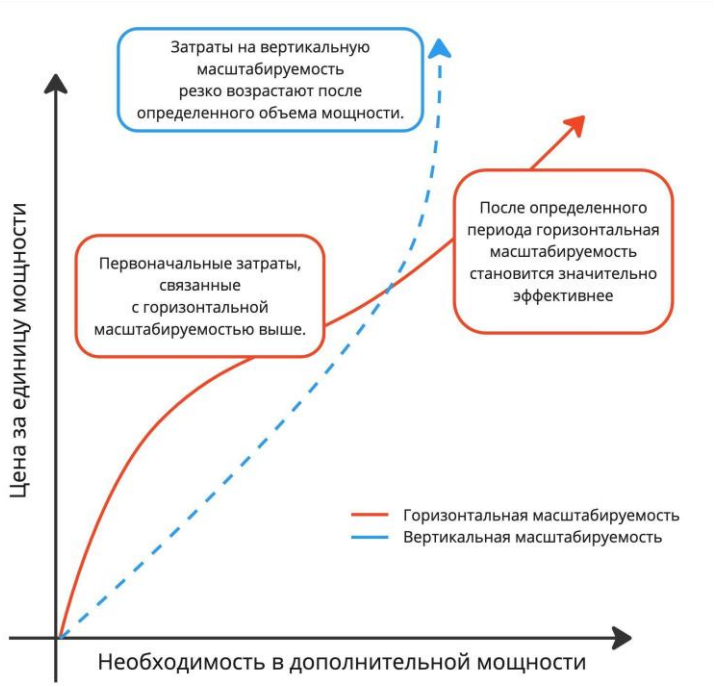


Рисунок 3 – После определенного порога горизонтальное масштабирование становится гораздо дешевле вертикального

В начальной стадии горизонтальное масштабирование может обходиться дороже из-за необходимости покупки и поддержки нескольких серверов. Однако, в долгосрочной перспективе, при увеличении нагрузки, расширение вертикального масштабирования может стать многократно дороже и менее эффективным. Это обусловлено ограничениями ресурсов

одного сервера и потребностью в приобретении более дорогостоящего оборудования.

1.2.3 Алгоритмы распределения нагрузки

Одним из ключевых вопросов, стоящих перед архитекторами распределенных систем, является эффективное распределение нагрузки между различными экземплярами микросервисов. В этом контексте, алгоритмы балансировки нагрузки, такие как Round Robin, Weighted Round Robin и Least Connections, играют важную роль, обеспечивая равномерное распределение запросов и оптимизацию производительности системы [3].

1.2.3.1 Round Robin (RR)

Этот алгоритм распределяет запросы по всем доступным серверам или сервисам по очереди. Каждый новый запрос направляется к следующему серверу в списке. После обслуживания последнего сервера, цикл повторяется снова с первого сервера. Round Robin характеризуется простотой и равномерностью распределения нагрузки, но не учитывает различия в производительности или нагрузке между серверами.

1.2.3.2 Weighted Round Robin (WRR)

Этот вариант алгоритма Round Robin дополнен весовыми коэффициентами для каждого сервера или сервиса. Каждый сервер получает вес, который определяет его долю запросов в общей нагрузке. Серверы с более высоким весом получают больше запросов, чем серверы с более низким весом. Этот подход позволяет учитывать различия в производительности или мощности серверов, обеспечивая более эффективное распределение нагрузки.

1.2.3.3 Least Connections

Этот алгоритм направляет запрос к серверу с наименьшим количеством активных соединений в данный момент времени. Он учитывает текущую нагрузку на серверы и направляет запросы туда, где количество активных соединений минимально. Это позволяет избегать перегрузок на отдельных серверах и обеспечивать равномерное распределение нагрузки даже при различной интенсивности запросов.

Таким образом, эффективное взаимодействие между API Gateway и указанными алгоритмами становится краеугольным камнем для создания высокопроизводительных и масштабируемых распределенных систем.

1.2.3.4 Оценка алгоритмов

Зачастую, особое внимание уделяется временным задержкам, которые определяются интервалом времени от момента создания запроса до его обработки, измеряемым в миллисекундах. При анализе данных задержек в данном контексте часто упоминаются различные процентили. Например, 50-й перцентиль, также известный как медиана, представляет собой значение в миллисекундах, ниже которого находится 50% запросов (рисунок 4).

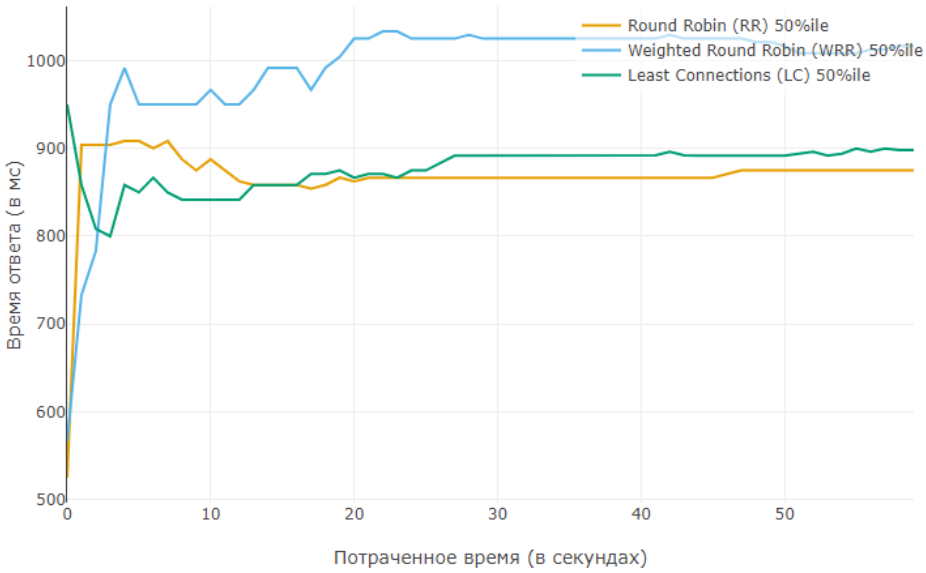


Рисунок 4 – Анализ задержек между алгоритмами Round Robin, Weighted Round Robin и Least Connections на 50 процентили

Round Robin демонстрирует наиболее низкую медианную задержку. Однако, для полноты картины важно рассмотреть также 95-й и 99-й перцентили (рисунок 5).

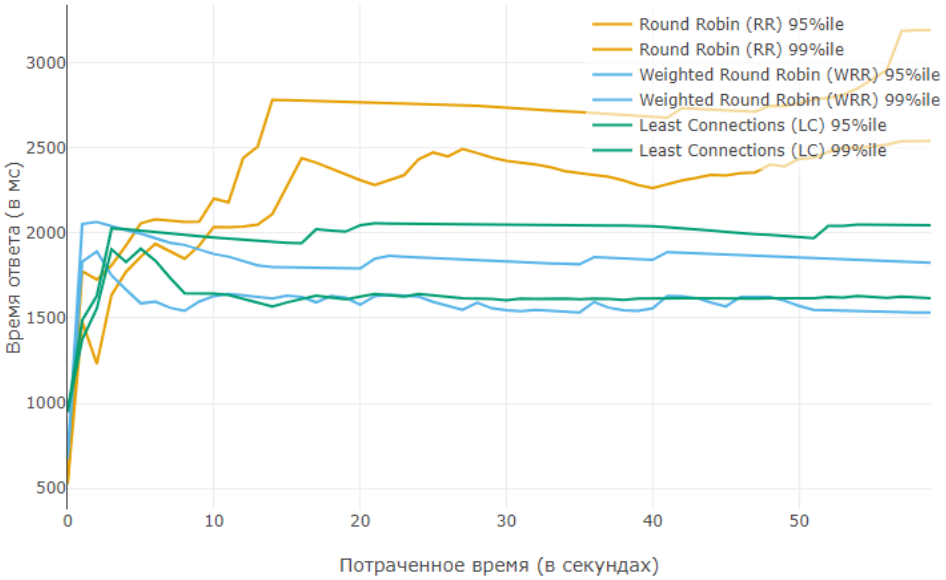


Рисунок 5 - Анализ задержек между алгоритмами Round Robin, Weighted Round Robin и Least Connections на 95 и 99 процентили

Анализ данных указывает на недостаточное качество работы алгоритма Round Robin при высоких перцентилях. Это обусловлено тем, что алгоритм не учитывает текущее состояние каждого сервера, что может привести к направлению значительной части запросов на серверы, находящиеся в состоянии простоя, что обуславливает низкий 50-й перцентиль. В то же время, запросы могут быть направлены на перегруженные серверы, что приводит к неудовлетворительным значениям 95-го и 99-го перцентилей.

Несмотря на указанные ограничения, алгоритм Round Robin обычно применяется в случаях, когда требуется более низкая надежность, но при этом важна простота настройки и реализации.

1.2.3 Сервер авторизации

Сервер авторизации – это программное обеспечение или служба, ответственная за управление процессом аутентификации пользователей и предоставление им доступа к защищенным ресурсам или услугам. Основная задача сервера авторизации заключается в проверке подлинности идентификационных данных пользователя (логин и пароль, цифровые сертификаты, токены и т. д.) и выдаче учетной записи сессии или токена

доступа, который затем используется для аутентификации пользователя и авторизации его запросов к защищенным ресурсам или службам [4]. Пример схемы взаимодействия пользователя, клиентского приложения, resource server’а и сервера авторизации представлен на рисунке 6.

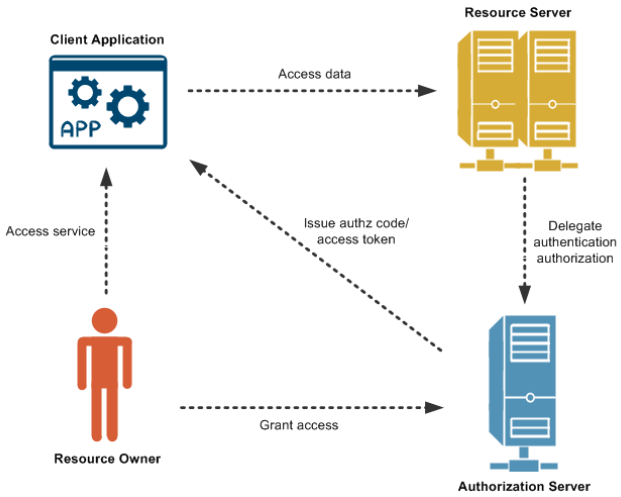


Рисунок 6 – Схема взаимодействия пользователя, клиентского приложения, resource server’а и сервера авторизации

1.2.4 Необходимость кроссплатформенного приложения

Учитывая данные, представленные на рисунке 7[5], и широкое распространение мобильных операционных систем Android и iOS по всему миру, разработка нативных приложений под одну из этих платформ может быть нецелесообразной вследствие того, что возникает риск потерять значительную часть потенциальных пользователей, использующих другую операционную систему.

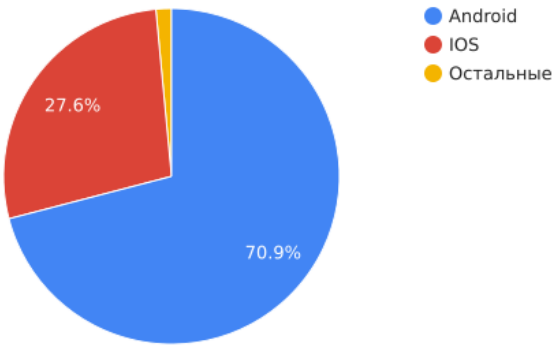


Рисунок 7 – Распределение мобильных ОС у пользователей по всему миру

Учитывая потребности современного рынка и требования пользователей к универсальности и доступности, для пользовательского интерфейса был выбран кроссплатформенный подход.

Использование кроссплатформенных технологий, таких как Ionic, представляется наиболее эффективным способом достижения этой цели. Однако следует отметить, что кроссплатформенные решения могут немного проигрывать в быстродействии по сравнению с нативными приложениями, разработанными специально для определенной платформы. Это обусловлено тем, что кроссплатформенные фреймворки добавляют дополнительный уровень абстракции и могут иметь небольшую задержку при рендеринге интерфейса и выполнении некоторых операций.

1.2.5 Требования к разрабатываемому приложению

Результатом разработки должно стать приложение, обладающее функционалом автоматизации и управления процессом доставки для студентов Московского авиационного института. Приложение должно предоставлять студентам функционал удобно заказывать блюда из столовых, минимизируя время ожидания. Благодаря внедренной платформе доставки

должна обеспечивается возможность получения пищи непосредственно в корпус заказчика.

Приложение должно обладать такими качествами, как простота в использовании, отказоустойчивость и масштабируемость. Оно также обеспечивает транзакционность и обладает надежной защитой от несанкционированных вмешательств извне.

2 ГЛАВА

В этой главе будут приведены технологии, которые были использованы при реализации приложения; будут рассмотрены серверные и клиентские технологии, базы данных, системы контроля версий и другие инструменты, обеспечивающие полный цикл разработки. Подробный анализ каждого компонента позволит понять его роль, преимущества и вклад в достижение целей проекта.

2.1 Spring-фреймворк

Spring Framework – это ведущий фреймворк для разработки приложений на Java, который предоставляет обширный набор инструментов и функциональности [6]. Он облегчает создание надежных и масштабируемых приложений благодаря концепции инверсии управления и внедрению зависимостей. С помощью Spring можно эффективно работать с базами данных, веб-технологиями, безопасностью и другими аспектами приложения. Фреймворк поддерживает аспектно-ориентированное программирование (AOP) и обладает модульной архитектурой.

2.2 Eureka server

Eureka Server, разработанный компанией Netflix, является основой для организации микросервисной архитектуры. Данный инструмент предназначен для обеспечения масштабируемости и устойчивости системы путем обеспечения динамической регистрации, нахождения и управления микросервисами в распределенной среде [7].

В основе работы Eureka Server лежит принцип клиент-серверной архитектуры, где сервер предоставляет централизованный реестр микросервисов, а клиенты, в свою очередь, регистрируются в этом реестре и могут обращаться к нему для поиска необходимых сервисов (рисунок 8).

Основной функционал Eureka Server включает в себя обнаружение новых микросервисов, удаление недоступных сервисов из реестра, а также предоставление информации о доступности сервисов клиентам.

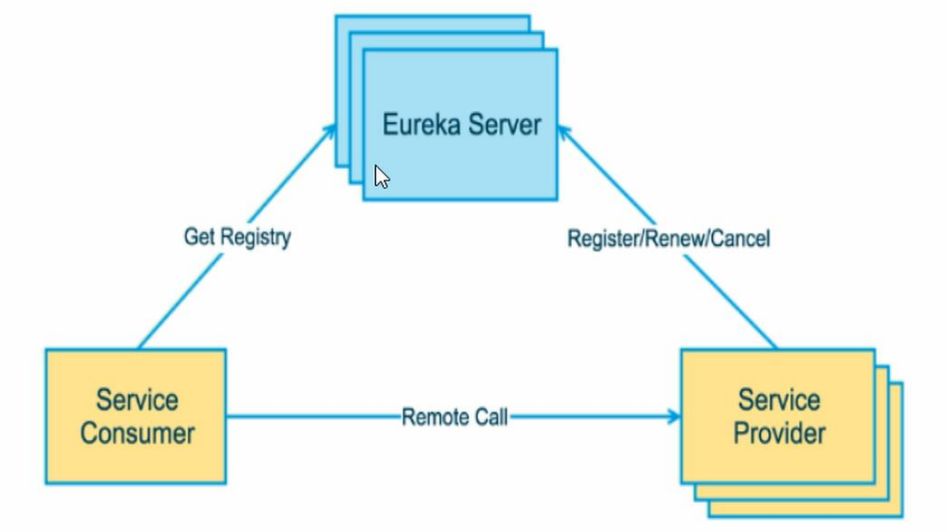


Рисунок 8 – Схема работы Netflix Eureka server

2.3 Spring Cloud Gateway

API Gateway – это высокоуровневый компонент архитектуры микросервисов, который выполняет ряд ключевых функций для обеспечения эффективного управления и маршрутизации запросов к микросервисам. Одним из основных применений API Gateway является реализация механизмов балансировки нагрузки (load balancing), которые позволяют равномерно распределять запросы между экземплярами микросервисов для обеспечения оптимальной производительности и надежности системы [8].

Это достигается путем использования различных алгоритмов балансировки нагрузки, таких как round-robin или weighted round-robin, которые учитывают текущую нагрузку на каждый экземпляр микросервиса.

Дополнительно, API Gateway предоставляет механизмы отслеживания доступности микросервисов. Это позволяет автоматически мониторить состояние каждого сервиса и выявлять недоступные или неисправные экземпляры. При обнаружении проблем API Gateway может перенаправлять запросы на другие доступные экземпляры или предпринимать другие необходимые действия для обеспечения непрерывной работы системы.

В данной работе применена реализация API Gateway от Spring Cloud, ее схема приведена на рисунке 9.

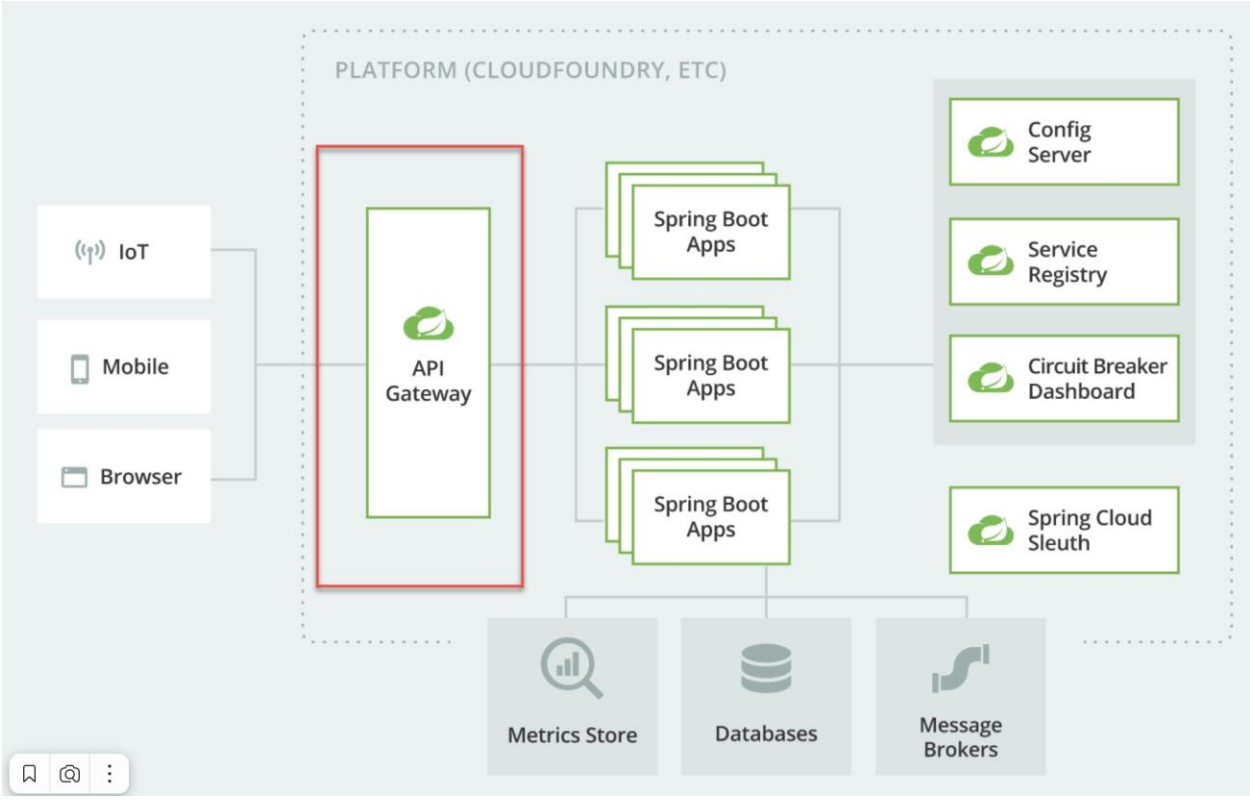


Рисунок 9 – Схема работы Spring Cloud Gateway

2.4 Apache Kafka

2.4.1 Общие сведения

Apache Kafka — распределенный программный брокер сообщений с открытым исходным кодом, разрабатываемый в рамках фонда Apache на языках Java и Scala. Цель проекта — создание горизонтально масштабируемой платформы для обработки потоковых данных в реальном времени с высокой пропускной способностью и низкой задержкой. Использует собственный двоичный протокол передачи данных на основе TCP, группирующий сообщения для снижения накладных расходов на сеть [9].

2.4.2 Архитектура

Kafka хранит сообщения, которые поступают от других процессов, называемых «производителями» (producers), в формате «ключ — значение». Данные могут быть разбиты на разделы (англ. partitions, визуализация представлена на рисунке 10) в рамках разных тем (topics). Внутри раздела сообщения строго упорядочены по их смещениям (offset), то есть по положению сообщения внутри раздела, а также индексируются и сохраняются

вместе с временем создания. Другие процессы, называемые «потребителями» (consumers), могут считывать сообщения из разделов.

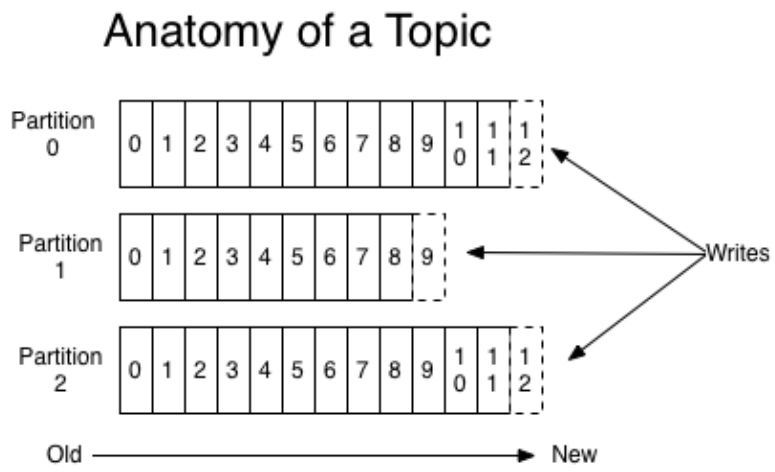


Рисунок 10 – Устройство партиций Apache Kafka

Система работает в кластере из одного или нескольких узлов-брокеров, где разделы всех тем распределены по узлам кластера. Для обеспечения отказоустойчивости разделы реплицируются на несколько брокеров.

Kafka поддерживает два типа тем: обычные и компактные. Обычные темы можно настроить с указанием срока хранения или ограничения по максимальному занимаемому пространству.

2.5 KeyCloack

В качестве сервера авторизации был использован KeyCloack. Он представляет собой открытое программное обеспечение, предназначенное для управления аутентификацией и авторизацией в распределенных системах. Этот инструмент предоставляет механизмы централизованного управления доступом к ресурсам, обеспечивая безопасность и контроль в среде микросервисной архитектуры. Keycloak поддерживает различные протоколы аутентификации и авторизации, такие как OAuth 2.0 и OpenID Connect, что делает его гибким решением для различных сценариев авторизации и аутентификации [10].

Authorization Code Flow (поток с кодом авторизации) – это спецификация протокола OAuth 2.0, используемая для безопасного получения

доступа к ресурсам от имени пользователя через веб-приложения. Схема работы Authorization Code Flow представлена на рисунке 11.

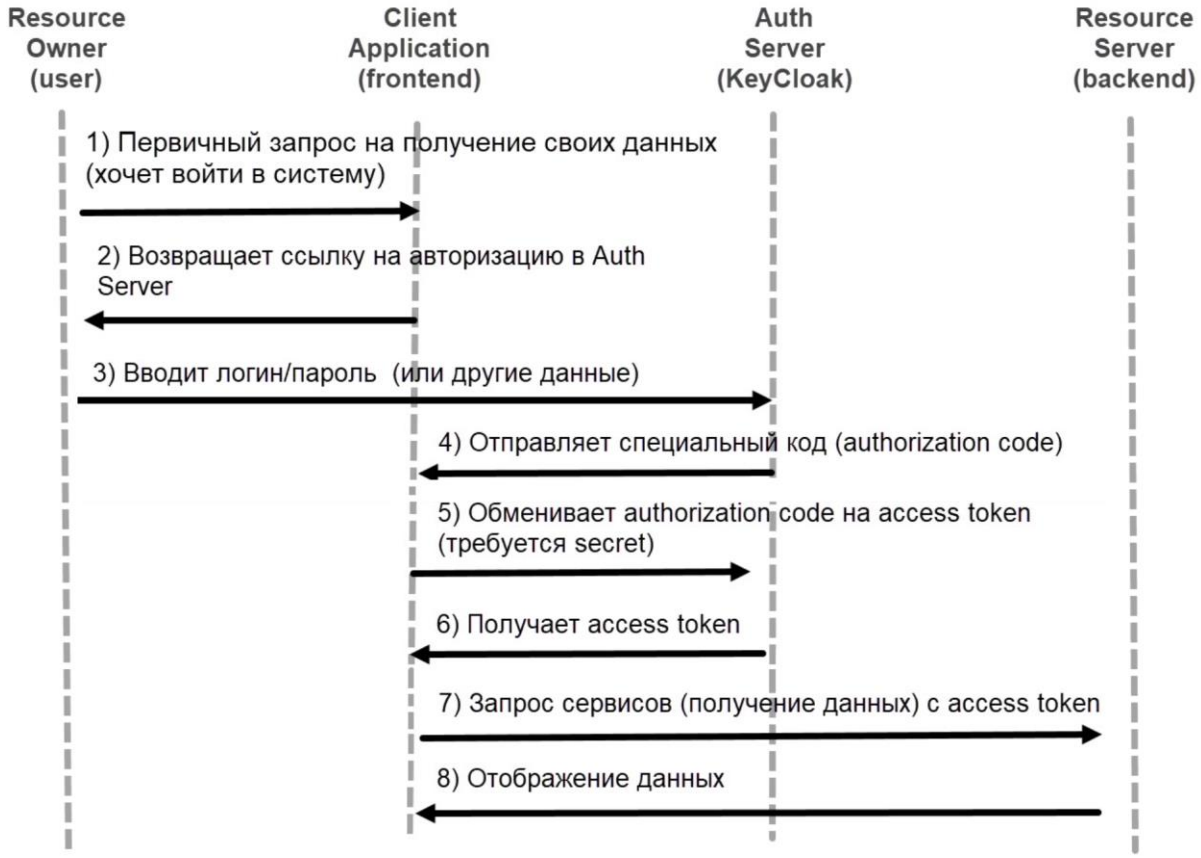


Рисунок 11 – Схема работы Authorization Code Flow

2.4 Angular

Фреймворк, разработанный и поддерживаемый Google, для создания веб-приложений с использованием TypeScript. Angular предоставляет множество инструментов для разработки, таких как компоненты, модули, сервисы, маршрутизация и многое другое. Angular также предлагает эффективную систему управления данными и реактивное программирование [11].

2.5 Ionic

Фреймворк для разработки гибридных мобильных приложений с использованием веб-технологий, таких как HTML, CSS и JavaScript/TypeScript [12]. Он предоставляет большой набор UI компонентов, которые позволяют создавать приложения с нативным для мобильных устройств внешним видом и ощущением. Ionic также обеспечивает интеграцию с Angular, что делает его

отличным выбором для разработки кроссплатформенных приложений с использованием Angular.

3 ГЛАВА

3.1 Пользовательский сценарий

В рамках разработки данного приложения предусмотрены различные роли пользователей, включающие клиентов-заказчиков, клиентов-исполнителей и администраторов. Каждая из этих ролей имеет свой собственный пользовательский сценарий, который для наглядности приведен на графике в приложении А.

3.1.1 Клиент-заказчик

– Авторизация;

– Выбор кафе, откуда планируется заказ;

– В появившемся меню кафе выбор понравившегося блюда;

– Просмотр карточки блюда с полной информацией о нем (вес, калораж, состав);

– Добавление блюдо в корзину;

– Выбор способа получения:

а) Самовывоз:

1) Получение qr-кода для того, чтобы забрать заказ из кафе;

б) Доставка:

1) Заказ публикуется на портале доставки;

2) Исполнитель принимает заказ;

3) Исполнитель выполнил заказ;

4) Подтверждение доставки заказчиком.

3.1.2 Клиент-исполнитель

– Авторизация;

– Переход на портал доставки;

– Принятие заказа на доставку;

– Получение заказа по qr-коду из кафе;

– Доставка заказа заказчику;

– Получение бонусного баланса.

3.1.3 Администратор кафе

– Отображение заказа на панели заказов;

– Сборка заказа;

– Передача заказу клиенту (заказчику или исполнителю в зависимости от вида получения);

3.2 Хранение данных

Данные, которыми оперирует данный проект можно условно разделить на три части – требующие долгосрочного, кратковременного (около 3 часов) и среднего по продолжительности хранения (7 дней).

3.2.1 Долгосрочное хранение

К данным, требующим долгосрочного хранения, можно отнести информацию о кафе, блюдах и пользовательскую информацию
Такие данные хранятся в PostgreSQL (рисунок 12).

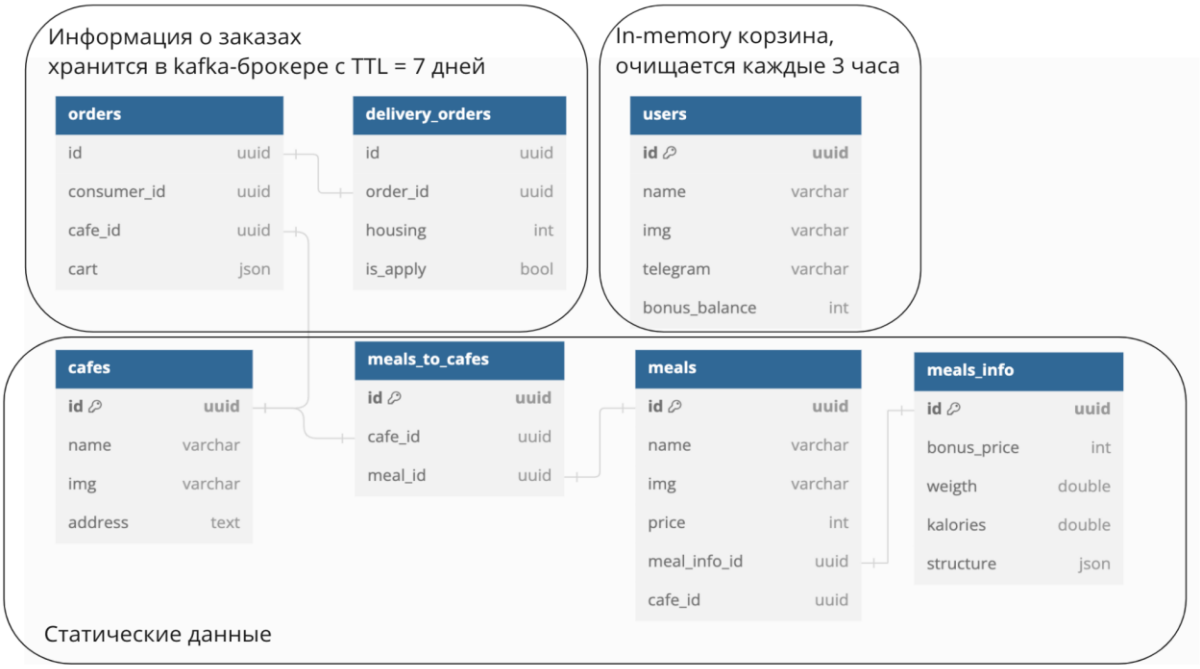


Рисунок 12 – Схема хранимых данных (как в PostgreSQL, так и в Apache Kafka)

Подробнее про поля таблиц:

1. Таблица: meals

– id (uuid): Первичный ключ для таблицы meals, уникально идентифицирует каждый прием пищи;

– name (varchar): Название приема пищи;

– img (varchar): URL или путь к изображению, представляющему прием пищи;

– price (int): Цена приема пищи;

– meal_info_id (uuid): Внешний ключ, ссылающийся на поле id в таблице meals_info, связывающий дополнительную информацию о приеме пищи;

– cafe_id (uuid): Внешний ключ, ссылающийся на поле id в таблице cafes, указывает на кафе, где доступен прием пищи.

2. Таблица: meals_info

– id (uuid): Первичный ключ для таблицы meals_info, уникально идентифицирует каждый набор дополнительной информации о приеме пищи;

– bonus_price (int): Бонусная цена за прием пищи, если применимо.

– weight (double): Вес приема пищи;

– calories (double): Калорийность приема пищи;

– structure (json): Поле JSON, хранящее структуру или ингредиенты приема пищи.

3. Таблица: users

– id (uuid): Первичный ключ для таблицы users, уникально идентифицирует каждого пользователя;

– name (varchar): Имя пользователя;

– img (varchar): URL или путь к изображению, представляющему пользователя;

– bonus_balance (int): Бонусный баланс, связанный с пользователем.

4. Таблица: meals_to_cafes

– id (uuid): Первичный ключ для таблицы meals_to_cafes, уникально идентифицирует каждую запись;

– cafe_id (uuid): Внешний ключ, ссылающийся на поле id в таблице cafes, указывает на кафе, где доступен прием пищи;

– meal_id (uuid): Внешний ключ, ссылающийся на поле id в таблице meals, указывает на прием пищи, доступный в кафе.

5. Таблица: cafes

– id (uuid): Первичный ключ для таблицы cafes, уникально идентифицирует каждое кафе;

– name (varchar): Название кафе;

– img (varchar): URL или путь к изображению, представляющему кафе;

– address (text): Адрес кафе.

3.2.2 Кратковременное хранение

Данные, которые необходимо очищать чаще всего – это информация о пользовательской корзине, за их очистку отвечает scheduler, она происходит раз в 3 часа.

Scheduler является компонентом из библиотеки *java.util.concurrent*, который ответственен за планирование и выполнение ряда задач в определенное время или через определенные промежутки времени в соответствии с заданным расписанием. Пример кода приведен на рисунке 13.

Этот подход обеспечивает систематическое освобождение ресурсов и поддержание корректного состояния приложения, предотвращая накопление устаревших или неиспользуемых данных в корзине, что может привести к нежелательным последствиям, таким как перегрузка памяти или неправильное отображение информации для пользователей.

```
1 usage
private void scheduleCartClearing() {
    scheduler.scheduleAtFixedRate(this::clearCart, initialDelay: 0, period: 3, TimeUnit.HOURS);
}
```

Рисунок 13 – Реализация работы sheduler

3.2.3 Хранение средней продолжительности

К этой категории относится информация о заказах, поступивших от пользователей, обеспечивает это хранение Apache Kafka.

В брокере хранятся объекты следующих видов:

1. Топик: orders

– id (uuid): Уникальный идентификатор для заказа;

– consumer_id (uuid): Внешний ключ, ссылающийся на поле id в таблице users, указывает на пользователя, сделавшего заказ;

– cafe_id (uuid): Внешний ключ, ссылающийся на поле id в таблице cafes, указывает на кафе, откуда был сделан заказ;

– cart (json): Поле JSON, хранящее детали заказанных позиций.

2. Топик: delivery_orders

– id (uuid): Уникальный идентификатор для заказа с доставкой;

– order_id (uuid): Внешний ключ, ссылающийся на поле id в таблице orders, указывает на заказ, связанный с доставкой;

– housing (int): Номер квартиры или дома для доставки;

– is_apply (bool): указывает, был ли применен заказ с доставкой или нет.

3.3 Работа с брокером сообщений

Графически flow работы с брокером сообщений представлен в приложении Б.

3.3.1 Почему выбор пал на Kafka?

В данном проекте для пересылки информации о заказе между микросервисами используется Apache Kafka.

Kafka обеспечивает масштабируемость и высокую производительность, что особенно важно при увеличении количества клиентов. Это связано с тем, что в нем часть работы выполняют клиенты, беря на себя часть нагрузки с

брокера сообщений. Таким образом, при росте числа клиентов нагрузка на брокер остается относительно стабильной, что способствует эффективной работе системы.

Во-вторых, использование Apache Kafka позволяет совместить функции брокера сообщений и базы данных, хранящей данные, не требующие долговременного хранения. Время жизни (TTL) данных в базе данных установлено на 7 дней, что позволяет эффективно управлять ресурсами и избегать накопления устаревших данных, сохраняя при этом необходимую информацию для работы системы.

Также Kafka обладает гибкой архитектурой и широким набором инструментов для обработки потоков данных, что делает его предпочтительным выбором для проектов с высокими требованиями к обработке и передаче данных.

3.3.2 Описание топиков

На схеме указан flow работы Kafka брокера, в нем создано четыре топика, в которые записываются сообщения:

– Все поступающие заказы;

а) После создания заказа клиентом-заказчиком, заказ попадает в этот топик;

б) Сервисы администратора и портала доставки считывают информацию из этого топика;

– Запросы на доставку от сервиса доставки;

а) После принятия клиентом-исполнителем запроса на доставку, информация об этом попадает в сервис администратора кафе;

– Ответы от сервиса администратора кафе, что заказ принят;

а) В случае самовывоза;

Как только сервис администратора кафе получил заказ с самовывозом, он отправляет клиенту uuid заказа и время timestamp ответа;

б) В случае доставки;

Как только сервис администратора кафе получил информацию о том, что клиент-исполнитель принимает заказ, он отправляет клиенту uuid заказа и время timestamp ответа;

– Подтверждение доставки;

а) После подтверждения клиентом-заказчиком завершения доставки, доставка считается завершенной и начинается начисление бонусного баланса на счет клиента-исполнителя.

3.4 Безопасность

В современной разработке веб-приложений все чаще встает вопрос обеспечения безопасности данных и аутентификации пользователей. Одним из методов защиты конфиденциальности и безопасности информации является управление доступом через токены. В данном контексте реализация технологии "backend for frontend" (BFF) приобретает значимость, поскольку она позволяет эффективно управлять доступом и безопасностью, минимизируя риски утечки конфиденциальной информации [13]. В частности, BFF может быть использован для того, чтобы избежать хранения access token'ов непосредственно в браузере, что повышает уровень безопасности веб-приложения. Вместо этого токены на стороне frontend’а хранятся в виде безопасных cookie. Схема реализации процесса авторизации представлена в приложении В.

3.4.1 Безопасные cookie

Куки являются небольшими фрагментами данных, которые веб-сервер отправляет браузеру, а браузер сохраняет их и возвращает обратно при каждом запросе к тому же серверу.

Основные различия между обычными cookie и безопасными cookie (cookie, с включенными флагами secure = true и httpOnly = true):

3.4.1.1 HttpOnly Cookie:

Куки с флагом httpOnly = true доступны только для сервера и не могут быть прочитаны или изменены JavaScript'ом в браузере. Это помогает предотвратить различные типы атак, такие как XSS (межсайтовый скриптинг),

когда злоумышленник может попытаться украсть куки через выполнение вредоносного JavaScript на веб-странице.

3.4.1.2 Secure Cookie:

Куки, у которых установлен флаг secure = true, могут быть переданы только через защищенное (HTTPS) соединение. Это помогает защитить cookie от перехвата злоумышленниками в открытом виде при передаче данных между клиентом и сервером.

Безопасные куки не отправляются по незашифрованным каналам связи, что повышает уровень безопасности передаваемых данных.

3.4.1.2.1 SSL

Для реализации этого способа защитить cookie было необходимо получить для рассматриваемого сервиса ssl-сертификат и ssl-ключ, я сделал это используя OpenSSL.

Для самоподписания SSL-сертификата и последующего добавления его в браузер для использования веб-приложением без необходимости верификации со стороны центра сертификации, выполняются следующие действия:

1. Генерация самоподписанного SSL-сертификата с помощью утилиты OpenSSL, указав параметры, такие как срок действия сертификата и общее имя (Common Name) для сервера;
2. В разделе сертификатов браузера произведено добавление самоподписанного сертификата как доверенного.

После добавления сертификата в список доверенных, браузер начал использовать его для установки безопасного соединения с веб-приложением

2.4.2 Процесс авторизации

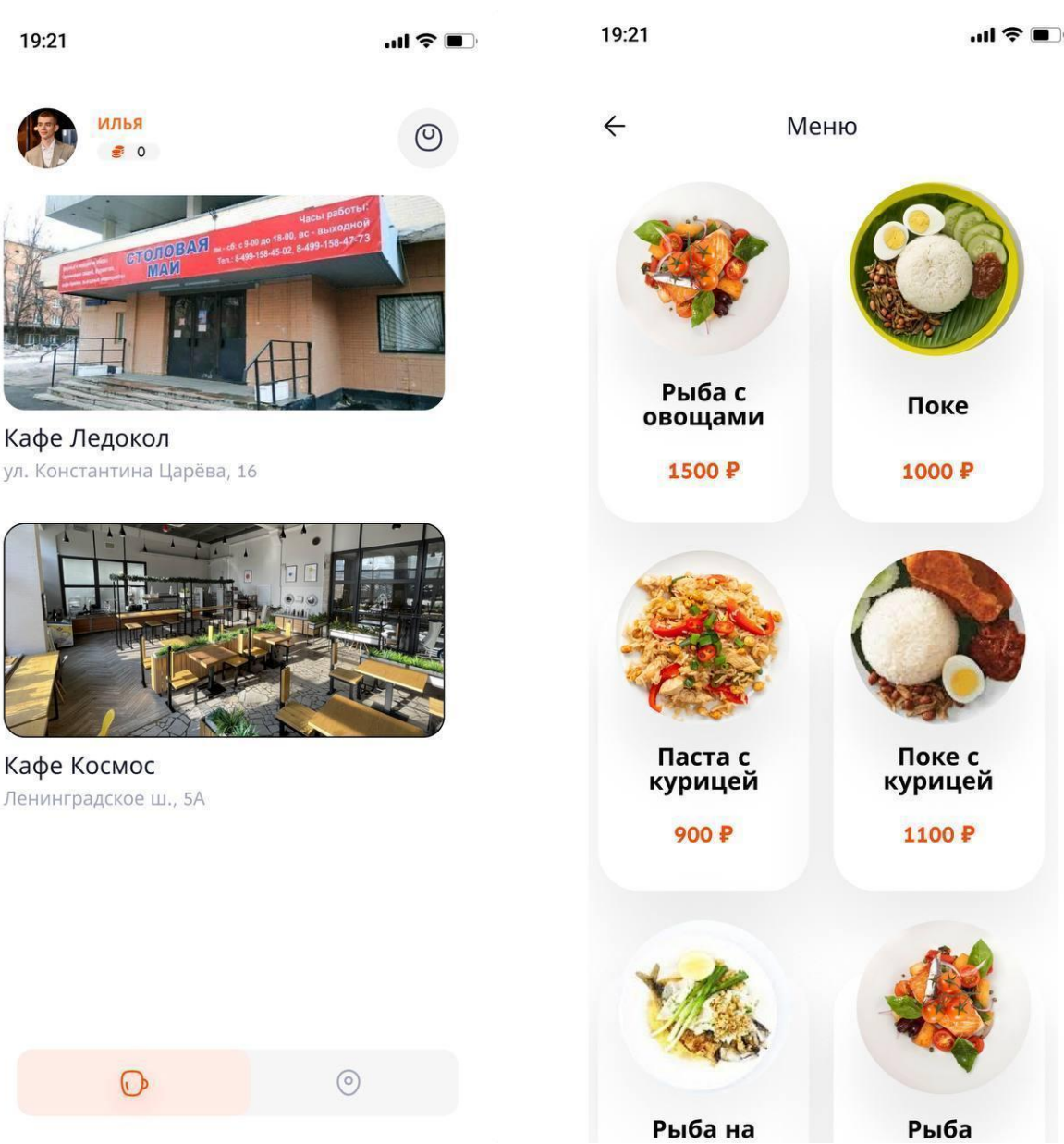
1. При переходе на порт с работающим api-gateway идет переадресация на порт keycloack-server для авторизации;
2. Авторизация (или регистрация, если учетной записи этого пользователя до этого не было);

3. Получение Authentification code;
4. Передача этого токена на BFF сервис, чтобы он обменял его у authorization server на ID token, Access token и Refresh token;
5. Составление ответа на фронтенд с информацией про эти токены, которые хранятся в защищенных cookie;
6. Сохранение этой информации в браузере, для последующего использования этих cookie при работе с этим доменом (в данном случае с комбинацией ip-адрес и порт);
7. Отправка бизнес-запроса, к которому уже добавлены данные cookie;
8. Парсинг cookie на bff сервисе и отправка конечного запроса на resource-server;
9. Перед выполнением бизнес-логики идет проверка у authorization server’a выданных им токенов на валидность;
10. Если токен валидный и роль пользователя допускает получение информации по его запросу, происходит работа бизнес-логики приложения и возвращает ответ на frontend.

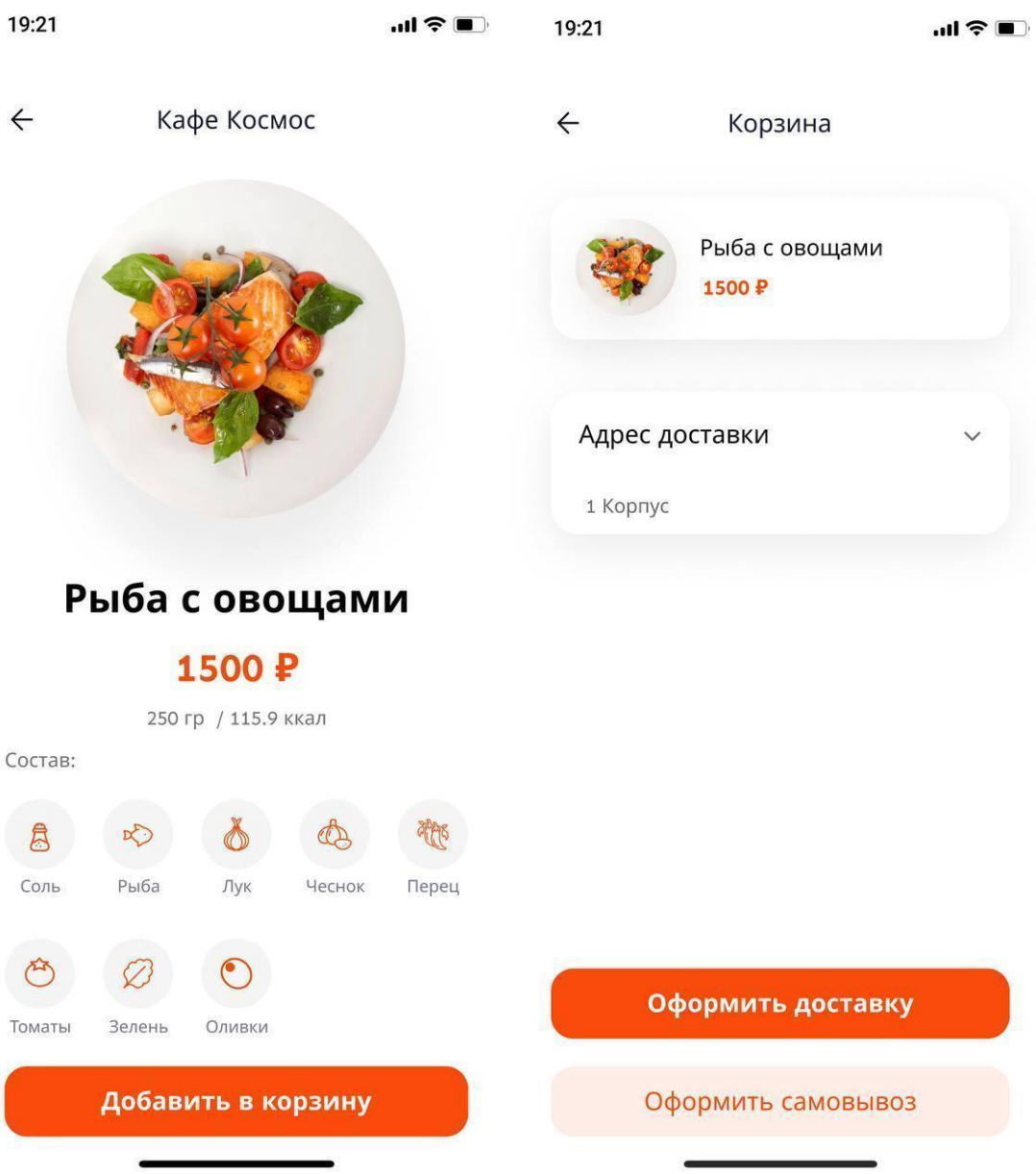
3.5 UI

Рассмотрение UI приложения будет рассмотрено исходя из user flow для каждой роли из трех клиент-заказчик, клиент-исполнитель и администратор.

3.5.1 Клиент-заказчик



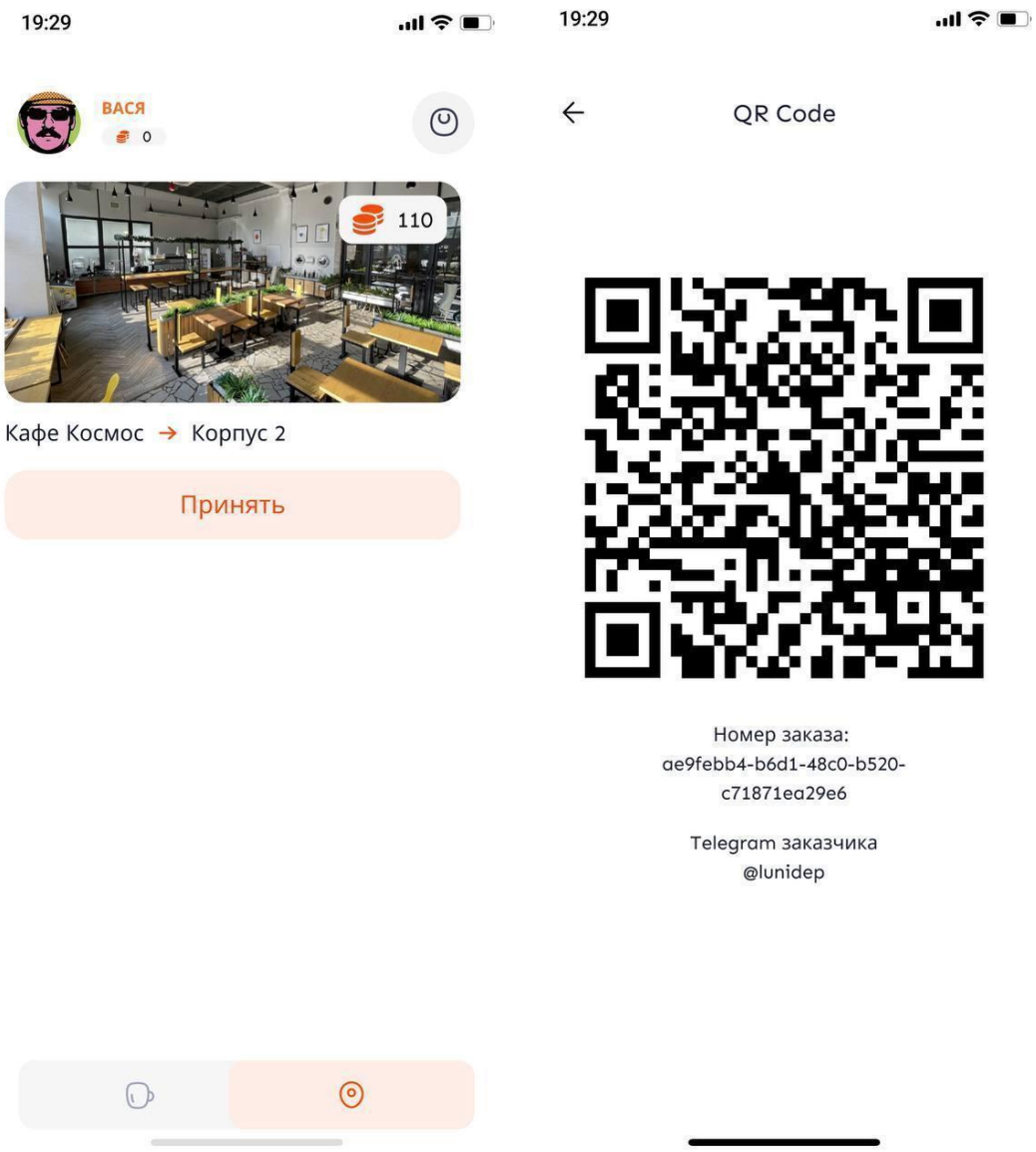
Рисунки 14, 15 – Экраны выбора кафе и меню выбранного кафе



Рисунки 16, 17 – Экраны карточки блюда и корзины

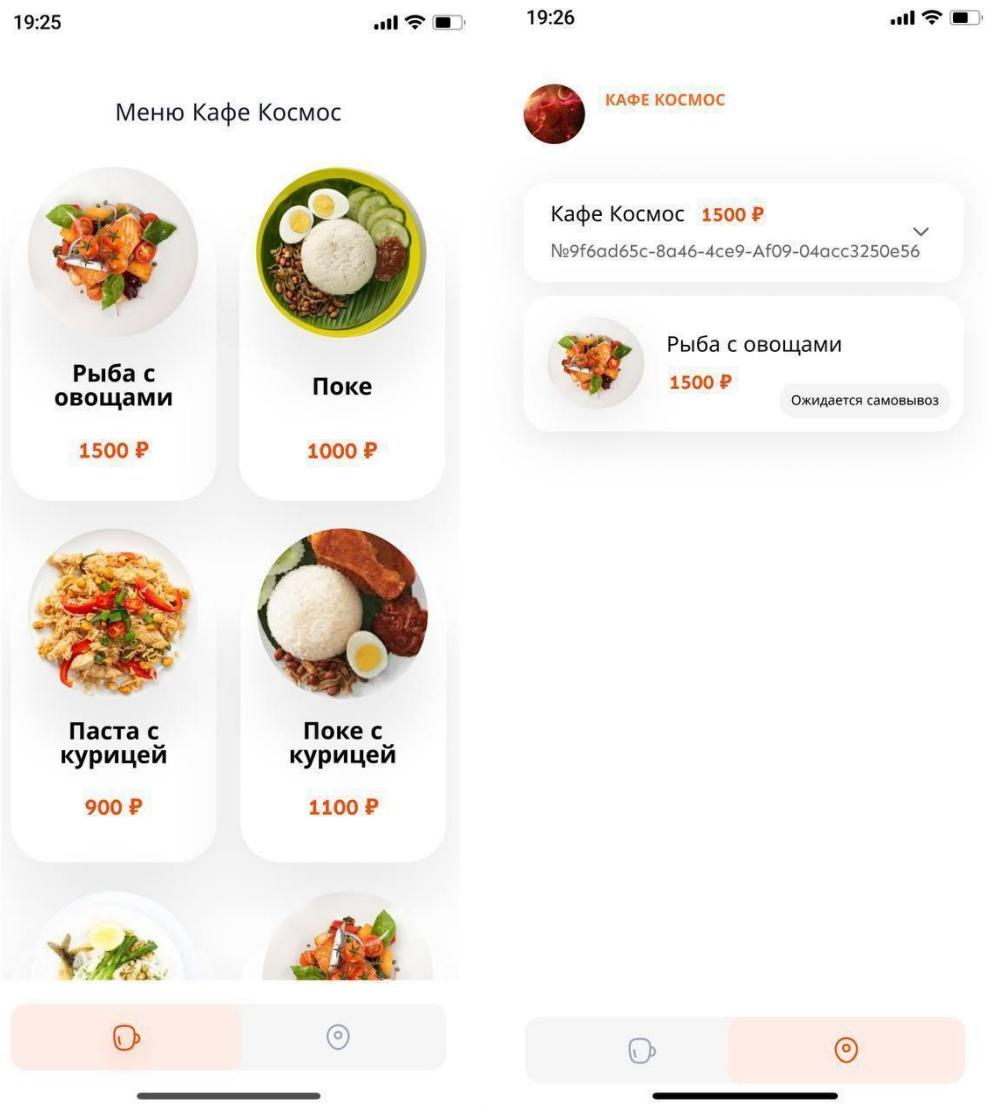
Рисунок 18 – Экран qr-кода для получения заказа самовывозом

3.5.2 Клиент-исполнитель



Рисунки 19, 20 – Экраны портала доставки и qr-кода для получения заказа клиентом-исполнителем

3.5.3 Администратор кафе



Рисунки 21, 22 – Экраны меню кафе и списка заказов от аккаунта администратора

3.6 Перспективы развития

Среди перспектив развития приложения, рассматривается использование кэширования данных с помощью технологий Redis или MongoDB, репликацию баз данных для обеспечения надежности, а также оркестрацию на сервере с помощью Kubernetes для эффективного управления инфраструктурой и масштабирования приложений.

ЗАКЛЮЧЕНИЕ

Разработанное приложение стало значимым шагом в решении проблемы, связанной с ограниченным доступом студентов к обеденному перерыву из-за долгих очередей. Благодаря его функционалу, студенты получат возможность заказывать блюда из столовых удобным и эффективным способом, что позволит им не только сэкономить время, но и насладиться полноценным обедом, минимизируя стресс и негативное воздействие на их здоровье.

Внедрение микросервисной архитектуры, использование надежных механизмов безопасности и защиты данных обеспечивают стабильную и устойчивую работу приложения, гарантируя высокий уровень защиты от внешних угроз и вмешательств.

Приложение, которое получилось в результате разработки не только повысит качество жизни пользователей, но и позволит им иметь дополнительный источник дохода, что в студенческой среде никогда не бывает лишним.

СПИСОК ЛИТЕРАТУРЫ

1. REST // wikipedia URL: https://ru.wikipedia.org/wiki/REST (дата обращения: 20.04.2024).
2. Микросервисная архитектура // wikipedia URL: https://ru.wikipedia.org/wiki/Микросервисная_архитектура (дата обращения: 20.04.2024).
3. Балансировка нагрузки // samwho URL: https://samwho.dev/load-balancing/ (дата обращения: 20.04.2024).
4. Сервер авторизации // keycloak URL: https://www.keycloak.org/docs/latest/release_notes/index.html (дата обращения: 20.04.2024).
5. Статистика использования браузеров и мобильных устройств // vc.ru URL: https://vc.ru/u/1274559-l-tech/674848-statistika-ispolzovaniya-brauzerov-i-mobilnyh-ustroistv (дата обращения: 20.04.2024).
6. Spring in Action, Fourth Edition: Covers Spring 4 / C. Walls. – Manning Publications, 2015. – 624 с. – URL: [https://github.com/cuocsongquanhta/Books-1/blob/master/Spring%20in%20Action%2C%204th%20Edition.pdf] (дата обращения: 03,11,.2023). – Режим доступа: свободный.
7. Introduction to Spring Cloud Netflix – Eureka // spring-cloud-netflix-eureka URL: https://www.baeldung.com/spring-cloud-netflix-eureka (дата обращения: 20.04.2024).
8. Exploring the New Spring Cloud Gateway // spring-cloud-gateway URL: https://www.baeldung.com/spring-cloud-gateway (дата обращения: 20.04.2024).
9. Kafka // kafka URL: https://kafka.apache.org/ (дата обращения: 20.04.2024).
10. Keycloak // wikipedia URL: https://ru.wikipedia.org/wiki/Keycloak (дата обращения: 20.04.2024).
11. Angular // angular.io URL: https://angular.io/ (дата обращения: 20.04.2024).

12. Testing ng-add in ionic // npmjs.com URL: https://www.npmjs.com/package/@ionic/angular (дата обращения: 20.04.2024).
13. Backend-for-Frontend: когда простого API не хватает // habr.com URL: https://habr.com/ru/articles/557406/ (дата обращения: 20.04.2024).

ПРИЛОЖЕНИЕ А

User flow

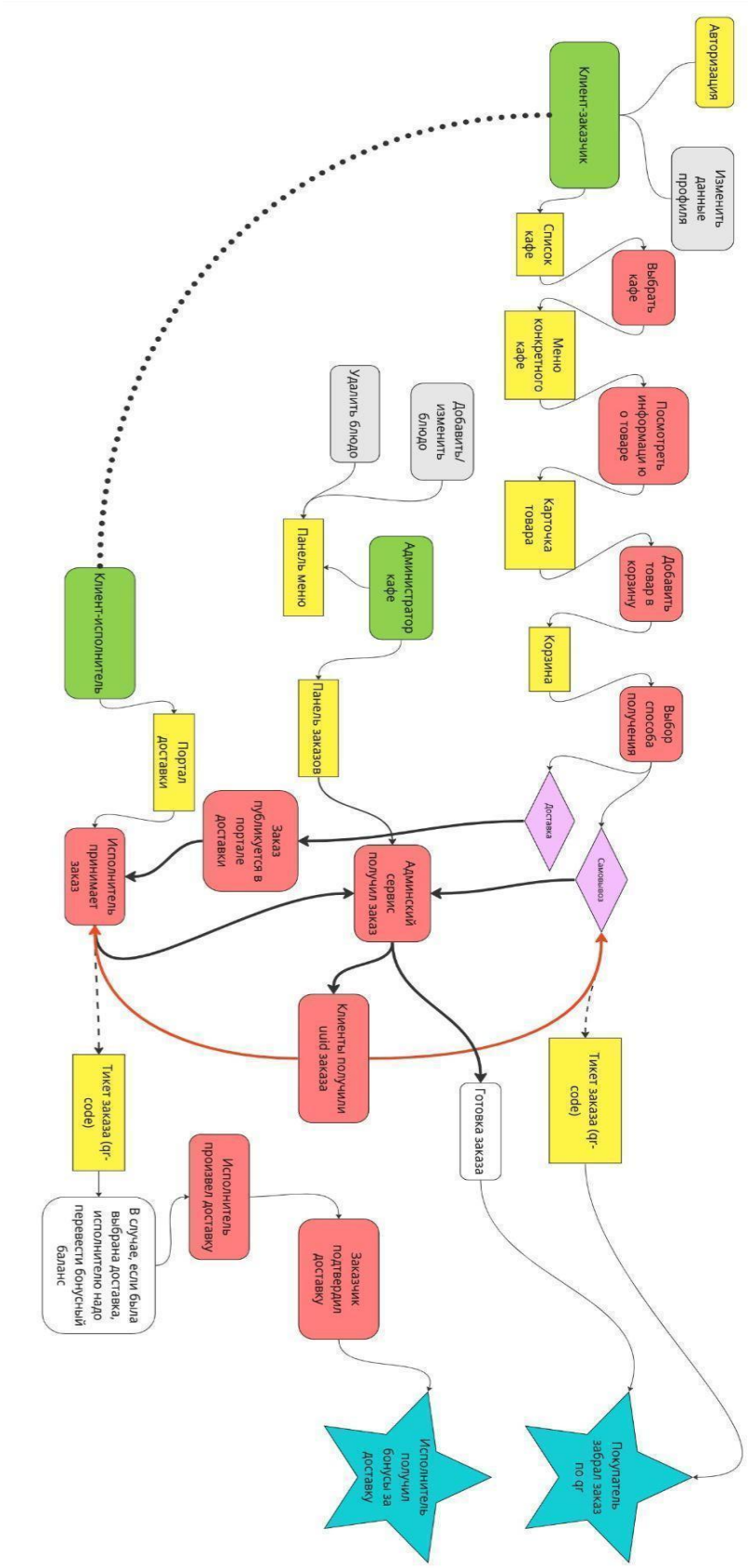


Рисунок А.1 – Пользовательский сценарий

ПРИЛОЖЕНИЕ Б

Flow брокера сообщений

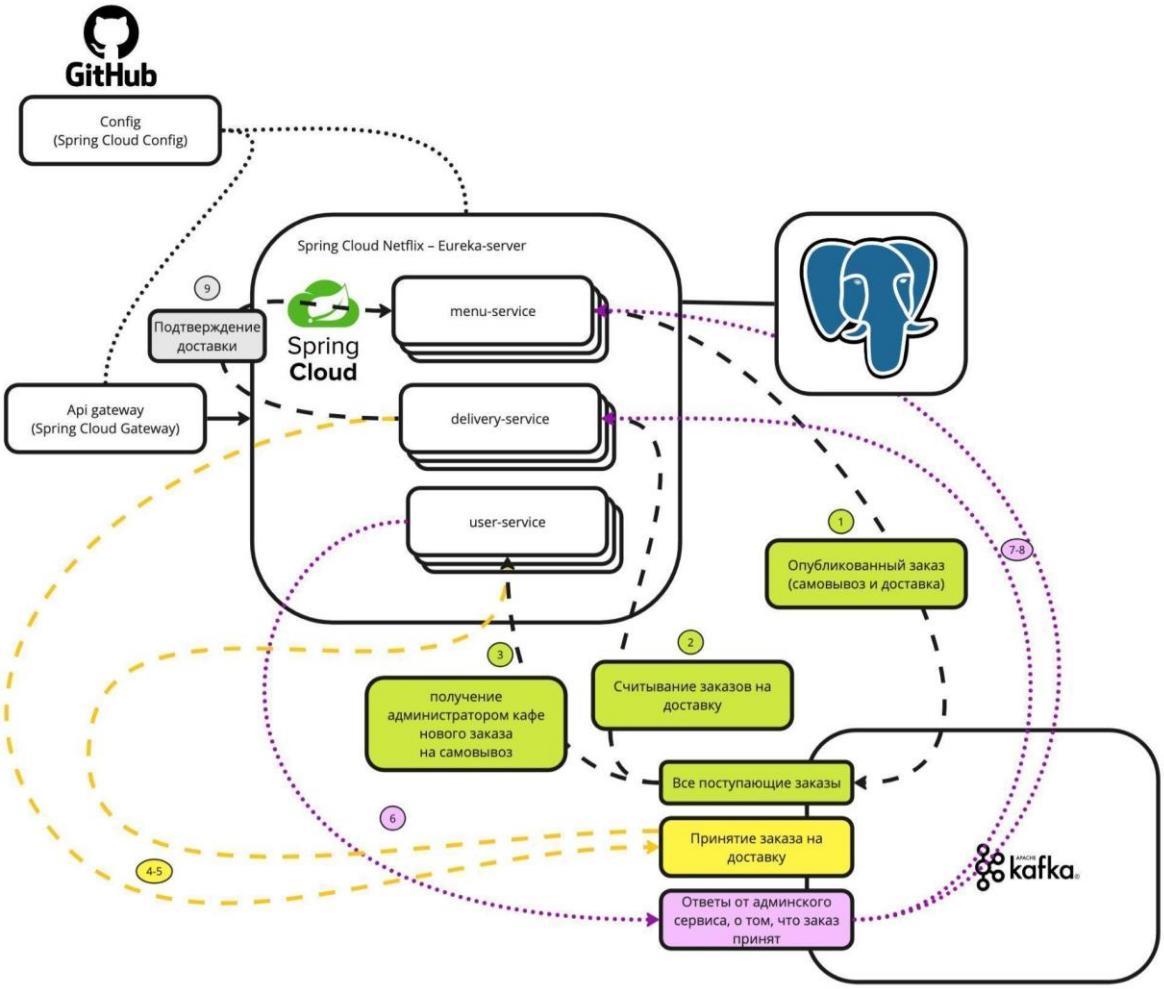


Рисунок Б.1 – Flow брокера сообщений

ПРИЛОЖЕНИЕ В

Flow работы bff, resource server и authorization server

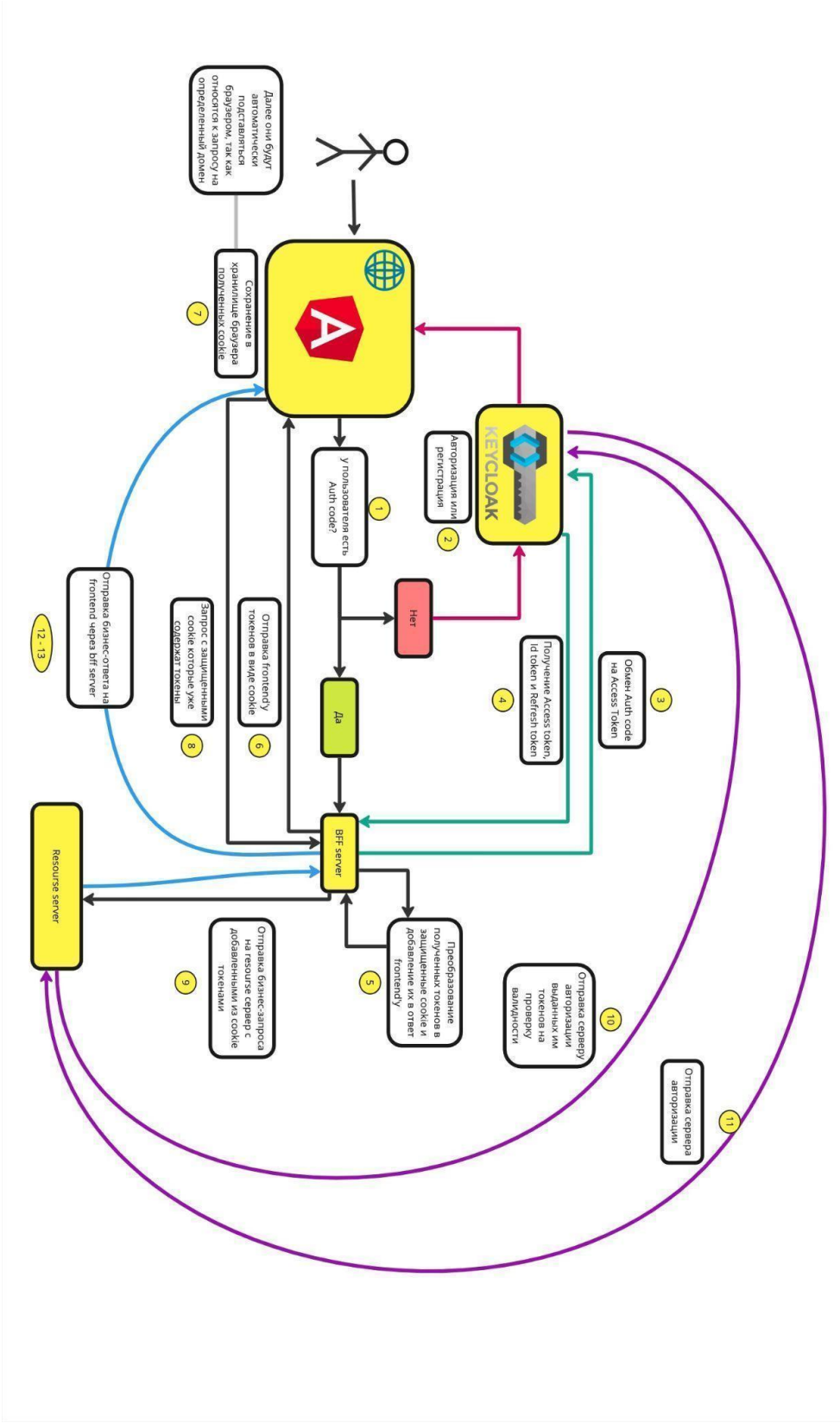


Рисунок В.1 – Flow работы bff, resource server и authorization server

ПРИЛОЖЕНИЕ Г

Исходный код

Рисунок Д.1 – QR-код на репозиторий работы в GitHub