МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра Прикладная математика и информатика
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки)

Управление корпоративными информационными процессами
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему «Разработка микросервисной архитектуры для web-приложения»

Обучающийся Е.И. Чернякова
(Инициалы Фамилия) (личная подпись)

Научный руководитель к.т.н., доцент, О.В. Аникина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2023

Оглавление

Введение

Разработка веб-приложений — это быстроразвивающаяся область, требующая новых подходов к проектированию архитектуры системы. Монолитная архитектура широко использовалась в разработке программного обеспечения в течение многих лет, но она имеет ряд ограничений, включая трудности с масштабированием, поддержкой и обновлением приложения. В результате разработчики изучают альтернативные архитектурные подходы, такие как микросервисные и сервис-ориентированные архитектуры.

Актуальность темы исследования обусловлена тем, что реализация микросервисной архитектуры обеспечивает такие преимущества, как масштабируемость, поддерживаемость и гибкость. Кроме того, анализ предметной среды даст представление о различных архитектурных подходах, что позволит разработчикам выбрать оптимальный подход для своих конкретных нужд. Реализация микросервисной архитектуры обеспечивает практический опыт проектирования и разработки сложной распределенной системы.

Объектом исследования в данной магистерской диссертации является архитектура микросервисных веб-приложений.

Предметом исследования в данной диссертации является разработка микросервисной архитектуры для монолитного веб-приложения, которая устраняет ограничения монолитной архитектуры.

Цель данного исследования — внести вклад в разработку инновационных и эффективных решений для устранения ограничений монолитной архитектуры при создании веб-приложений. Реализуя микросервисную архитектуру, данное исследование призвано обеспечить комплексное понимание микросервисной архитектуры и ее применения в имплементации сложных веб-приложений, которые являются масштабируемыми, обслуживаемыми и гибкими.

Гипотеза исследования данной магистерской диссертации заключается в том, что внедрение микросервисной архитектуры для монолитного веб-приложения улучшает масштабируемость, поддерживаемость и гибкость приложения. Ожидается, что благодаря декомпозиции монолитного приложения на более мелкие, независимые сервисы, которые можно разрабатывать, развертывать и поддерживать независимо друг от друга, микросервисная архитектура позволит ускорить циклы разработки и развертывания, сократить время простоя и повысить общую производительность приложения.

Для достижения и проверки гипотезы будут решены следующие задачи:

– проанализирована предметная среда, включая современные тенденции и практику разработки программного обеспечения и ограничения монолитной архитектуры;
– спроектирована архитектура системы, предварительно изучив различные архитектурные подходы и выбрав наиболее подходящий подход для конкретных нужд приложения;
– реализована микросервисная архитектура путем декомпозиции монолитного приложения на более мелкие, независимые сервисы, которые можно разрабатывать, развертывать и поддерживать независимо друг от друга;
– оценена эффективность микросервисной архитектуры с точки зрения масштабируемости, поддерживаемости и гибкости;
– разработан бизнес-план проекта, анализирующий рыночные возможности разработанной системы, позволяющий определить потенциальную окупаемость инвестиций.

Методы исследования: теоретический анализ, системный анализ, анализ архитектурных подходов в разработке программного обеспечения, современные подходы к сопровождению и модернизации программного обеспечения, сравнительный анализ шаблонов микросервисной архитектуры.

Основные этапы исследования: исследование состояло из нескольких этапов и проводилось с 2021 по 2023 годы:

Первый этап включал в себя определение темы исследования, сбор и тщательное изучение соответствующих данных по теме исследования. Кроме того, на этом этапе были определены цель, задачи, предмет, объект и гипотеза исследования.

На втором этапе было проведено исследование и разработка архитектуры программного обеспечения для поиска мероприятий, было разработано программное обеспечение для web-сервиса, был проведен анализ эффективности и тестирование разработанного решения, проведен обзор результатов.

На третьем этапе была проведена разработка бизнес-плана проекта, выполнен анализ технологического аудита проекта, был проведен анализ возможностей выхода на рынок, рыночной стратегии и маркетинговой программы. Этот шаг предполагает создание плана вывода на рынок разработанной системы микросервисной архитектуры. Основная цель этого этапа — оценить жизнеспособность проекта, его потенциальный рыночный спрос и финансовую осуществимость.

Результаты работы обладают научной новизной в области проектирования и разработки архитектуры приложений, так как практически показывают плюсы микросервисной архитектуры перед монолитным веб-приложением и проводится оценка эффективности, производительности, масштабируемости и гибкости двух систем в тестовых условиях.

Теоретическая значимость данной магистерской диссертации заключается в предоставлении всестороннего анализа проблемы разработки программного обеспечения и потенциального решения, предлагаемого микросервисной архитектурой. В исследовании рассматриваются монолитная архитектура, сервис-ориентированная архитектура (SOA) и микросервисная архитектура (MSA), а также дается обзор их сильных и слабых сторон. Кроме

того, в диссертации представлен подробный анализ микросервисной архитектуры, включая ее общее описание, шаблоны и схему прецедентов.

Практическая значимость данного исследования важна для организаций и разработчиков, заинтересованных во внедрении архитектуры микросервисов в свои системы. Предложенный подход предлагает практическую основу для реализации микросервисов, которая может быть легко адаптирована к различным технологическим стекам и доменам. Кроме того, исследование содержит рекомендации по эффективному внедрению и управлению архитектурой микросервисов, которые могут помочь организациям избежать распространенных подводных камней и улучшить практику разработки программного обеспечения.

Методологическую базу исследования составляют различные источники, включая литературу и интернет-ресурсы. Использованные источники охватывают различные аспекты архитектуры программного обеспечения, микросервисов и связанных с ними технологий. К ним относятся работы авторов: Р. Мачадо, Т. Эрла, М. Фаулера, Г. Шилдта, Р. Баратана и др. Также была использована официальная документация проектов Netflix OSS, Nodejs, Express API, Spring Cloud/Data JPA/Cloud Netflix, JPA, учебники по Java от Oracle.

По теме исследования опубликована 1 статья: Чернякова Е.И. «Исследование микросервисной архитектуры для веб-приложения», результаты исследования были представлены в научном журнале «Вестник магистратуры». 2022. №12 (135) (ISSN 2223-4047).

На защиту выносятся:

– реализация микросервисной архитектуры путем декомпозиции монолитного приложения на более мелкие, независимые сервисы;
– оценка эффективности микросервисной архитектуры с точки зрения масштабируемости, поддерживаемости и гибкости;
– разработка бизнес-плана проекта, анализирующий рыночные возможности разработанной системы, позволяющий

предпринимателям определить потенциальную окупаемость инвестиций.

Диссертация состоит из введения, четырех глав, заключения и списка используемой литературы.

Глава 1 представляет обзор архитектурных подходов в разработке программного обеспечения, а именно монолитной архитектуры, сервис-ориентированной архитектуры и микросервисной архитектуры. Эта глава включает подробное обсуждение каждого архитектурного подхода, включая его преимущества и недостатки.

Глава 2 содержит подробное описание подхода микросервисной архитектуры. Эта глава включает обсуждение различных шаблонов микросервисной архитектуры, таких как «Decompose by business capability», «Database per service», «API Gateway» и др. Кроме того, в этой главе представлена общая схема прецедентов и обзор языков программирования и веб-фреймворков.

Глава 3 включает в себя подробное обсуждение реализации микросервисов для проектируемой системы. Глава включает конкретные сервисы, такие как сервис шлюза API, сервер UAA, сервер реестра и сервис событий. В главе также приводится общее описание системы и анализ разработанного решения, был проведен практический анализ эффективности и тестирование разработанного решения, проведен обзор результатов.

Глава 4 включает анализ технологического аудита проекта, возможностей выхода на рынок, рыночной стратегии и маркетинговой программы. Глава также включает описание идеи проекта и завершается общим выводом по разделу.

В заключении приводятся результаты исследования.

Работа изложена на 121 странице, включает 48 рисунков, 28 таблиц, 32 источника используемой литературы.

Глава 1 Анализ предметной среды

1.1 Анализ проблем при разработке программного обеспечения

Проблема выбора и проектирования оптимальной архитектуры для веб-приложений является предметом исследования в научной литературе на протяжении многих лет. Традиционно, многие годы в подавляющем числе случаев использовались монолитные веб-приложения, которые имеют единую, большую кодовую базу, которая обрабатывает все функциональные возможности приложения. Хотя такая архитектура проста и удобна в разработке, она имеет ряд ограничений, таких как масштабируемость, устойчивость и ремонтопригодность [11].

Монолитные архитектуры проектируются как единое целое, в котором все компоненты тесно интегрированы и зависят друг от друга. Этот подход хорошо работал в течение многих лет, когда разработка программного обеспечения была менее сложной и имела меньшее количество компонентов [14]. Однако по мере того, как приложения становились сложнее и содержали больше компонентов, ограничения монолитной архитектуры становились все более очевидными.

Одной из основных проблем монолитной архитектуры является отсутствие модульности и гибкости, что описано в исследовании [1]. В монолитной архитектуре все компоненты приложения тесно связаны друг с другом, что затрудняет изменение или добавление новых функций без влияния на всю систему. Это затрудняет обслуживание и масштабирование приложения с течением времени, особенно по мере роста его размера и сложности. Более того, из-за тесно связанной природы архитектуры приложение становится менее гибким и его сложнее адаптировать к изменениям в бизнес-требованиях или новым технологиям [15].

Еще одной проблемой монолитных архитектур является высокий уровень эксплуатационных расходов, связанных с ними. Монолитные

приложения обычно требуют больших и дорогих аппаратных ресурсов, таких как серверы и базы данных, которые сложно поддерживать и масштабировать [22]. Обслуживание и эксплуатационные расходы, связанные с монолитной архитектурой, могут стать существенным барьером для малого или среднего бизнеса, поскольку у них может не быть ресурсов для инвестиций в крупные аппаратные инфраструктуры, что было описано в исследовании [2].

Классическая монолитная архитектура имеет ряд ограничений, которые делают ее менее подходящей для современных веб-приложений. Отсутствие модульности и гибкости, сложность в обслуживании и масштабировании приложения, а также высокие эксплуатационные расходы, связанные с ней, — вот некоторые из основных проблем. Поэтому новые архитектурные модели, такие как SOA и MSA, появились как потенциальные решения для преодоления этих ограничений и обеспечения более гибкого и масштабируемого подхода к разработке программного обеспечения.

Для преодоления этих ограничений применяются новые современные распределенные архитектуры: сервис-ориентированная архитектура и микросервисная архитектура, которые обеспечивают ряд преимуществ, таких как масштабируемость, отказоустойчивость и модульность.

Переход от монолитной архитектуры к архитектуре SOA требует тщательного планирования и исполнения. Одной из проблем при переходе к SOA является проектирование самих сервисов. Сервисы должны иметь четкий интерфейс и функции, а также должны быть слабо связаны друг с другом, чтобы обеспечить возможность их изменения и масштабирования независимо друг от друга. Кроме того, каждая служба должна иметь собственное хранилище данных, что может создать проблемы с согласованностью и синхронизацией данных между службами [19].

Еще одной проблемой при переходе к SOA является управление зависимостями и версионированием сервисов. При наличии множества сервисов, взаимодействующих друг с другом, важно обеспечить совместимость каждого сервиса с другими и правильную обработку версий.

Это требует надежного процесса тестирования и развертывания, а также использования таких инструментов и методов, как реестры сервисов и системы контроля версий.

Несмотря на трудности, несколько исследований показали, что переход к SOA может привести к значительным преимуществам. Например, исследование [3] показало, что переход к архитектуре SOA улучшает масштабируемость и доступность веб-приложения. Другое исследование [4] показало, что SOA может улучшить гибкость и адаптивность веб-приложения к изменяющимся бизнес-требованиям.

В целом, научная литература свидетельствует о том, что переход к SOA может обеспечить более гибкий и масштабируемый подход к разработке программного обеспечения по сравнению с монолитными архитектурами. Хотя процесс перехода может быть сложным и требует тщательного планирования и управления, преимущества могут быть значительными, особенно для больших и более сложных веб-приложений.

Переход от монолитной архитектуры к архитектуре MSA требует тщательного планирования и исполнения. Одной из ключевых проблем при переходе к MSA является проектирование самих служб [24]. Службы должны быть спроектированы так, чтобы быть небольшими и независимыми, с четким интерфейсом и функциями, и должны быть слабо связаны друг с другом, чтобы обеспечить возможность их изменения и масштабирования независимо друг от друга. Кроме того, каждая служба должна иметь собственное хранилище данных, что может создать проблемы с согласованностью и синхронизацией данных между службами.

Еще одной проблемой при переходе на MSA является управление зависимостями и координацией сервисов, что было описано в исследовании [32]. При наличии нескольких служб, взаимодействующих друг с другом, важно обеспечить совместимость каждой службы с другими и правильную обработку версий. Это требует надежного процесса тестирования и

развертывания, а также использования таких инструментов и методов, как реестры служб, балансировщики нагрузки и прерыватели цепи.

При переходе на MSA также могут возникнуть проблемы с согласованностью и безопасностью данных. Поскольку данные распределены по нескольким службам, важно обеспечить их согласованность и актуальность во всех службах. Кроме того, необходимо учитывать такие аспекты безопасности, как аутентификация, авторизация и шифрование, чтобы защитить конфиденциальные данные и предотвратить несанкционированный доступ.

Несмотря на трудности, несколько исследований показали, что переход на MSA может привести к значительным преимуществам. Например, исследование [5] показало, что переход на архитектуру MSA повысил скорость разработки и устойчивость веб-приложения. Другое исследование [7] показало, что MSA может улучшить масштабируемость и доступность веб-приложения.

В целом, научная литература свидетельствует о том, что переход на MSA может обеспечить очень гибкий и масштабируемый подход к разработке программного обеспечения с потенциалом значительных преимуществ. Хотя процесс перехода может быть сложным и требует тщательного планирования и управления, преимущества могут быть значительными, особенно для больших и более сложных веб-приложений.

Для более глубокого понимания проблемы использования монолитной архитектуры далее мы дадим ее подробное описание, а также рассмотрим преимущества и недостатки SOA и MSA архитектур.

1.2 Монолитная архитектура

Термин «монолит» первоначально обозначал массивный отдельный камень, но в программной инженерии он приобрел другое значение. Монолитный объект в данном контексте относится к архитектуре

программного обеспечения, которая объединяет все компоненты программы в единый, неделимый блок, как правило, на одной платформе [6]. Этот подход, известный как монолитное программное обеспечение, широко используется и сегодня, хотя некоторые разработчики считают его устаревшим.

Монолитное приложение обычно состоит из трех основных компонентов: базы данных, пользовательского интерфейса на стороне клиента и приложения на стороне сервера. Все части программного обеспечения интегрированы в единую программу, все функции которой управляются в одном месте. Взаимосвязанность компонентов облегчает самостоятельную работу с программным обеспечением, поэтому оно по-прежнему популярно среди небольших команд, например, стартапов.

Однако некоторые разработчики критически относятся к монолитной архитектуре из-за ее ограничений. Может быть сложно масштабировать и поддерживать программное обеспечение по мере роста приложения, а любые изменения в одной части программы могут повлиять на всю систему [20].

Несмотря на эти недостатки, монолитная архитектура остается традиционным решением для создания приложений, как показано на рисунке 1, где изображена схема структуры монолитного веб-приложения.

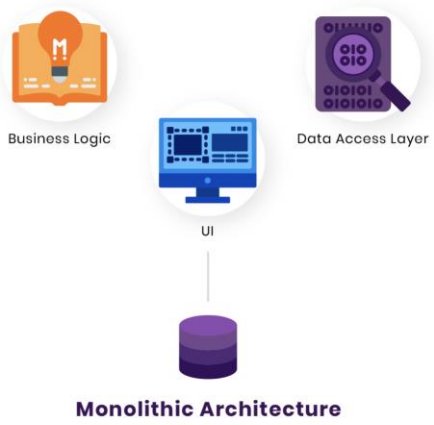

Рисунок 1 – Структура монолитного веб-приложения

Рассмотрим более подробно преимущества монолитной архитектуры [9]:

– простая разработка и процесс развертывания. Есть множество инструментов, которые можно интегрировать для облегчения разработки монолитного приложения. Кроме того, все действия выполняются с одним проектом, предусматривающим более легкое развертывание приложения на сервере. Имея монолитное ядро, разработчикам не нужно развертывать изменения или обновления в отдельности, поскольку они могут это делать сразу и сэкономить много времени;
– меньше кроссбраузерных проблем. Большинство приложений сталкиваются с проблемами кроссбраузерности. Монолитные приложения решают эти проблемы гораздо проще за счет их единственной кодовой базы. Проще разрешать эти проблемы, когда все работает в одном приложении;
– простота в масштабировании. Монолитные приложения просты в масштабировании по горизонтали путем запуска нескольких копий для балансировки нагрузки.

На ранних стадиях проекта монолитные веб-приложения работают хорошо, и в основном большинство существующих сегодня больших и успешных приложений начались как монолит.

Но существует достаточное количество недостатков монолитной архитектуры, которые приведены ниже:

– кодовая база с течением времени становится громоздкой. С течением времени большинство продуктов развивается и увеличивается по объему, и их структура становится размытой. Кодовая база начинает выглядеть действительно большой и становится трудно понимать и менять, особенно для новых разработчиков. Также становится труднее обнаружить побочные эффекты и зависимости. С ростом

проекта качество кодовой базы снижается, и интегрированная среда разработки (IDE) перегружается;

– сложность в интеграции новейших технологий. Если требуется добавить какую-либо новую технологию в приложение, разработчики могут столкнуться с препятствиями в адаптации кода. Добавление новой технологии означает значительные изменения в кодовой базе всей программы, что дорого и занимает много времени;
– ограниченная упругость. В монолитных приложениях каждое небольшое обновление требует полного развертывания приложения. Таким образом, все разработчики должны ждать, пока это будет сделано. Когда над одним проектом работает несколько команд, это может стать важной проблемой;
– монолитные программы также могут быть тяжело масштабируемыми, к примеру, когда разные модули имеют конфликтные требования к ресурсам;
– другой проблемой с монолитными приложениями является надежность. Ошибка в любом модуле (например, утечка памяти) может потенциально сбить весь процесс работы приложения. Более того, поскольку все экземпляры программы идентичны, эта ошибка влияет на доступность всей программы.

Монолитная модель не устарела и все еще отлично работает во многих проектах. Некоторые гигантские компании, как Etsy, остаются монолитными, невзирая на популярность микросервисов сегодня. Монолитная архитектура программного обеспечения может быть полезна, если команда разработчиков находится на стадии основания, создается новый продукт и у разработчиков нет опыта работы с микросервисами [23]. Монолит идеально подходит для стартапов, которым нужно как можно скорее запуститься. Однако определенные вопросы, упомянутые выше, делают монолиты не лучшим вариантом для современного приложения, который со временем будет расти и включать новые технологии.

1.3 Сервисно-ориентированная архитектура (SOA)

Естественным переходом от монолитной архитектуры являлось использование сервисно-ориентированной архитектуры (SOA). Используя такой подход, приложение разбивается на более мелкие модули. Тогда все службы работают со слоем агрегации, который можно назвать шиной (рисунок 2).

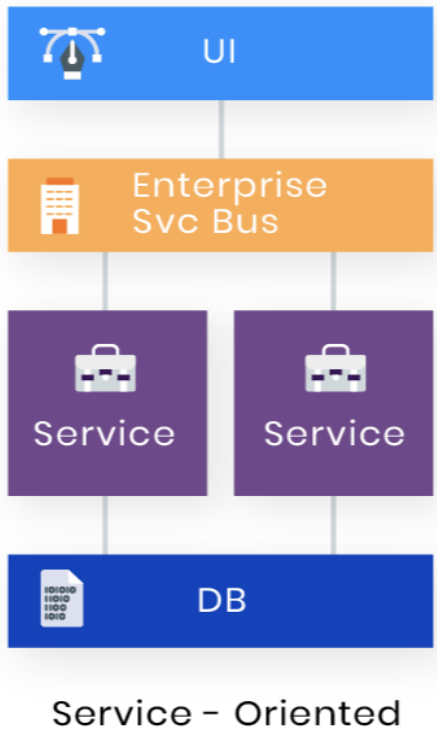

Рисунок 2 – Структура SOA приложения

SOA играет две главные роли: поставщик услуг и потребитель услуг. Обе эти роли может играть программный агент. Концепция SOA заключается в следующем: приложение может быть спроектировано и построено таким образом, что его модули интегрируются без проблем и могут быть легко использованы [1].

Относительно плюсов и минусов сервисно-ориентированной архитектуры [1].

Рассмотрим следующие преимущества сервис-ориентированной архитектуры (SOA):

– повторное использование сервисов: функциональные компоненты в SOA самодостаточны и могут быть повторно использованы в нескольких приложениях без влияния на другие сервисы;
– лучшие возможности поддержки приложений: поскольку каждый программный сервис является независимой сущностью, его можно легко обновлять и поддерживать, не вмешиваясь в работу других сервисов. Это особенно полезно для крупных корпоративных программ, которые могут быть разбиты на сервисы для упрощения управления;
– повышенная надежность: сервисы в SOA легче настраивать и тестировать, чем большие куски кода в монолитном подходе, что приводит к созданию более надежных продуктов;
– параллельная разработка: SOA состоит из слоев, что позволяет параллельно разрабатывать независимые сервисы и объединять их вместе.

Однако у SOA есть и ряд недостатков:

– сложное управление: основным недостатком SOA является ее сложность. Каждая служба должна обеспечивать одновременную доставку сообщений, и количество этих сообщений может превышать миллион одновременно, что делает управление всеми службами сложным;
– высокие инвестиционные затраты: разработка SOA требует значительных инвестиций в человеческие ресурсы, технологии и развитие;
– дополнительные перегрузки: в SOA все входы проверяются перед тем, как одна служба взаимодействует с другой службой, что

приводит к увеличению времени отклика и снижению общей эффективности при использовании нескольких служб.

В целом, слой агрегации (SOA шина) является самой большой проблемой для решения. Проблема заключается в добавлении оперативной логики в шину. Поскольку этот слой становится все больше и больше с добавлением все большего количества компонентов в систему, то возникают проблемы соединения внутри системы [18].

По мнению экспертов, еще одна самая большая проблема возникла в форме обращения с ошибками. С этой архитектурой система получает большое количество запросов за условную единицу времени без перерыва и не имеет возможности, после возникновения ошибки, обработать ее должным образом, что очень усложняет процесс поиска ошибок и их решения.

В целом SAO так и не получила большой популярности именно из-за описанных недостатков. Было отдано предпочтение монолитным приложением или же микросервисной архитектуре.

В заключение следует отметить, что хотя SOA предлагает ряд преимуществ, таких как повышенная надежность и параллельное развитие, она также имеет существенные недостатки, включая сложное управление, высокие инвестиционные затраты и дополнительные перегрузки.

1.4 Микросервисная архитектура (MSA)

Микросервисы — это архитектурный подход, используемый при создании сложных приложений. Хотя официального определения не существует, идея микросервисов заключается в построении приложений как набора небольших, самодостаточных и независимых сервисов. Каждый сервис отвечает за определенную задачу, и они взаимодействуют друг с другом в зависимости от необходимости выполнения того или иного действия (рисунок 3). Этот подход отличается от традиционного монолитного стиля, когда приложения создаются как единое целое [7].

В архитектуре микросервисов связь между сервисами осуществляется через API, а используемый язык программирования не имеет значения. Этот подход похож на работу команды, где каждый член сосредоточен на определенной задаче, и ему предоставляется свобода, доверие и ответственность, чтобы выполнить свою работу наилучшим образом. Такой подход обычно приводит к высокой производительности.

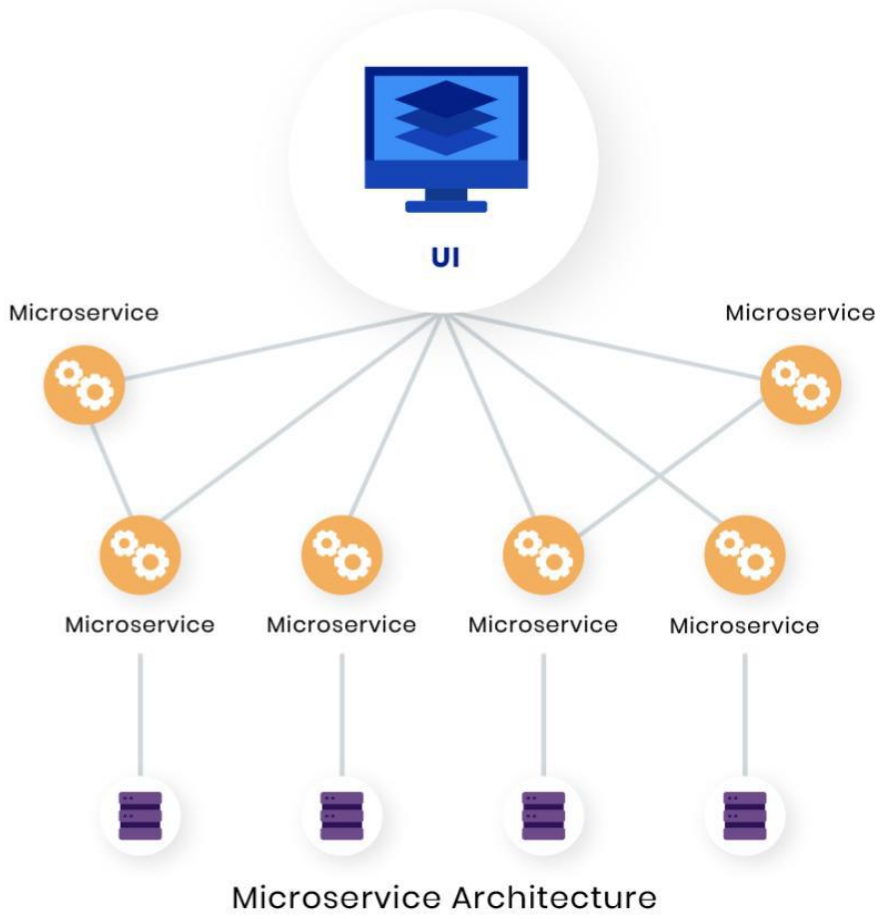

Рисунок 3 – Структура микросервисной архитектуры

Микросервисная архитектура может использоваться для создания новых приложений или разбиения монолитных приложений на более мелкие

сервисы. Использование микросервисов имеет множество преимуществ, в том числе:

- простота развертывания и быстрота изменений: микросервисы могут быть развернуты независимо друг от друга, что облегчает внесение изменений, не затрагивая остальную часть системы. Это отличается от монолитной архитектуры, где изменения требуют развертывания всей системы;
- модернизация без перестройки всей системы: отдельные микросервисы могут быть заменены или переписаны без необходимости перестраивать всю систему. Это означает, что приложение может быть легко модернизировано и обновлено с течением времени;
- легче поддерживать и тестировать: микросервисы обычно меньше по размеру, поэтому их легче понять, поддерживать и тестировать. Ошибки в одном микросервисе не повлияют на всю систему, и тестировщики могут сосредоточиться на тестировании отдельных микросервисов;
- совместимость с различными языками и платформами: микросервисы могут быть написаны на разных языках программирования и могут работать на разных платформах. Это позволяет более гибко подходить к выбору инструментов для работы.

Использование микросервисов позволяет нескольким командам работать над различными частями приложения, причем каждая команда специализируется в своей области знаний. Такой подход может привести к повышению производительности и лучшим результатам.

Что касается минусов микросервисов, то главным минусом данного архитектурного решения является сложность. Поддерживать работу большого количества компонентов, их независимого развертывания, логирования и т.п. является очень тяжелым процессом [16].

Подытожим преимущества и недостатки описанных архитектурных решений в таблице 1.

Таблица 1 – Сравнение архитектурных подходов к разработке веб-приложений

	Микросервисы	SOA	Монолитная архитектура
Дизайн	Сервисы построены как небольшие единицы, выраженные формально с помощью API, ориентированного на бизнес.	Услуги могут быть разного размера от небольших приложений к очень большим корпоративным сервисам, включая достаточно крупные бизнес-компоненты.	Монолитные программы вырастают до огромных размеров и возникают ситуации, когда понять всю программу сложно.
Удобство использования	Службы, подвергающиеся стандартному протоколу, например, RESTful API, и потребляются/повторяются другими службами и приложениями.	Сервисы, которые подвергаются стандартному протоколу, такому как SOAP и потребляются/повторяются другими службами.	Повторное использование монолитных приложений очень ограничено.
Масштабируемость	Сервисы существуют как независимые компоненты для развертывания, и доступна возможность отдельно масштабировать каждый сервис.	Зависимости между сервисами и многократными компонентами могут вызвать проблемы масштабирования.	Масштабирование монолитных приложений является задачей достаточно тяжелой
Гибкость	Малые независимые сервисы облегчают управление циклом разработки, тем самым, придавая высокую гибкость.	Высокая связь между компонентами ограничивает возможности менеджмента системы.	Для монолитных приложений достаточно трудно достичь гибкости в развертывании.

На основе этого можно заключить, что монолитные программы состоят из взаимосвязанных, неделимых единиц и отличаются очень низкой скоростью развития. SOA разбивается на более мелкие, умеренно совмещенные сервисы и все еще имеют медленное развитие, а микросервисы – это очень малые, неплотно связанные независимые сервисы, имеющие быстрое постоянное развитие.

Как видим из анализа, микросервисная архитектура (MSA) имеет ряд преимуществ перед сервис-ориентированной архитектурой (SOA), которые делают ее лучшим выбором в данном случае для решения поставленной задачи улучшения приложения с монолитной архитектурой. Визуальное сравнение показано на рисунке 4.

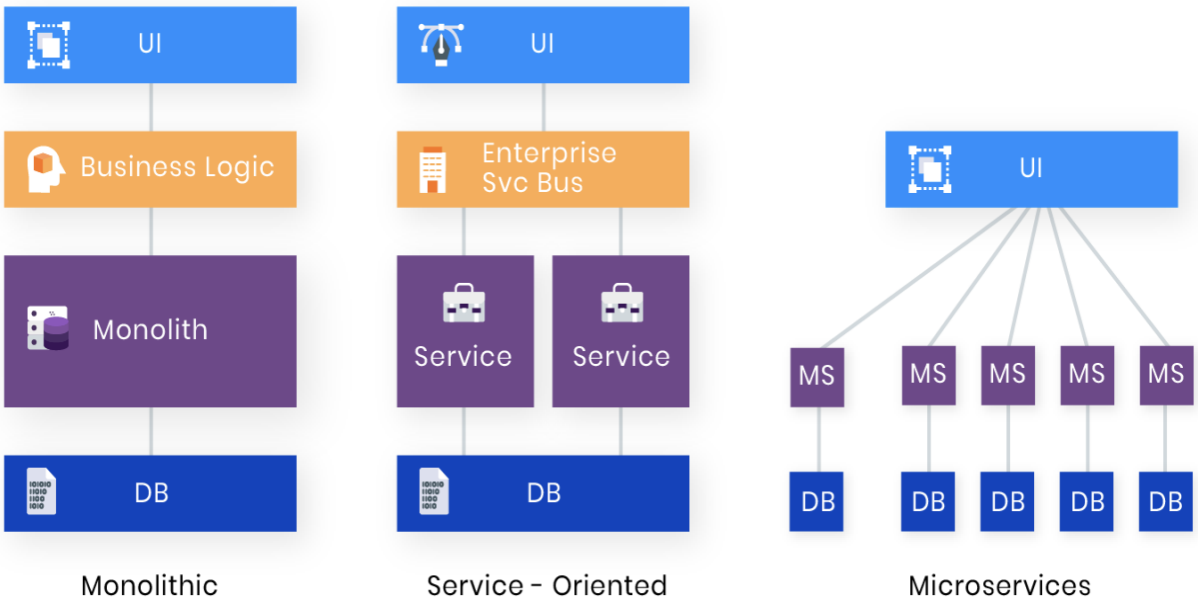

Рисунок 4 – Сравнение архитектурных решений Monolith, SOA и MSA

Вот некоторые из ключевых преимуществ MSA, которые мы выяснили:

– масштабируемость – MSA позволяет масштабировать отдельные сервисы независимо друг от друга, что означает, что система может

быть легко адаптирована к изменениям трафика или моделей использования;

- гибкость – MSA обеспечивает большую гибкость в выборе технологий, поскольку каждая услуга может быть разработана и развернута независимо от других. Это также позволяет легче модернизировать и изменять систему;
- устойчивость – MSA разработана с учетом устойчивости к сбоям, поскольку каждая услуга работает независимо и может продолжать функционировать, даже если другие услуги недоступны;
- гибкость – MSA позволяет организациям более гибко реагировать на изменения бизнес-требований или рыночных условий, поскольку новые услуги могут быть разработаны и развернуты быстро и легко.

В целом – MSA обеспечивает большую гибкость, масштабируемость, устойчивость и маневренность по сравнению с SOA. Эти преимущества делают MSA лучшим выбором для современных, сложных и высокораспределенных систем, где важна способность быстро реагировать на изменения и эффективно масштабироваться.

Выводы по главе 1

После анализа монолитных архитектур, архитектур SOA и MSA становится ясно, что монолитная архитектура имеет ограничения с точки зрения масштабируемости, ремонтопригодности и гибкости.

По мере усложнения приложений эти ограничения становятся все более очевидными. Поэтому новые архитектурные модели, такие как SOA и MSA, появились как потенциальные решения для преодоления этих ограничений и обеспечения более гибкого и масштабируемого подхода к разработке программного обеспечения.

Хотя SOA и MSA имеют свои плюсы и минусы, после тщательного сравнения этих двух моделей можно сделать вывод, что MSA имеет больше

плюсов и является лучшим выбором. MSA обеспечивает очень гибкий и масштабируемый подход к разработке программного обеспечения, что может привести к значительным преимуществам, таким как повышение скорости разработки, устойчивости, масштабируемости и доступности.

Однако процесс перехода может быть сложным и требует тщательного планирования и управления для обеспечения согласованности данных, безопасности и совместимости между сервисами.

Несмотря на трудности, научная литература свидетельствует о том, что переход на MSA может обеспечить более гибкий и масштабируемый подход к разработке программного обеспечения с потенциальными значительными преимуществами.

Глава 2 Проектирование архитектуры системы

2.1 Общее описание микросервисной архитектуры

Предположим, вы разрабатываете корпоративную программу на сервере. Она должна поддерживать множество разных клиентов, включая настольные браузеры, мобильные браузеры и гибридные мобильные приложения. Приложение также должно предоставлять API для использования сторонними ресурсами. Оно также должно интегрироваться с другими приложениями через веб-сервисы или брокеры сообщений. Приложение также должно обрабатывать запросы (HTTP-запросы и сообщения) и, выполняя бизнес-логику, предоставлять доступ к базе данных, разрешать обмен сообщениями с другими системами и возвращать ответы в форматах HTML/JSON/XML. Именно в таких случаях можно использовать микросервисную архитектуру.

Такое архитектурное решение дает массу преимуществ и позволяет осуществить описанную выше функциональность. Таким образом, использование микросервисной архитектуры предоставляет следующие преимущества:

- появляется возможность использовать непрерывную доставку и развертывание крупных, сложных приложений;
- каждый сервис имеет относительно небольшие размеры, а соответственно прост для понимания и легко подвергается изменениям и масштабированию;
- небольшой размер сервисов позволяет быстро и удобно тестировать работу каждого из них;
- упрощение процесса развертывания приложения на сервере разработки и возможность при этом развертывать отдельные сервисы не останавливая работу всего приложения;

– микросервисная архитектура позволяет организовать работу по разработке целого приложения вокруг нескольких автономных команд разработчиков. Каждая команда является владельцем и отвечает за один или несколько сервисов. Каждая команда может разрабатывать, тестировать, развертывать и масштабировать свои услуги независимо от всех других команд. Это все придает незаурядную гибкость при разработке;
– небольшой размер каждого микросервиса имеет больше преимуществ - сервисы становится легче для понимания разработчиками, а это ускоряет процесс разработки;
– IDE работает быстрее и позволяет ускорить работу команды разработчиков;
– запуск каждого микросервиса отдельно от других - также довольно быстрый процесс. Это также довольно сильно ускоряет и процесс разработки, и процесс развертывания приложения на сервере;
– улучшенная изоляция ошибок. Например, если в одном сервисе произошла утечка памяти, это повлияет только на этот сервис. Остальные сервисы будут продолжать обрабатывать запросы. Для сравнения, если один компонент монолитной архитектуры станет неисправным в результате ошибки — это может сбить всю систему;
– позволяет гибко выбирать технологии для разработки. При разработке каждого нового сервиса можно выбрать новый стек технологий. Также, внося существенные изменения в существующий сервис, вы можете переписать его с помощью нового стека технологий.

Но нужно понимать, что все же эти преимущества имеют и обратную сторону, а именно, - сложность в реализации и поддержке системы. Тем не менее, существует достаточно большое количество шаблонов, целью которых является упрощение системы и процесса разработки и поддержки такой архитектуры приложения. Некоторые из них мы рассмотрим далее.

2.2 Шаблоны микросервисной архитектуры

Когда речь идет о микросервисной архитектуре, существует множество шаблонов для проектирования и реализации этой архитектуры. Эти шаблоны призваны помочь организациям разбить монолитные приложения на более мелкие, независимые сервисы, которые можно разрабатывать, развертывать и поддерживать более эффективно. В этом разделе мы рассмотрим семь распространенных шаблонов для микросервисной архитектуры, которые приобрели популярность в последние годы. Пример разных шаблонов можно видеть на рисунке 5.

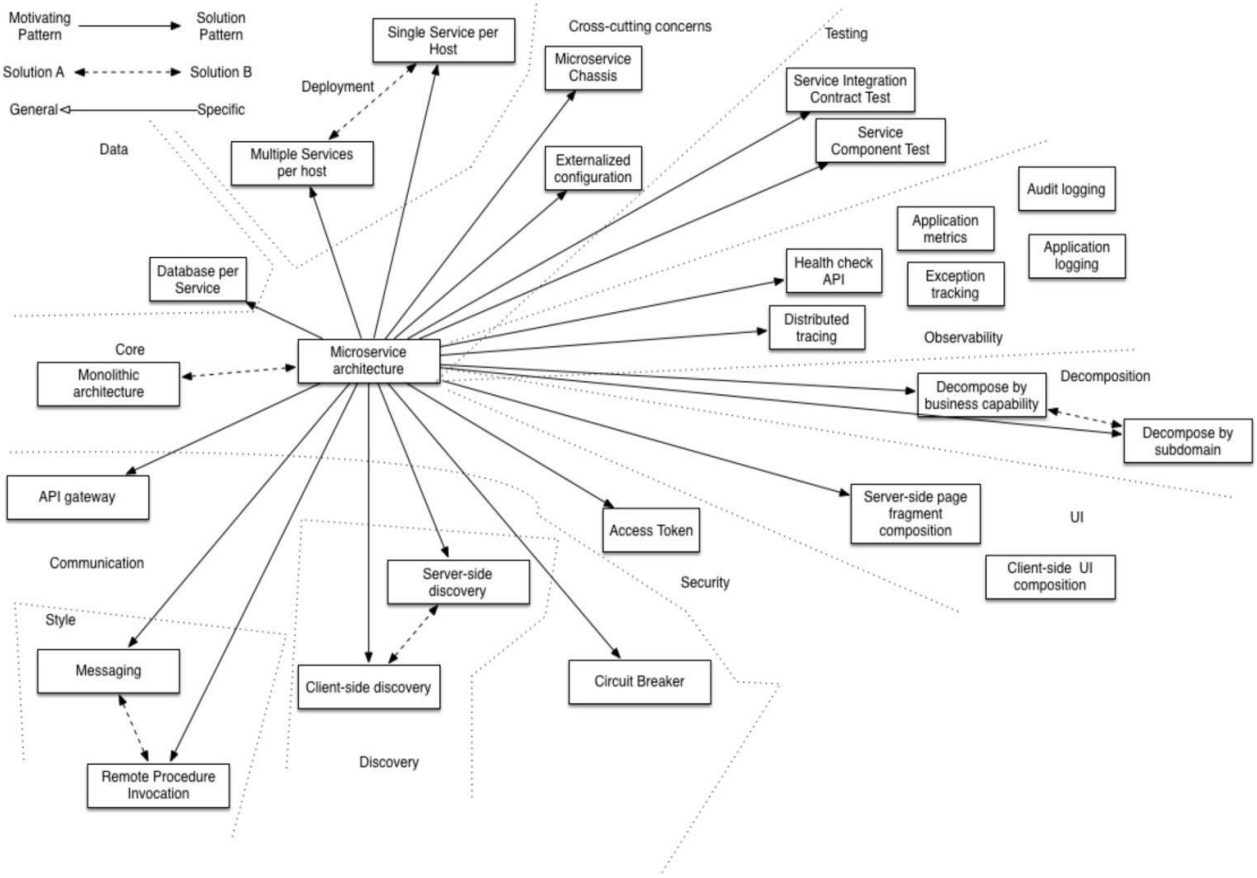

Рисунок 5 – Шаблоны микросервисной архитектуры

Первый шаблон, который мы рассмотрим, - «Decompose by business capability», который предполагает разбиение приложения на отдельные

сервисы на основе бизнес-возможностей, которые они поддерживают. Такой подход гарантирует, что каждый сервис отвечает за определенную бизнес-возможность, что облегчает управление и масштабирование.

Второй шаблон - «Database per service», который предполагает создание базы данных для каждой службы. Этот подход позволяет каждому сервису иметь собственную схему базы данных, что улучшает масштабируемость, снижает зависимость и упрощает управление данными в нескольких сервисах.

Третий шаблон - «API Gateway», который обеспечивает единую точку входа для всех клиентских запросов. Такой подход может упростить общую архитектуру и облегчить управление клиентскими запросами в нескольких сервисах.

Четвертый и пятый шаблоны - «Client-side service discovery» и «Server-side service discovery». Эти шаблоны предполагают создание механизма для обнаружения сервисов и их взаимодействия друг с другом. Подход на стороне клиента полагается на клиента для обнаружения сервисов, в то время как подход на стороне сервера включает в себя централизованный реестр сервисов.

Шестой шаблон — «Externalized configuration», который предполагает хранение информации о конфигурации вне кода приложения. Этот подход облегчает управление и изменение конфигураций без необходимости перекомпиляции или развертывания приложения.

Наконец, седьмой шаблон — это «Circuit Breaker», который предоставляет механизм для обработки отказов в распределенных системах. Этот подход помогает предотвратить каскадные сбои путем изоляции отказавших сервисов и предотвращения их влияния на другие сервисы [8].

В следующих подразделах мы более подробно рассмотрим каждый из этих шаблонов, изучим их преимущества и недостатки, а также дадим рекомендации по их эффективной реализации, а также проведем сравнение.

2.2.1 Шаблон «Decompose by business capability»

Разрабатываемая нами система ставит перед нами следующие задачи:

- архитектура приложения должна быть стабильна;
- сервисы должны быть сплочены. Сервис должен реализовывать небольшое количество связанных функций;
- сервисы должны быть разработаны согласно общему принципу закрытия – данные, которые изменяются вместе, должны быть упакованы вместе – чтобы гарантировать, что каждое изменение касается только одного сервиса;
- службы должны быть слабо связаны – каждый сервис функционирует как API, инкапсулирующий его реализацию, где реализацию можно изменить, не влияя на клиентов; сервис нужно просто тестировать;
- каждый сервис должен быть достаточно небольшим, чтобы его могла разрабатывать команда в 6 -10 разработчиков;
- каждая команда, владеющая одним или несколькими сервисами, должна быть автономной. Команда должна уметь разрабатывать и развертывать свои услуги при минимальном сотрудничестве с другими командами.

Решением данных задач является разделение системы на сервисы в соответствии с бизнес-возможностями. Бизнес-возможность — это концепция с моделированием архитектуры бизнеса. Собственно, это то, что делает бизнес, чтобы получить результат. Чаще всего бизнес-возможности опираются на бизнес-объекты, для примера менеджмент заказов отвечает за заказ, а менеджмент пользователей отвечает за пользователей. Приведем пример. Предположим, мы имеем следующие бизнес-возможности:

- менеджмент продукции;
- менеджмент запасами;
- менеджмент заказов;
- менеджмент доставки.

Тогда соответствующая микросервисная архитектура будет иметь следующее разбиение на сервисы (рисунок 6).

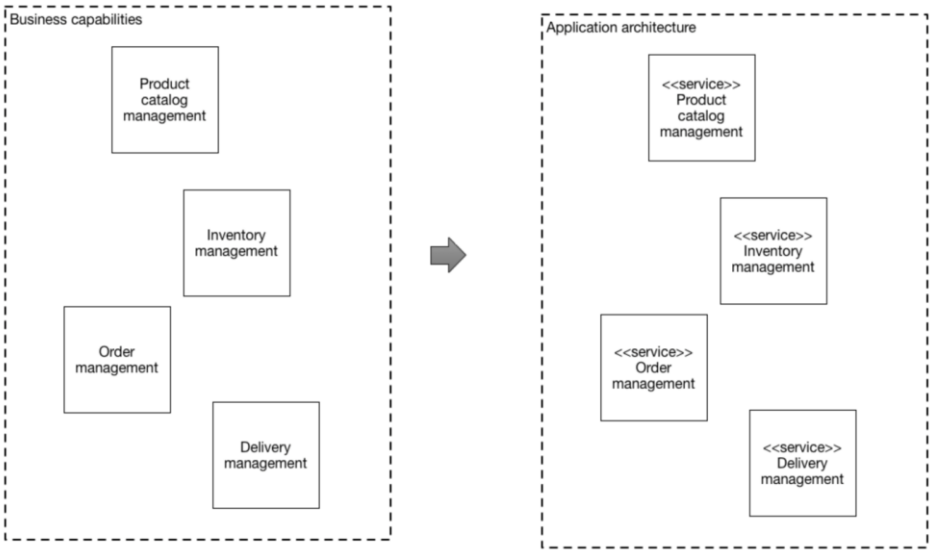

Рисунок 6 – Пример разбиения системы по бизнес-возможностям

Данный шаблон имеет следующие преимущества:

– стабильность архитектуры, поскольку бизнес-возможности чаще всего стабильны;
– команды разработчиков мультифункциональны, автономны и организованы вокруг бизнес-требований, а не вокруг технических аспектов;
– сервисы слабо связаны между собой;
– сервисы объединенные, то есть содержащие тесно связанные между собой функции.

2.2.2 Шаблон «Database per service»

Данный шаблон решает какой принцип следует использовать при работе с БД в приложении на основе микросервисов. Также он решает следующие проблемы:

– сервисы должны быть слабо связаны между собой, чтобы их можно было самостоятельно разрабатывать, развертывать и масштабировать;
– некоторые бизнес-транзакции должны модифицировать сущности разных сервисов;
– некоторые запросы должны иметь возможность получать данные, принадлежащие другим сервисам;
– базы данных должны быть скопированными и мигрированными;
– различные сервисы должны использовать различные базы данных. Для некоторых случаев более подходят реляционные базы данных, для некоторых лучше использовать NoSQL, такие как MongoDB, которые лучше подходят для хранения сложных, неструктурированных данных.

Главной идеей является - хранение данных каждого микросервиса индивидуальными для этого микросервиса и доступными только через API. Транзакции сервиса включают только работу с базой данных.

На рисунке 7 представлена идея данного шаблона.

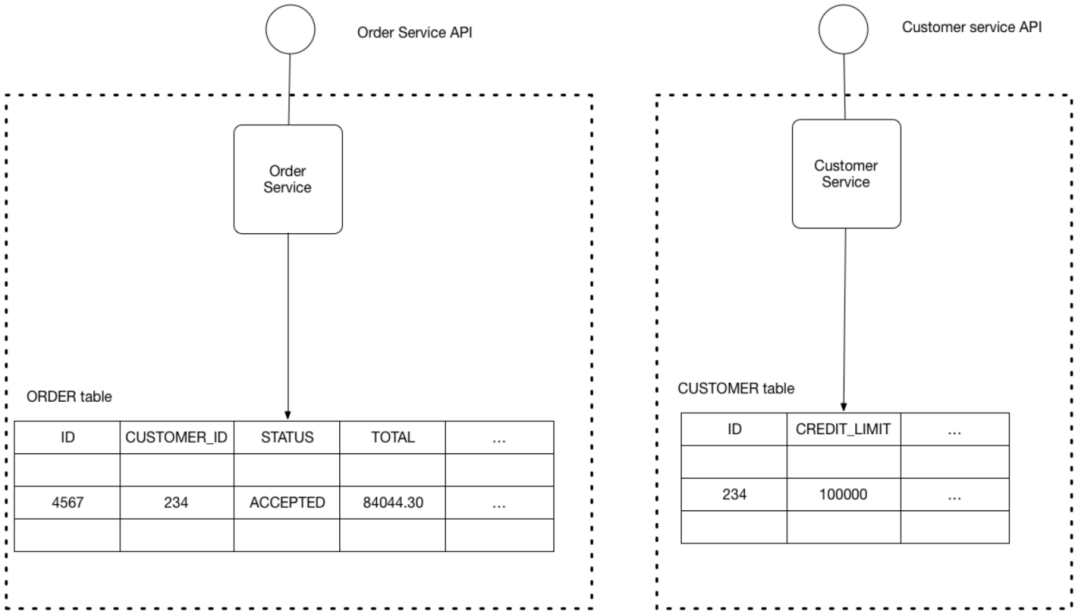

Рисунок 7 – Шаблон «Database per service»

База данных сервиса является частью сервиса и доступ к ней можно получить только через API самого сервиса.

Существует несколько возможностей хранить данные, относящиеся к бизнес-логике определенного сервиса. Собственно, не обязательно создавать отдельную базу данных для каждого сервиса. Скажем, если используется реляционная база данных, то возможны следующие варианты:

- отдельная таблица для отдельного сервиса – каждый сервис имеет свой набор таблиц, к которому можно получить доступ только с помощью API данного сервиса, но при этом саму БД используют и другие сервисы;
- отдельная схема для отдельного сервиса – каждый сервис имеет свои схемы, к которым можно получить доступ только с помощью API данного сервиса;
- отдельная база данных для отдельного сервиса – каждый сервис имеет свою базу данных.

Подытоживая, можно сказать, что данный шаблон предоставляет нам следующие преимущества:

- помогает убедиться, что сервисы слабо связаны. Внесение изменений в одну БД не влияет на другие;
- каждый сервис имеет возможность использовать свой тип БД, что придает гибкости при разработке.

2.2.3 Шаблон «API Gateway»

Данный шаблон, по сути, является одним из ключевых при разработке системы на основе микросервисов.

Использование API Gateway хорошо себя оправдывает, когда разрабатываются и создаются большие или сложные программы на основе микросервисной архитектуры с несколькими клиентскими частями. В целом, это сервис, обеспечивающий единую точку входа для определенных групп микросервисов. Этот шаблон похож на модель Фасад из объектно-ориентированного дизайна, но в этом случае это часть распределенной

системы. Шаблон API Gateway иногда также известен как «бэкенд для интерфейса» (BFF) [27].

Поэтому API Gateway находится между клиентской частью и микросервисами. Он выступает как обратный прокси, связывая запросы от клиентов к службам. Он также может обеспечить дополнительные сквозные функции, такие как проверка подлинности, обработка SSL и кэширование. На рисунке 8 схематически изображена структура данного шаблона.

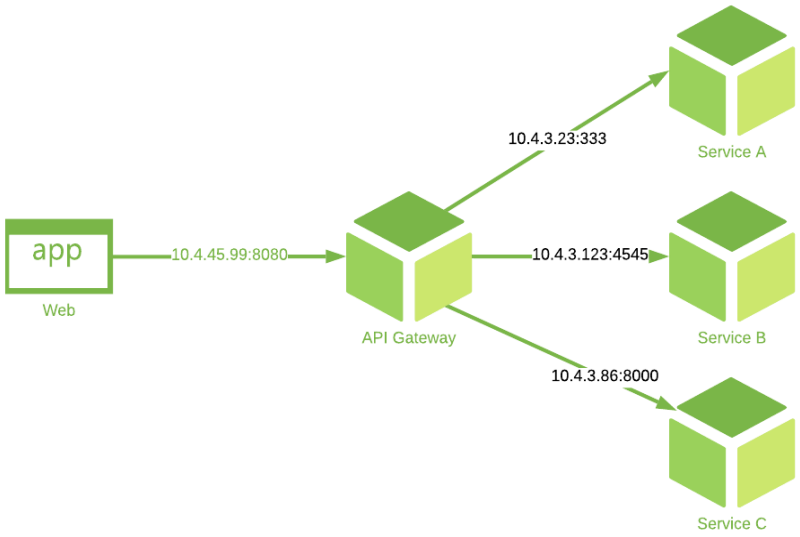

Рисунок 8 – API Gateway

К возникновению данного шаблона привели такие проблемы при разработке систем на основе микросервисов:

– требования для API, предоставляемые микросервисами, чаще всего отличаются от необходимых клиенту. Микросервисы обычно предоставляют гранулярные API, что означает, что клиенты должны взаимодействовать с несколькими службами. Например, когда клиент хочет получить данные о пользователе, ему могут понадобиться другие сервисы, кроме сервиса работы с пользователями, если скажем для клиентской части необходимы и

данные, сколько этот пользователь сделал покупок, и на какие события подписан и т.д.;

- различные клиенты нуждаются в различных данных. Например, версия данных, необходимая для браузерной версии приложения, может отличаться от версии данных, необходимых для мобильного приложения;
- производительность сети отличается для разных типов клиентов. Например, мобильная сеть, как правило, гораздо медленнее и имеет гораздо большую задержку, чем не мобильная сеть. И, конечно, любая WAN гораздо медленнее, чем LAN. Это означает, что гибридный мобильный клиент использует сеть, отличающуюся характеристиками производительности от локальной сети, которую использует веб-приложение на сервере. Веб-приложение, работающее на стороне сервера, может отправить множество запросов в различные сервисы, не влияя на процесс использования пользователями всего приложения, в то время как мобильный клиент может сделать только несколько запросов;
- количество развернутых экземпляров сервисов и их местонахождение (хост + порт) могут динамически изменяться;
- разграничение в сервисах может изменяться с течением времени и его следует скрывать от клиентов;
- сервисы могут использовать разнообразный набор протоколов, некоторые из которых могут не быть удобными для Интернета.

Реализацией API Gateway является единственная точка входа для всех клиентов. Поскольку API Gateway находится между всеми запросами от клиента к отдельным сервисам, он также выступает (PEP) для запросов к сервисам. Использование централизованного PEP означает, что сквозные проблемы, связанные с сервисами, могут быть решены в одном месте, избегая необходимости, чтобы отдельные команды разработчиков решали эти проблемы и могли разработать дополнительные сервисы.

2.2.4 Шаблон «Client-side service discovery»

Обычно сервисы выполняют запросы друг другу. В монолитной программе сервисы делают запросы друг к другу методами на уровне языка или вызовами на процедурном уровне. В традиционном развертывании распределенной системы сервисы работают в фиксированных, хорошо известных местах (хосты и порты), и поэтому они могут легко запросить друг друга с помощью HTTP/REST или какого-либо механизма RPC.

Однако современное приложение на основе микросервисов, как правило, работает в виртуализированных или контейнерных средах, где количество экземпляров сервиса и их расположение динамически изменяются.

Решением всех этих сложностей и стал шаблон «Client-side service discovery», который изображен на рисунке 9.

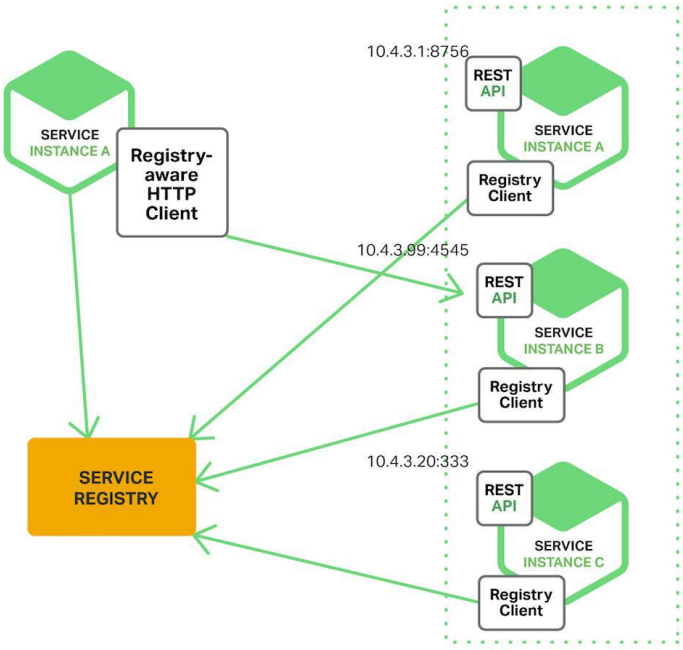

Рисунок 9 – Шаблон «Client-side service discovery»

Данный паттерн возник из-за следующих сложностей при построении системы на основе микросервисов:

– каждый экземпляр сервиса предоставляет удаленный API, такой как HTTP/REST и т.п. в определенном месте (хост и порт);
– количество экземпляров сервисов и их расположение динамически изменяются;
– виртуальным машинам и контейнерам обычно назначаются динамические IP-адреса.

Собственно, осуществляя запрос к сервису, клиент получает местоположение экземпляра сервиса путем запроса в SR (Service Registry), который знает местоположение всех экземпляров сервиса.

2.2.5 Шаблон «Server-side service discovery»

Данный шаблон является альтернативой описанному выше, то есть к шаблону «Client-side service discovery», и возник из-за тех же проблем. В чем же разница между «Server-side service discovery» и «Client-side service discovery»? При использовании шаблона «Server-side service discovery», осуществляя запрос к сервису, клиент делает запрос через маршрутизатор (т.н. Load Balancer). Схематически этот шаблон изображен на рисунке 10.

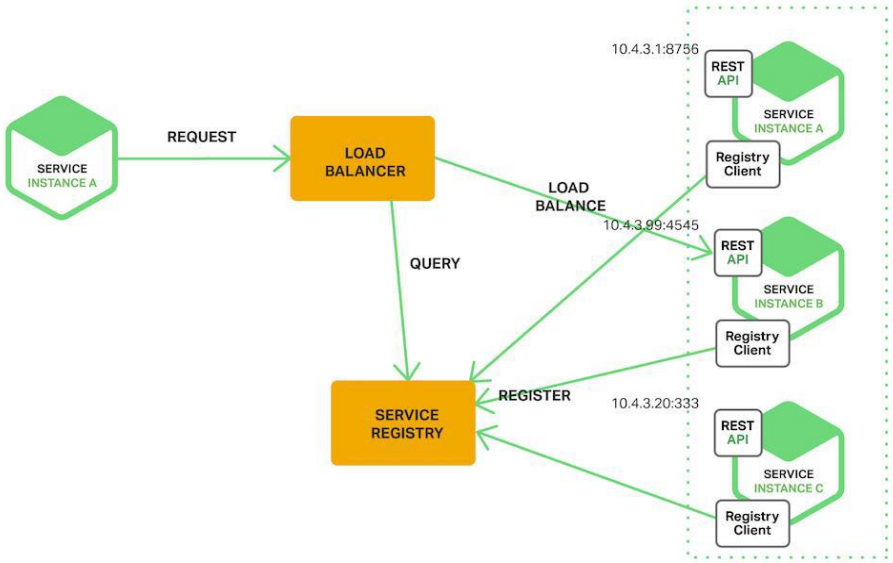

Рисунок 10 – Шаблон «Server-side service discovery»

Маршрутизатор запрашивает Service Registry, который может быть встроен в маршрутизатор и передает запрос доступному экземпляру сервиса.

2.2.6 Шаблон «Externalized configuration»

Данный шаблон отвечает на вопрос, как разрешить сервису работать на разных серверах с разными настройками и иметь возможность модифицировать настройки без переворачивания сервисов.

Шаблон «Externalized configuration» решает следующие вызовы: сервис должен быть снабжен данными конфигурации, указывающими, как подключиться к внешним/сторонним службам. К примеру, расположение сети и учетные данные базы данных; сервисы должны работать в нескольких средах – dev, test, qa, staging, production – без модификации и/или перекомпиляции; в разных средах разные экземпляры внешних/сторонних служб.

Главной идеей шаблона является экстернализация всего приложения, включая учетные данные базы данных, сетевое расположение и т.д. При запуске сервис считывает конфигурацию из внешнего источника, например, переменные среды ОС, конфигурационные файлы из системы GIT и т.д. Приведем пример, возьмем язык программирования Kotlin и веб-фреймворк Spring. Вместе они позволяют создавать следующие конструкции кода, изображенные на рисунке 11.

```
@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate)
extends RegistrationService {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _
```

Рисунок 11 – Пример использования конфигурационных переменных

Где user_registration_url это переменная, которая динамически считывается с сервиса конфигурации Spring, что позволяет задать необходимые переменные для различных сред и ситуаций.

Если необходимо изменить конфигурацию для определенного сервера и т.д., можно просто изменить конфигурационные файлы, которые хранятся как системные файлы или файлы под GIT, и т.д.

Сервер конфигураций автоматически распознает изменения и обновит измененную переменную, после чего используемые сервисы получат уже обновленное значение.

2.2.7 Шаблон «Circuit Breaker»

Контролировать большое количество сервисов достаточно тяжело. Скажем, мы имеем сервис A, который делает запрос к сервису B, а сервис B делает запрос еще к нескольким сервисам, и так далее. И вот один из сервисов дает сбой, как тогда избежать каскадного воздействия на другие сервисы и правильным образом обработать ошибку? Эту проблему решает шаблон Circuit Breaker [8].

Идея этого шаблона состоит в том, что клиент сервиса должен запрашивать удаленную услугу через прокси-сервер, функционирующий аналогично электрическому предохранителю. Когда количество последовательных отказов переступает порог, предохранитель выключается, и в течение продолжительности периода ожидания все попытки вызвать удаленный сервис немедленно прекращаются.

По истечении времени блокировки предохранитель позволяет выполнить ограниченное количество тестовых запросов. В случае успеха этих запросов предохранитель возобновит нормальную работу. В противном случае, если есть сбой, период ожидания начинается снова.

2.2.8 Сравнение шаблонов микросервисной архитектуры

Проведем сравнение шаблонов микросервисной архитектуры. В сравнительной таблице 2 представлен обзор преимуществ и недостатков семи различных шаблонов архитектуры микросервисов.

Каждый шаблон оценивается на основе его возможности способствовать гибкости, масштабируемости, отказоустойчивости и сопровождаемости в архитектуре микросервисов.

Таблица 2 – Сравнение шаблонов микросервисной архитектуры

Шаблон	Преимущества	Недостатки
Decompose by business capability	- Четкое разделение проблем - Позволяет легче масштабировать - Обеспечивает лучшее повторное использование кода	- Может привести к тому, что сервисы будут слишком гранулярными и ими будет сложно управлять - Требуется тщательный учет зависимостей между услугами
Database per service	- Обеспечивает лучшую изоляцию и уменьшает связь между сервисами - Легче масштабировать по горизонтали	- Может привести к увеличению сложности из-за нескольких экземпляров базы данных - Может привести к увеличению затрат на инфраструктуру
API Gateway	- Обеспечивает единую точку входа для всех сервисов - Можно легко реализовать политику безопасности, ограничения скорости и другие правила.	- Может возникнуть единая точка отказа - Может привести к превышению потерь производительности, если не реализовано должным образом.
Client-side service discovery	- Позволяет клиентам самостоятельно обнаруживать сервисы - Снижает нагрузку на шлюз API и реестр сервисов	- Клиентам необходимо реализовать логику обнаружения сервисов - Сервисы должны регистрироваться в реестре сервисов или сервисе обнаружения сервисов
Server-side service discovery	- Включает механизмы балансировки нагрузки и обхода отказа - Снижает нагрузку на клиентов	- Требуется дополнительная инфраструктура для реестра услуг - Может возникнуть единая точка отказа

Продолжение таблицы 2

Шаблон	Преимущества	Недостатки
Externalized configuration	- Позволяет изменять конфигурацию без перераспределения служб - Обеспечивает гибкость и упрощает обслуживание	- Может привести к увеличению сложности при неправильном управлении - Файлы конфигурации должны быть защищены и зарезервированы
Circuit Breaker	- Повышает доступность услуг за счет предотвращения каскадных сбоев - Обеспечивает лучшую отказоустойчивость	- Может создать дополнительную сложность в системе - Требуется тщательная настройка и конфигурация для эффективной работы

Шаблон «Decompose by business capability» четко разделяет проблемы, что позволяет легко масштабироваться и лучше использовать код повторно. Однако это может привести к тому, что сервисы будут слишком гранулярными и ими будет сложно управлять, а также требует тщательного рассмотрения зависимостей между сервисами.

Шаблон «Database per service» обеспечивает лучшую изоляцию и уменьшает связь между сервисами, что облегчает горизонтальное масштабирование. Но это может привести к увеличению сложности из-за наличия нескольких экземпляров базы данных, а также к увеличению затрат на инфраструктуру.

Шаблон «API Gateway» предоставляет единую точку входа для всех сервисов и легкую реализацию политик безопасности, ограничения скорости и других политик. Однако, при неправильной реализации он может создать единую точку отказа и привести к перерасходу производительности.

Шаблон «Client-side service discovery» позволяет клиентам самостоятельно обнаруживать службы и снижает нагрузку на шлюз API и реестр услуг. Но для этого клиентам необходимо реализовать логику

обнаружения сервисов, а сервисам необходимо зарегистрироваться в реестре сервисов или службе обнаружения.

Шаблон «Server-side service discovery» предоставляет возможность использования механизмов балансировки нагрузки и обхода отказа, а также позволяет снизить нагрузку на клиентов. Однако, он требует дополнительной инфраструктуры для регистрации сервисов и может привести к возникновению единой точки отказа.

Шаблон «Externalized configuration» дает возможность изменения конфигурации без перераспределения сервисов, гибкость и упрощение обслуживания. Но при неправильном управлении она может привести к повышенной сложности, а файлы конфигурации необходимо защищать и создавать резервные копии.

Шаблон «Circuit Breaker» дает преимущество повышения доступности услуг за счет предотвращения каскадных сбоев и обеспечения лучшей отказоустойчивости. Но и он может внести дополнительную сложность в систему и требует тщательной настройки и конфигурации для эффективной работы.

В заключении следует отметить, что каждый шаблон архитектуры микросервисов имеет свои сильные и слабые стороны. Выбор подходящего паттерна для архитектуры микросервисов зависит от конкретных потребностей, таких как сложность системы, количество сервисов, желаемый уровень отказоустойчивости и квалификация команды разработчиков. Поэтому важно тщательно оценить каждый паттерн и принять обоснованное решение на основе уникальных требований системы.

2.3 Общая схема прецедентов

После сравнения различных шаблонов для архитектуры микросервисов в предыдущем разделе пришло время перейти к проектированию архитектуры системы. В этой главе мы дадим общее описание архитектуры микросервисов

и подробно обсудим различные шаблоны. Затем мы сравним эти шаблоны, чтобы понять их сильные и слабые стороны. Наконец, мы представим общую схему прецедентов, которой мы будем руководствоваться при проектировании собственной микросервисной архитектуры. К концу этой главы вы будете иметь твердое понимание ключевых концепций и принципов микросервисной архитектуры и будете готовы приступить к проектированию собственной системы.

Схема микросервисной архитектуры рассматриваемой предметной области представлена на рисунке 12.

Построим диаграмму прецедентов (Use Case Diagram), которая позволит лучше понять, какое поведение должна реализовать система и какую функциональность иметь. После чего будет возможно начать проектировать систему.

На основе требований к системе были выделены следующие акторы:

– незарегистрированный пользователь – пользователь, зашедший на веб-сайт и еще не аутентифицировался в системе. Данный пользователь имеет возможность ознакомиться с системой и доступ только к базовой странице;
– зарегистрированный пользователь – пользователь, зарегистрировавшийся в системе и, соответственно, имеющий свой аккаунт. Данный пользователь может войти в систему, просмотреть и изменить собственную информацию;
– администратор – пользователь, который имеет доступ ко всем частям сайта и всей функциональности.

Схематически данные акторы изображены на рисунке 12.

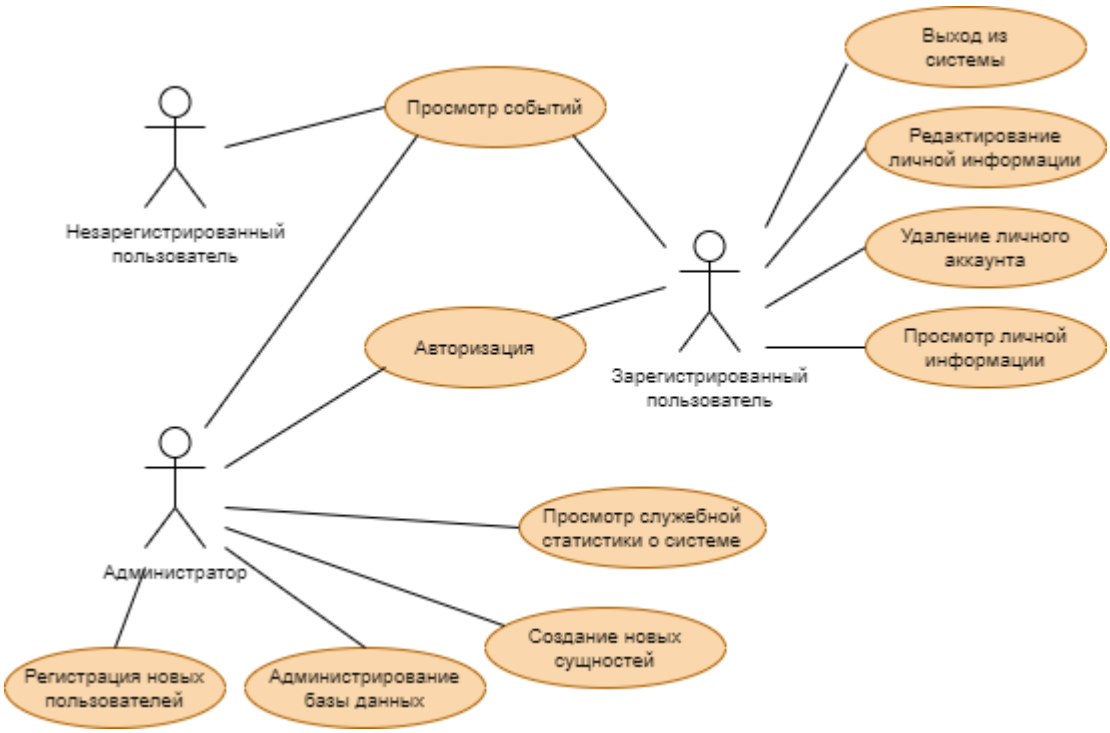

Рисунок 12 – Общая схема прецедентов

На основе выявленных акторов для данной системы можно определить следующие варианты использования.

Доступ к базовой странице — этот вариант использования доступен всем участникам системы, и он позволяет им получить доступ к базовой странице веб-сайта.

Регистрация — этот вариант использования доступен только для незарегистрированных пользователей и позволяет им создать новую учетную запись в системе.

Вход — этот вариант использования доступен зарегистрированному пользователю и администратору, и позволяет им войти в систему, используя свои учетные данные.

Просмотр собственной информации — этот вариант использования доступен зарегистрированному пользователю и администратору и позволяет им просматривать собственную информацию, например, данные профиля и настройки учетной записи.

Изменить собственную информацию — этот вариант использования доступен зарегистрированным пользователям и администраторам, и позволяет им изменять собственную информацию, такую как данные профиля и настройки учетной записи.

Управление пользователями — этот вариант использования доступен только администратору и позволяет ему управлять пользователями в системе, например, создавать новых пользователей, удалять пользователей и обновлять информацию о пользователях.

Управление настройками системы — этот вариант использования доступен только администратору, и он позволяет управлять настройками системы, такими как настройка шлюза API, управление сервером авторизации и настройка сервера реестра.

Диаграмма вариантов использования системы может быть представлена следующим образом:

На основе построенной диаграммы прецедентов мы можем создать сценарии прецедентов. Сценарий прецедентов – это определенные данные, описывающие конкретное взаимодействие между пользователем и системой.

Рассмотрим некоторые возможные сценарии для случаев использования, определенных в системе.

Доступ к базовой странице:

– незарегистрированный пользователь посещает сайт и может получить доступ к базовой странице сайта;
– зарегистрированный пользователь и администратор посещают сайт и могут получить доступ к базовой странице сайта.

Регистрация:

– незарегистрированный пользователь нажимает на кнопку «Регистрация» и перенаправляется на страницу регистрации;
– незарегистрированный пользователь заполняет регистрационную форму своей личной информацией и создает новую учетную запись;

– система генерирует письмо с подтверждением и отправляет его на адрес электронной почты пользователя;
– пользователь нажимает на ссылку подтверждения, и его учетная запись активируется.

Вход в систему:

– зарегистрированный пользователь или администратор нажимает на кнопку «Вход» и перенаправляется на страницу входа в систему;
– пользователь вводит свой адрес электронной почты и пароль и нажимает на кнопку «Войти»;
– система проверяет учетные данные пользователя и регистрирует его в системе.

Просмотр собственной информации:

– зарегистрированный пользователь или администратор входит в систему;
– пользователь нажимает на кнопку «Профиль» и перенаправляется на страницу своего профиля;
– пользователь может просмотреть свою личную информацию, такую как имя, адрес электронной почты и настройки учетной записи.

Изменение собственной информации:

– зарегистрированный пользователь или администратор входит в систему;
– пользователь нажимает на кнопку «Профиль» и перенаправляется на страницу своего профиля;
– пользователь может редактировать свою личную информацию, такую как имя, адрес электронной почты и настройки учетной записи;
– пользователь нажимает на кнопку «Сохранить», и изменения сохраняются в системе.

Управление пользователями:

– администратор входит в систему;

– администратор нажимает на кнопку «Пользователи» и перенаправляется на страницу управления пользователями;
– администратор может создавать новых пользователей, заполнив форму создания пользователя;
– администратор может удалить пользователей, выбрав их и нажав на кнопку «Удалить»;
– администратор может обновить информацию о пользователе, выбрав его и нажав на кнопку «Редактировать».

Управление настройками системы:

– администратор входит в систему;
– администратор нажимает на кнопку «Настройки» и перенаправляется на страницу настроек системы;
– администратор может настроить шлюз API, обновив параметры конфигурации шлюза API;
– администратор может управлять сервером авторизации, обновляя параметры конфигурации сервера авторизации;
– администратор может настроить сервер реестра, обновив параметры конфигурации сервера реестра.

2.4 Обзор языков и веб-фреймворков

2.4.1 C#/.NET

C# — это язык программирования, который по синтаксису имеет сходство с C++ и Java. Он является объектно-ориентированным и поддерживает такие функции, как полиморфизм, наследование, перегрузка операторов и статическая типизация, что позволяет создавать гибкие, масштабируемые и расширяемые приложения. C# также постоянно развивается, и в каждой новой версии появляются новые функциональные возможности, такие как лямбда-выражения, динамическое связывание и асинхронные методы.

Хотя C# часто используется в сочетании с технологиями платформы .NET, такими как Windows Forms, WPF, ASP.NET и Xamarin, важно отметить, что эти понятия не являются взаимозаменяемыми. Хотя C# был разработан специально для использования с платформой .NET, сам .NET является более широкой концепцией. На самом деле, Билл Гейтс даже сказал, что .NET — это одна из лучших вещей, созданных Microsoft. Фреймворк .NET обеспечивает надежную платформу для разработки приложений.

Основные конструктивные особенности данной платформы это [12]:

– совместимость: позволяющая программам, разработанным на .NET, получить доступ к функциональным возможностям программ, разработанных вне .NET;
– общий двигатель исполнения: также известен как общий язык исполнения, позволяющий программам, разработанным на .NET, обрабатывать общее поведение при использовании памяти, обработке исключений и безопасности;
– языковая независимость: общие спецификации языковой инфраструктуры (CLI) позволяют обмениваться типами данных между двумя программами, разработанными на разных языках;
– библиотека базовых классов: библиотека кода для наиболее распространенных функций, используемых программистами, чтобы избежать повторной перезаписи кода;
– простота развертывания: существуют инструменты для обеспечения простоты установки программ без вмешательства в ранее установленные программы.

.NET также позволяет разрабатывать приложения и на основе микросервисов.

Собственно, для этого необходимо создать обычное .Net веб-приложение.

Диалоговое окно создания нового проекта видим на рисунке 13.

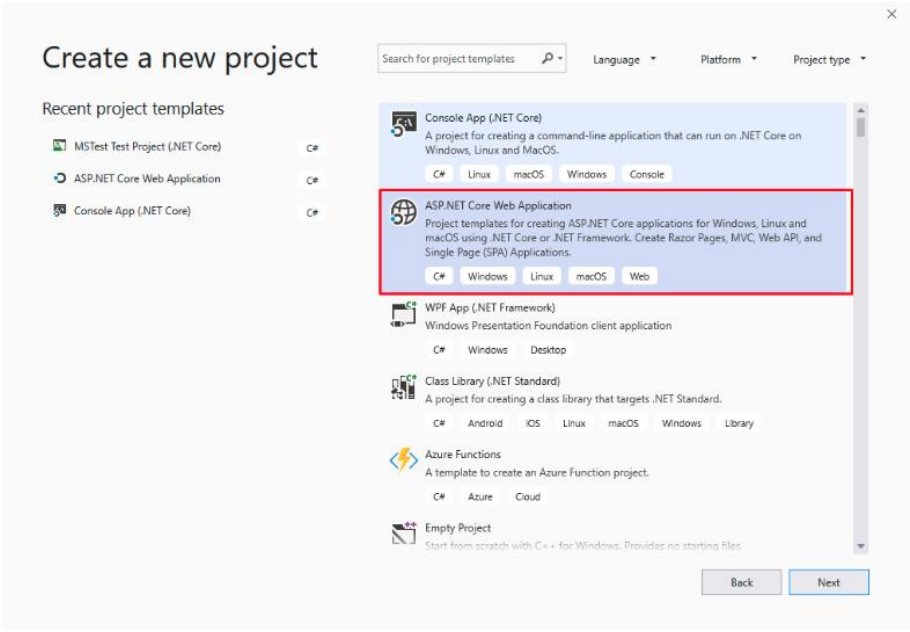

Рисунок 13 – Создание нового проекта на платформе .NET

После чего необходимо выбрать API опцию в следующем окне и выбрать поддержку Docker (рисунок 14).

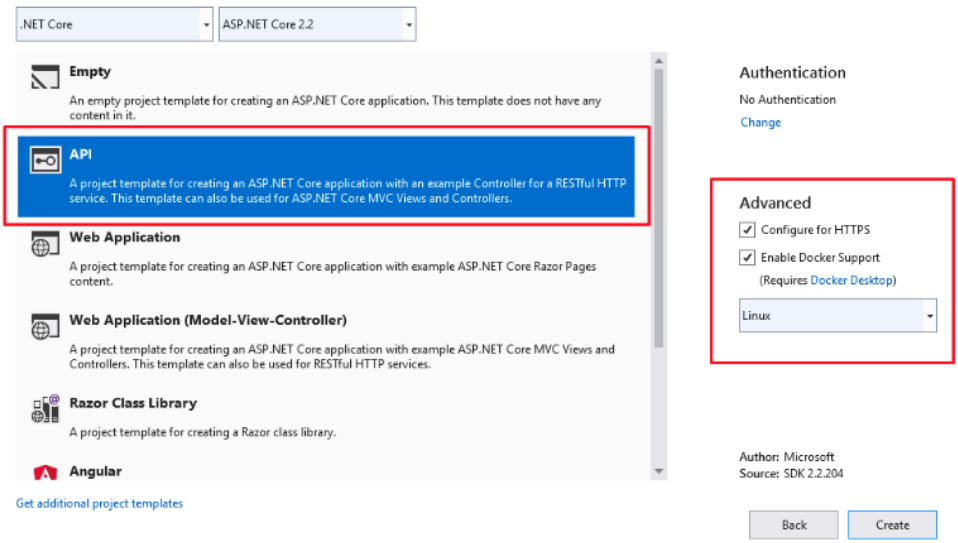

Рисунок 14 – Конфигурирование нового проекта для платформы .NET

Далее, соответственно, идет процесс разработки веб-сервиса и настройки Docker. Реализации всех паттернов для микросервисов, имеющихся в большом количестве библиотек и проектов, которые можно подключить к разрабатываемому приложению. Для примера реализацией API Gateway для .NET платформы является Ocelot, который сам по себе является открытым проектом, построенным с использованием языка C#, поддерживающего балансировку нагрузки, кэширования, может выступать как сервер авторизации или же прокси-сервер.

2.4.2 Java/Spring

Java – язык программирования высокого уровня, первоначально разработанный Sun Microsystems и выпущенный в 1995 году. Java работает на разных платформах, таких как Windows, Mac OS и разных версиях UNIX [31]. В список ключевых преимуществ языка можно включить [13]:

- объектно-ориентированный язык – в Java все является Объектом. Java можно легко расширить, поскольку она основана на модели Object;
- независимость от платформы – в отличие от многих других языков программирования, включая C и C++, когда Java компилируется, она не компилируется под конкретную платформу, а в независимый байт-код. Этот байткод распространяется через веб и интерпретируется виртуальной машиной (JVM) на любой платформе, на которой он работает;
- простота – Java разработана так, чтобы ее было легко изучить. Достаточно понять основную концепцию ООП Java;
- безопасность – благодаря защищенности Java, она позволяет разрабатывать системы, не боящиеся вирусов;
- портативность – байт-код делает Java портативным. Код Java можно запускать на любых платформах;

– надежность – Java имеет механизмы обработки ошибок времени выполнения и ошибок компиляции, что делает приложения разработанные на Java еще более надежными.

Существует достаточно много фреймворков для разработки веб-приложений на Java, и самым популярным является Spring. Spring Framework – это зрелый, мощный и очень гибкий фреймворк, ориентированный на создание веб-приложений на Java [29]. Одним из основных преимуществ Spring является то, что он заботится о большинстве аспектов низкого уровня создания программы, чтобы разработчики могли сосредоточиться на особенностях и логике бизнеса.

Еще одним важным моментом является то, что, хотя фреймворк достаточно зрел и хорошо отлажен, он очень активно поддерживается и имеет активное сообщество разработчиков. Это делает его достаточно современным и привлекательным для работы с экосистемой Java.

Spring Framework имеет множество модулей, которые позволяют разрабатывать разнообразные веб-приложения. Для разработки приложений на основе микросервисной архитектуры Spring Framework предоставляет библиотеки Spring Cloud, включающие в себя большое количество других библиотек для работы с микросервисами. Также Spring Cloud хорошо взаимодействует с библиотеками Netflix OOS [10].

Для создания различных Spring приложений компания Spring предоставляет удобный сервис Spring Initializr представленный на рисунке 15.

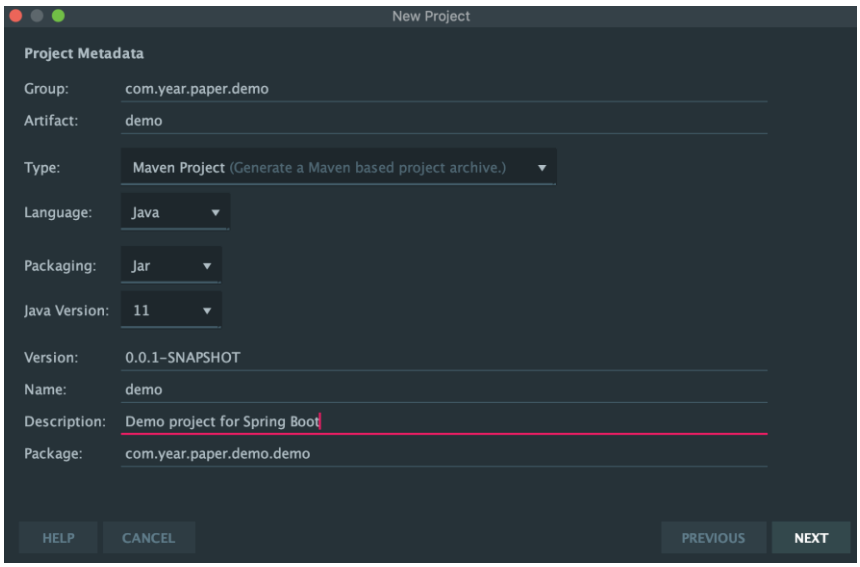

Рисунок 15 – Пример создания нового Spring проекта

После чего можно подключить все необходимые компоненты к проекту.

На данном этапе, уже существует возможность добавить интеграцию с Netflix OSS. В примере мы прилагаем к проекту поддержку библиотек Netflix Eureka, Netflix Zuul, Netflix Ribbon [28] (рисунок 16).

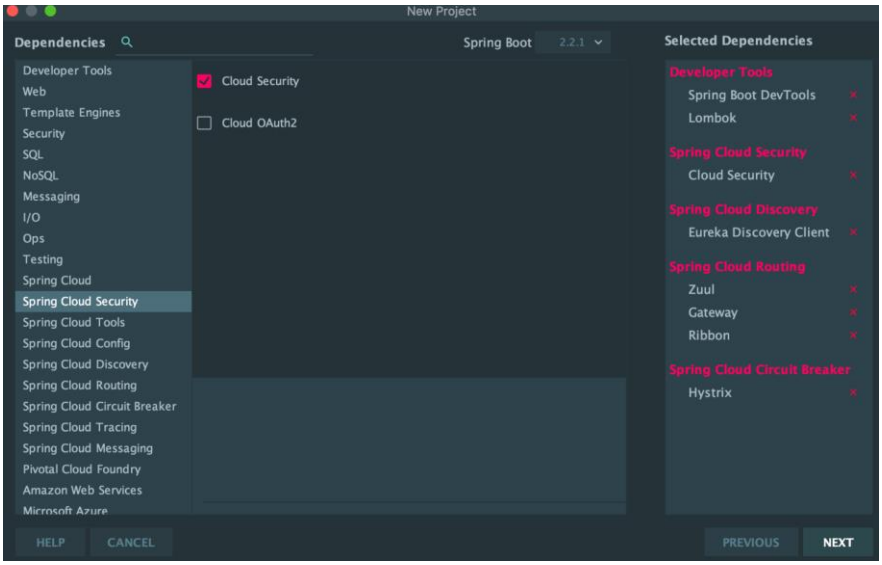

Рисунок 16 – Добавление зависимостей к проекту с помощью Spring Initializr

2.4.3 Ruby/Ruby on Rails

Ruby – интерпретируемый язык программирования высокого уровня. Обладает независимой от операционной системы реализацией многопоточности, строгой динамической типизацией, «собирателем мусора» и многими другими возможностями, поддерживающими много разных парадигм программирования, прежде всего классово-объектную. Ruby был задуман в 1993 году (24-го февраля) японцем Юкихиро Мацумото, стремящимся создать язык, объединяющий все качества других языков, способствующих облегчению труда программиста [2].

«Ruby on Rails» (или просто «Rails») – это фреймворк для разработки веб-приложений, написанный на языке программирования Ruby. Со времен своего дебюта в 2004 году Ruby on Rails довольно быстро стал одним из самых мощных и популярных инструментов для создания динамических веб-приложений. Rails используется множеством совершенно разных компаний: Airbnb, Basecamp, Disney, GitHub, Hulu, Kickstarter, Shopify, Twitter и Yellow Pages [2].

Что же делает Rails таким замечательным? Во-первых, Ruby on Rails – это на 100% открытый исходный код, доступный под MIT лицензией, и, как следствие, его загрузка и использование абсолютно бесплатны. В результате множество задач веб-программирования – таких как генерация HTML, создание моделей данных и маршрутизация URL – легки в Rails, а результирующий код приложений короткий и легко читаемый.

Rails также очень быстро адаптируется к новым тенденциям в веб-технологиях. К примеру, Rails был одним из первых, кто полностью реализовал архитектурный REST-стиль для структурирования веб-приложений. И когда другие фреймворки успешно разрабатывают новые техники, создатель Rails, Дэвид Хайнемайер и рабочая группа Rails не стесняются использовать их идеи [2]. Пожалуй, наиболее ярким и драматичным примером является слияние Rails и Merb (конкурирующий веб-

фреймворк), благодаря которому Rails получил модульный дизайн Merb, стабильный API и улучшенную производительность.

2.4.4 Node.JS/Express

Как асинхронное событие JavaScript-окружения, Node.js спроектирован для построения масштабируемых сетевых приложений. В приложениях Node.js для каждого соединения вызывается функция обратного вызова, однако, когда соединений нет Node.js засыпает [26].

Node.js создан под влиянием таких систем как Eventmachine в Ruby или Twisted в Python. Но при этом события модель, в нем, используется значительно шире, принимая event loop за основу окружения, а не в качестве отдельной библиотеки. В других системах же всегда происходит блокировка вызова, чтобы запустить цикл событий.

Express – самый популярный веб-фреймворк Node.js, являющийся базовой библиотекой для ряда других популярных веб-фреймворков Node. Js [17]. Этот фреймворк обеспечивает механизмы для таких вещей как: Написание обработчиков для HTTP/HTTPS запросов. Интеграция с клиентской частью. Добавление дополнительных слоев для дополнительной обработки запросов.

Хотя сам Express достаточно минималистичен, разработчики создали достаточно много пакетов программного обеспечения для решения практически любой проблемы веб-разработки. Есть библиотеки для работы с файлами cookie, сеансами, авторизацией пользователей, параметрами URL-адреса, данными POST, заголовками безопасности и многое другое.

Выводы по главе 2

В этой главе мы изучили микросервисную архитектуру, начав с общего описания того, что это такое и чем она отличается от традиционной монолитной архитектуры. Затем мы рассмотрели различные шаблоны, которые обычно используются для проектирования микросервисов, включая

«Decompose by business capability», «Database per service», «API Gateway», «Client-side service discovery», «Server-side service discovery», «Externalized configuration» и «Circuit Breaker».

Мы обсудили преимущества и недостатки каждого шаблона, а также рекомендации по их эффективной реализации. Кроме того, мы привели сравнение различных шаблонов, детально узнали их особенности и характеристики.

Также была представлена общая схема прецедентов, которая является полезным инструментом для выявления и анализа основных причин технических проблем в микросервисной архитектуре. Наконец, мы сделали обзор некоторых наиболее популярных языков и веб-фреймворков, используемых в микросервисной архитектуре, включая C#/.NET, Java/Spring, Ruby/Ruby on Rails и Node/Express.

В заключение следует отметить, что микросервисная архитектура обеспечивает гибкий, масштабируемый и устойчивый подход к разработке программного обеспечения. Разбивая приложения на более мелкие, независимые сервисы, организации могут повысить гибкость разработки, снизить затраты и повысить общую стабильность системы. Однако проектирование и внедрение микросервисной архитектуры требует тщательного планирования и учета различных доступных шаблонов и подходов.

Глава 3 Реализация микросервисной архитектуры

После анализа ключевых особенностей микросервисной архитектуры, а также обзора имеющихся технологий при работе с ними, мы можем перейти к процессу разработки системы. На рисунке 17 представлена схема микросервисной архитектуры разрабатываемого проекта. Система будет состоять из нескольких ключевых проектов, а именно:

– сервис API Gateway;
– сервер авторизации;
– registry Server;
– events Server.

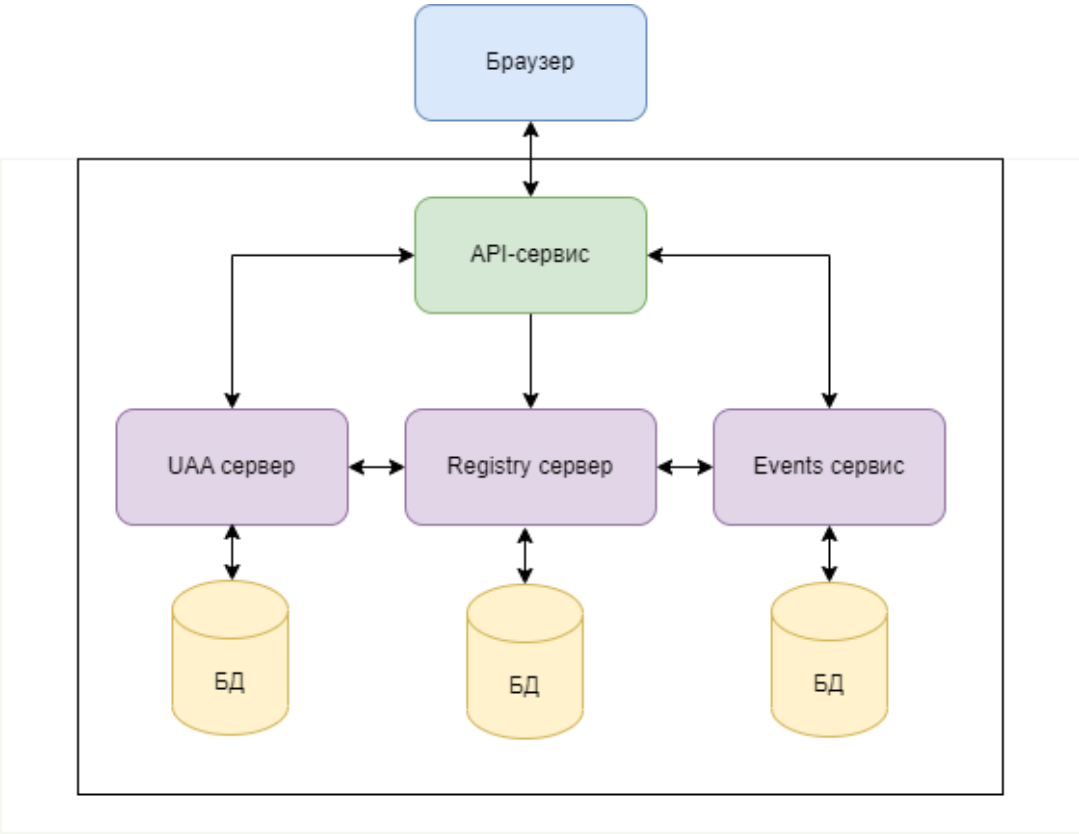

Рисунок 17 – Схема разработанной архитектуры

API Gateway служит точкой входа для внешних клиентов для доступа к сервисам системы. Он действует как API-шлюз, что означает, что он направляет запросы от внешних клиентов к соответствующим микросервисам системы. Шлюз API также действует как прокси-сервер, что позволяет ему кэшировать и оптимизировать ответы API.

Сервер авторизации генерирует JWT (JSON Web Tokens) [21] по запросу и обеспечивает средства авторизации и аутентификации для системы. JWT — это компактное, безопасное для URL средство представления утверждений для передачи между двумя сторонами. JWT можно использовать для авторизации и обмена информацией, и это безопасный способ передачи информации между сервисами.

Registry Server — это сервер конфигурации, сервер обнаружения и система мониторинга системы. Он служит центральной точкой для управления информацией о конфигурации и обнаружения микросервисов в системе. Он также отслеживает систему на наличие ошибок и других проблем и предоставляет предупреждения и уведомления системным администраторам.

Events Server — это микросервис, предоставляющий API для работы с событиями. Он получает запросы от других микросервисов в системе и выполняет необходимые действия для обработки событий.

В целом, архитектура этой системы разработана таким образом, чтобы быть масштабируемой, гибкой и надежной. Использование микросервисов позволяет разрабатывать и развертывать каждый сервис независимо, что помогает ускорить разработку и развертывание. Система также спроектирована как отказоустойчивая, каждая служба предназначена для изящного устранения сбоев, а сервер реестра служит центральным пунктом для мониторинга состояния системы.

3.1 API Gateway сервис

Разработка API Gateway сервиса начнется с создания проекта Spring. Процесс создания проекта Spring был описан во второй главе, поэтому здесь мы опишем лишь зависимости, которые необходимо подключить к данному проекту. Поскольку для сборки проекта мы используем технологию Maven, все зависимости будут описаны как артефакты xml [1].

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

Опишем необходимые нам зависимости [13]:

– Spring-cloud-starter – данная библиотека предоставляет все базовые средства для работы с Spring Cloud, представляющей собой экосистему по работе с микросервисной архитектурой;
– Spring-cloud-starter-netflix-eureka-client – библиотека Netflix OSS [25], позволяющая объявить наше приложение как Eureka Client, а это позволит в дальнейшем сервисе Registry находить данный сервис и реализовывать балансировку нагрузки на него;
– Spring-cloud-security – библиотека для поддержания авторизации и аутентификации. Доступ к сервисам системы будет предоставляться через API Gateway, а значит, JWT токен для авторизации тоже будет проходить через API Gateway, чтобы обрабатывать его правильным образом нам и нужна данная библиотека [5];
– Spring-cloud-starter-config – проекты, построенные на основе микросервисной архитектуры, содержат огромное количество конфигурационных файлов. Для того чтобы не усложнять и так сложные сервисы и сохранять конфигурационные файлы каждого из сервисов вместе с самим сервисами, было предложено разработать отдельный сервис, сохраняющий конфигурационные файлы всех

сервисов. Это будет удобным решением, ведь данный отдельный сервис будет предоставлять динамично изменяющуюся конфигурацию сервисов и не будет путаницы в поиске необходимых конфигурационных файлов;

- Spring-cloud-starter-netflix-hystrix – библиотека, придающая поддержку Netflix Hystrix [25], отвечающая за обработку задержек и отказов частей системы. Также данная библиотека предназначена для изоляции точек доступа к сторонним системам, правильной обработке каскадного отказа работы сервисов и обеспечению устойчивости системы в целом;
- Spring-cloud-starter-netflix-zuul – библиотека, в целом реализующая шаблон API Gateway, и в тесной связи с другими библиотеками отвечает за безопасность, мониторинг сервисов, динамическое связывание сервисов, устойчивость системы, балансировку нагрузки и многое другое прочего [13];
- Spring-cloud-starter-netflix-ribbon – библиотека, отвечающая за межпроцессорную коммуникацию (удаленные вызовы сервисов) и реализующая шаблон балансировки нагрузки [13].

Точкой входа в проект Spring является класс, обозначенный аннотацией @SpringBootApplication. В нашем случае класс также будет обозначен также аннотациями @EnableZuulProxy, @EnableDiscoveryClient, @EnableConfigurationProperties.

Аннотация @EnableZuulProxy обозначает этот проект как прокси Zuul. Это позволит нам переадресовать все запросы сделанные к данному API Gateway на соответствующие микросервисы. То есть клиентской части не нужно знать адреса каждого микросервиса, чтобы сделать запрос к нему. Учитывая то, что микросервисов может быть огромное количество и все они могут быть на различных портах. А еще, может быть запущено несколько экземпляров одного сервиса и нужно каким-то образом балансировать нагрузку на него и делать запросы по разным портам для доступа к одному и

тому же сервису. Также, Zuul позволяет использовать фильтры для работы с запросами, существуют так называемые pre-filters и post-filters, позволяющие обрабатывать запрос до его отправки и после его получения. Zuul хорошо работает вместе с Hystrix и Ribbon.

Общая схема такова: запросы, сделанные в Zuul сервисе, переадресуются в соответствующие микросервисы. Zuul использует Ribbon для того, чтобы узнать к какому именно сервису нужно переадресовывать запрос и каков его физический адрес, а также для балансировки нагрузки. При этом все запросы проходят через Hystrix, что позволяет собирать статистику и иметь метрику ошибок и соответственно обрабатывать сами ошибки. Диаграмма работы шаблона API Gateway изображена на рисунке 18.

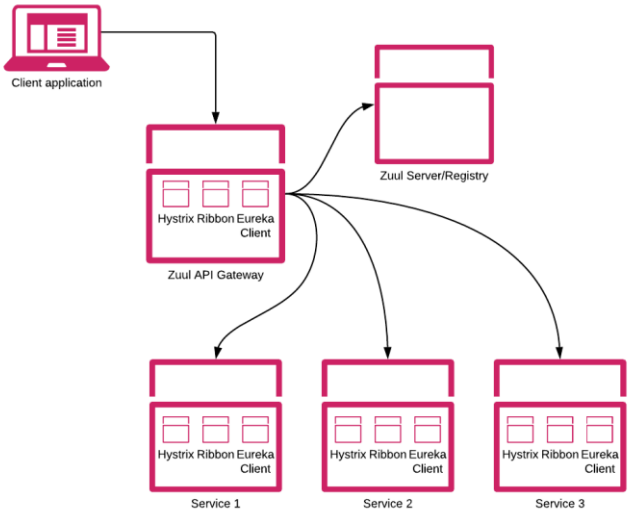

Рисунок 18 – Диаграмма работы API Gateway

Для базовой настройки Zuul, Hystrix и Ribbon нам необходимо модифицировать базовый конфигурационный файл проекта application.yml. Пример такого файла изображен на рисунке 19.

```
zuul:
  routes:
    echo:
      path: /myusers/**
      serviceId: myusers-service
      stripPrefix: true

hystrix:
  command:
    myusers-service:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: ...

myusers-service:
  ribbon:
    NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
    listOfServers: https://example1.com,http://example2.com
    ConnectTimeout: 1000
    ReadTimeout: 3000
    MaxTotalHttpConnections: 500
    MaxConnectionsPerHost: 100
```

Рисунок 19 – Пример конфигурационного файла

Здесь мы видим пример базовой конфигурации Zuul, Ribbon и Hystrix.

Значение serviceId, это id сервиса, зарегистрированного с помощью Eureka сервера, а в нашем случае Registry, поэтому отпадает необходимость записывать адрес сервиса полностью. А значение path, это путь, по которому будет доступно API сервису с id serviceId. На рисунке также изображены настройки Hystrix и Ribbon, но пока не будем на них останавливаться.

В целом это очень удобно, ведь на клиентской части не нужно думать о переадресации запросов, обрабатывать CORS и аутентификацию. Все это ложится на Zuul.

Аннотация @EnableDiscoveryClient регистрирует сервис для Netflix Eureka. Это означает, что данный сервис будет предоставлять метаинформацию о себе, такую как хост, порт, url для проверки жизнедеятельности сервиса, url домашней страницы и т.п. на сервер Eureka, что в нашем случае будет проектом Registry.

Для работы Eureka также необходимо добавить определенные настройки в конфигурационные файлы, которые изображены на рисунках 20 и 21.

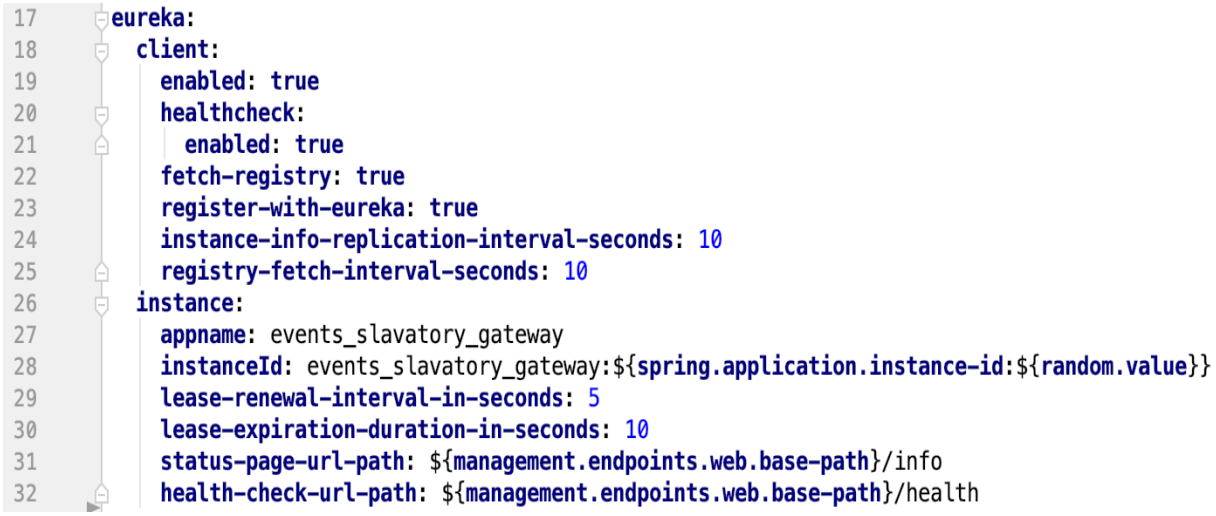

Рисунок 20 – Конфигурационный файл application.yml

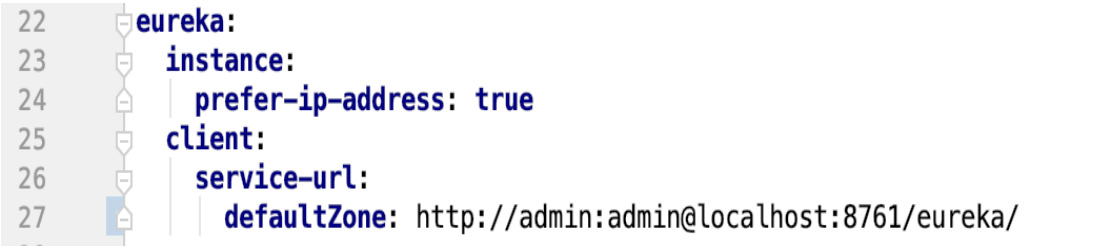

Рисунок 21 – Конфигурационный файл application-dev.yml

Как видим, в конфигурационном файле мы указываем путь к серверу Eureka/Discovery Server. Благодаря этому данный сервис будет знать, куда именно ему необходимо отправлять свои цели данные.

Далее опишем структуру API Gateway, представленную на рисунке 22.

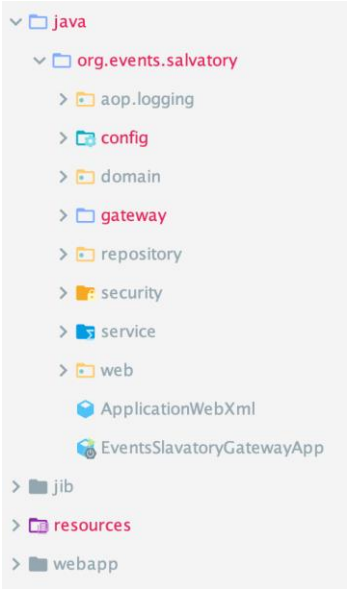

Рисунок 22 – Структура папок проекта API Gateway

Проект будет иметь следующую структуру папок:

– aop.logging – пакет, содержащий классы, отвечающие за логирование;
– config – пакет, содержащий конфигурационные классы, один из важнейших пакетов, так как содержащий все конфигурации сервиса;
– domain – пакет со вспомогательными классами;
– gateway – содержит Zuul фильтры, контролирующие доступ к микросервисам;
– security – содержит классы, отвечающие за аутентификацию и авторизацию пользователя и работу с UAA сервером;
– web – содержит контроллеры и фильтры для обновления JWT токена при его устаревании;
– resource – пакет, хранящий конфигурационные файлы;
– webapp – пакет, содержащий клиентскую часть кода.

Структура классов пакета config изображена на рисунке 23.

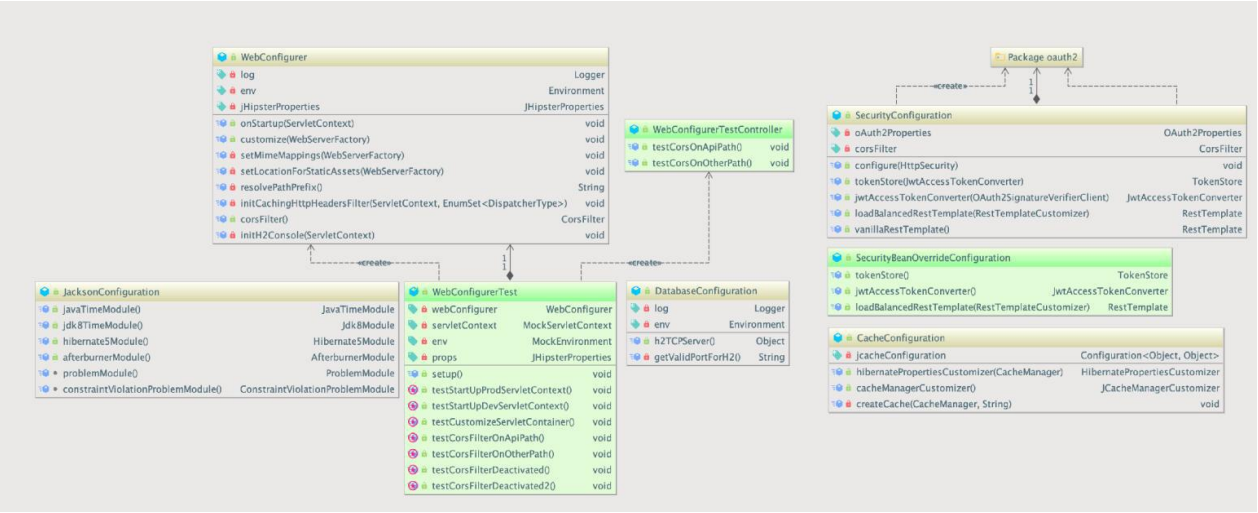

Рисунок 23 – Диаграмма классов пакета config

Рассмотрим более подробно каждый из ключевых конфигурационных классов пакета config:

– WebConfigurer – класс, задающий базовые веб-настройки, такие как кэширование хедеров запроса, конфигурация CORS фильтра и т.п.;
– DatabaseConfiguration – класс, содержащий конфигурацию баз данных, и отвечающий за инициализацию H2 базы данных при разработке;
– SecurityConfiguration – класс, отвечающий за настройку безопасности. В данном классе мы конфигурируем, какие пути доступны для обычного пользователя, а какие для администратора. Задаем алгоритм конвертации JWT токена, конфигурируем класс, используемый для того, чтобы делать запросы между микросервисами и т.д.
– Остальные конфигурационные классы являются второстепенными.

Далее рассмотрим классы пакета web, изображенные на рисунке 24.

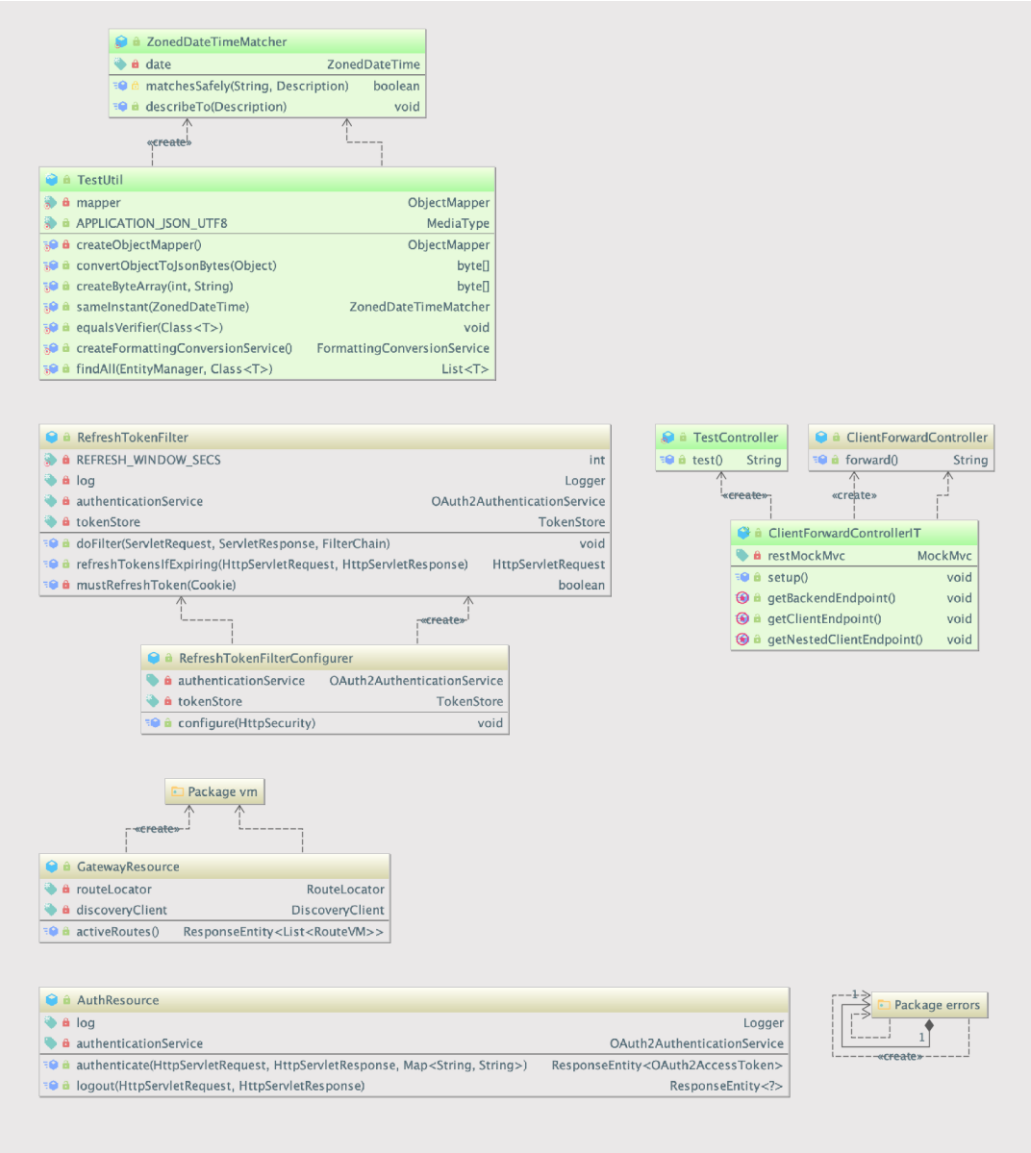

Рисунок 24 – Диаграмма классов пакета web

Здесь следует выделить классы контроллеры GatewayResource, ClientForward, AuthResource, отвечающие за обработку запросов клиента и отдачу определенного ответа на запрос.

Пакет Gateway содержит Zuul фильтры, обрабатывающие запросы при получении и перед отправкой (рисунок 25).

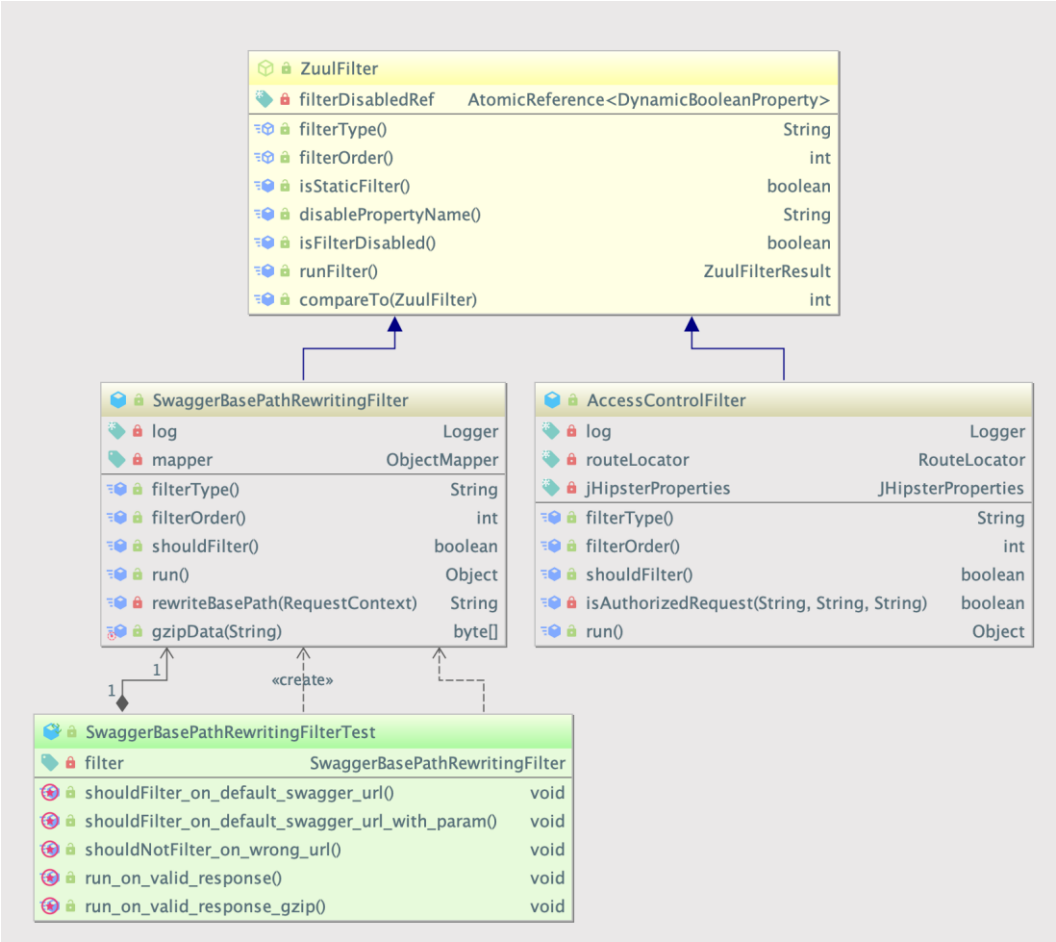

Рисунок 25 – Диаграмма классов пакета gateway

Следующим ключевым пакетом для сервиса является пакет security (рисунок 26). Данный пакет содержит классы по логике работы с JWT токеном и сервером авторизации в целом.

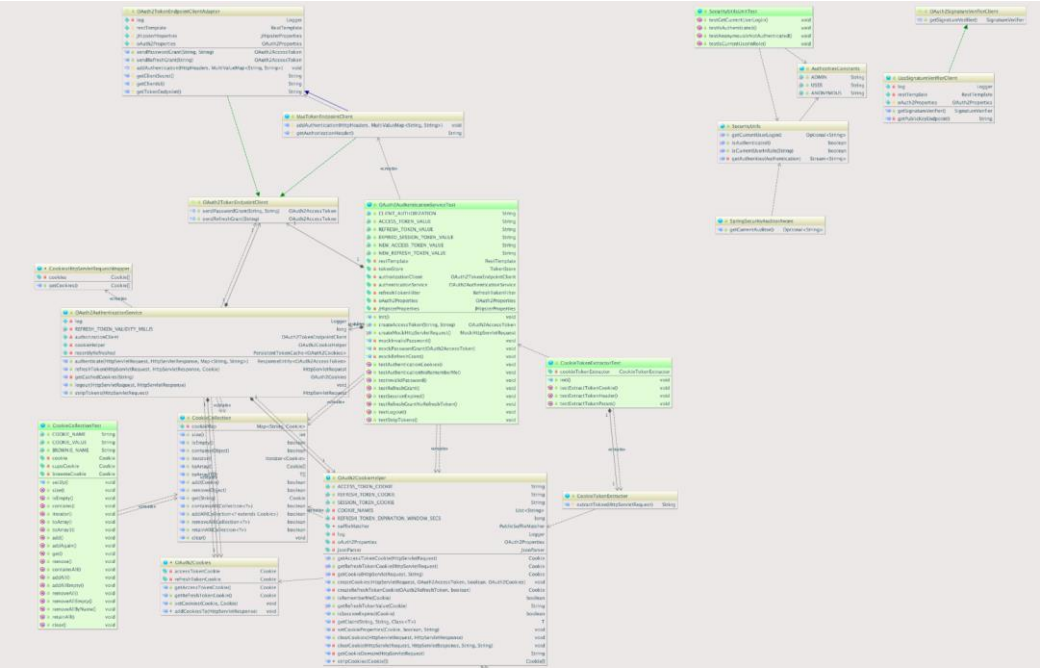

Рисунок 26 – Диаграмма классов пакета security

Класс UaaTokenEndpointClient реализует функции клиента доступа к UAA серверу для получения и валидации JWT токена и пользовательских данных.

Класс OAuth2Cookies сохраняет значение JWT токена и добавляет их к cookie при успешной аутентификации.

OAuth2AuthenticationService класс управляет процессом аутентификации, может делать запросы на получение JWT токена, запрос на обновление токена и очистить все куки, которые содержат информацию, относящуюся к пользователю, тем самым излагая пользователя.

3.2 UAA сервер

Следующим сервисом для реализации будет UAA сервис. UAA (User Account and Authentication) сервис будет отвечать за процессы авторизации и проверки подлинности и защищать систему используя OAuth2 протокол.

OAuth2 – это протокол, позволяющий реализующей системе отвечать таким запросам как:

– иметь центральный механизм аутентификации. Поскольку микросервисы это независимые и автономные программы, мы хотим быть уверены, что запрос пользователя не будет обрабатываться разными приложениями, возможно, разными системами защиты;
– отсутствие состояния. То есть данные о том, аутентифицированный ли пользователь не хранятся в сессии. Для этого используется токен JWT, который может храниться в cookies или в локальном хранилище [6]. И поскольку основное из главных преимуществ микросервисной архитектуры – это легкая масштабируемость, выбранное решение в поддержке безопасности не должно влиять на это ключевое преимущество;
– разница в механизмах получения доступа к системе для пользователей и машин. Использование микросервисной архитектуры приводит к созданию крупного многоцелевого центра обработки данных из разных доменов и ресурсов, поэтому возникает необходимость ограничивать доступ разных клиентов, таких как нативные программы, разные SPA и т.д.;
– разрешение контроля доступа. При сохранении централизованных ролей существует потребность в настройке разных политик контроля доступа для каждого микросервиса. Микросервис не должен отдавать себе отчет в распознавании пользователей и должен просто авторизировать входящие запросы;
– быть хорошо защищенными от атак;
– быть легко масштабируемым.

В целом алгоритм работы с сервером проверки подлинности можно описать следующим образом:

– каждый запрос к любому ресурсу по URL выполняется с помощью клиента;

- клиент — это некоторая абстракция, которая может быть как http клиентом, так и REST клиентом;
- клиент также может использоваться для аутентификации пользователя, например, с помощью ajax запроса из клиентской части;
- каждый микросервис, включая UAA сервер, предоставляющий определенные ресурсы по определенным URL, является так называемым ресурсным сервером;
- стрелки синего цвета изображают запросы аутентификации к серверу аутентификации UAA;
- стрелки зеленого цвета изображают запросы из клиентов к ресурсным серверам;
- UAA сервер представляет собой комбинацию сервера авторизации и ресурсного сервера;
- UAA сервер является владельцем всех данных внутри проекта и контролирует доступ к ним;
- клиенты, получающие доступ к ресурсам с аутентификацией пользователя, аутентифицируются с помощью «password grand» с идентификатором клиента (client id) и секретом (secret), безопасно хранящимся в файлах конфигурации Registry;
- клиенты, получающие доступ к ресурсам без пользовательских данных, аутентифицируются с помощью «client credentials grant»; каждый клиент определяется в конфигурационных файлах UAA сервера.

Визуальная схема работы сервера авторизации и проверки подлинности представлена на рисунке 27.

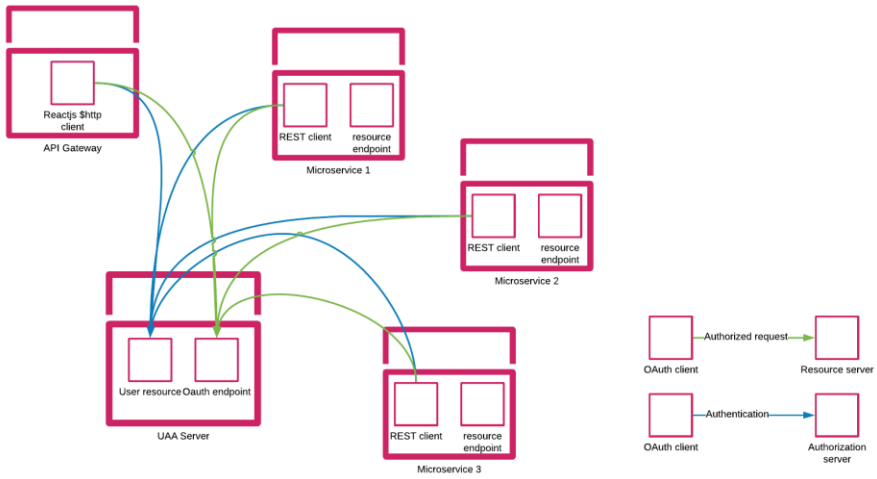

Рисунок 27 – Схема работы UAA сервера

В целом UAA сервер выполняет следующие три функции:

– предоставляет доступ к ресурсу, содержащему данные о пользователе и аккаунтах;
– реализует интерфейс AuthorizationServerConfigurerAdapter для реализации OAuth2 и определяет базовые клиенты, такие как web_app, internal и т.п.;
– генерирует JWT публичный ключ, который должен использоваться всеми другими микросервисами.

Когда микросервис загружается, обычно он ожидает, что сервер UAA уже готов предоставить свой открытый ключ. Служба сначала вызывает /oauth/token_key, чтобы получить открытый ключ и настроить его для подписания токена (JwtAccessTokenConverter) [8].

Для запросов пользователя на вход посылается запрос в API Gateway /auth/login. Эта конечная точка использует OAuth2TokenEndpointClientAdapter для отправки запроса UAA, что подтверждает аутентификацию с предоставлением пароля. Поскольку этот запрос происходит и обрабатывается в API Gateway, client id и secret не хранятся в коде клиента и недоступны для пользователей. API Gateway возвращает новый файл cookie, содержащий JWT

token, и этот файл cookie посылается с каждым запросом, выполненным от клиента, к серверной части проекта. Там он валидируется и при успешной валидации позволяет использовать определенные ресурсы и обрабатывать их [8].

Рассмотрим разрабатываемое приложение поближе. Зависимости, которые нужно добавить к проекту, похожи на те, что и для API Gateway. Разве что здесь необходимо также добавить следующие дополнительные зависимости:

– spring-boot-starter-data-jpa;
– spring-boot-starter-mail;
– spring-boot-starter-security;
– spring-security-jwt;
– spring-security-data.

Они помогут нам в работе с базами данных и JWT токеном.

Данный проект имеет аналогичную структуру папок к API Gateway. Структура проекта UAA сервера представлена на рисунке 28.

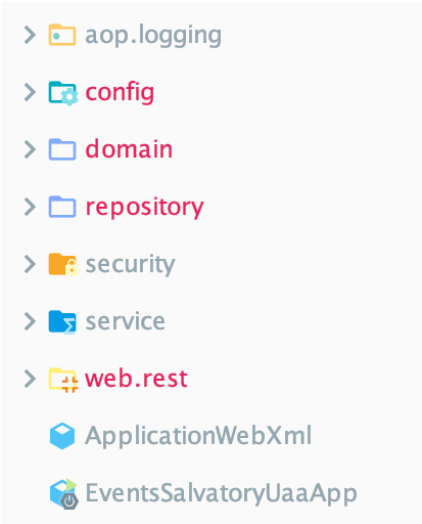

Рисунок 28 – Структура проекта UAA сервера

В данном случае у нас есть такие ключевые пакеты.

Config – в данном случае пакет содержит конфигурационные файлы для сервера авторизации.

Domain – содержит классы, сущности для работы с базой данных. На самом деле это ни что иное, как POJO классы, представляющие собой данные, которые можно хранить в базе данных. Сущность представляет собой таблицу, хранящуюся в базе данных. В то же время каждый экземпляр сущности представляет собой строку в таблице. Схематически пакет изображен на рисунке 29.

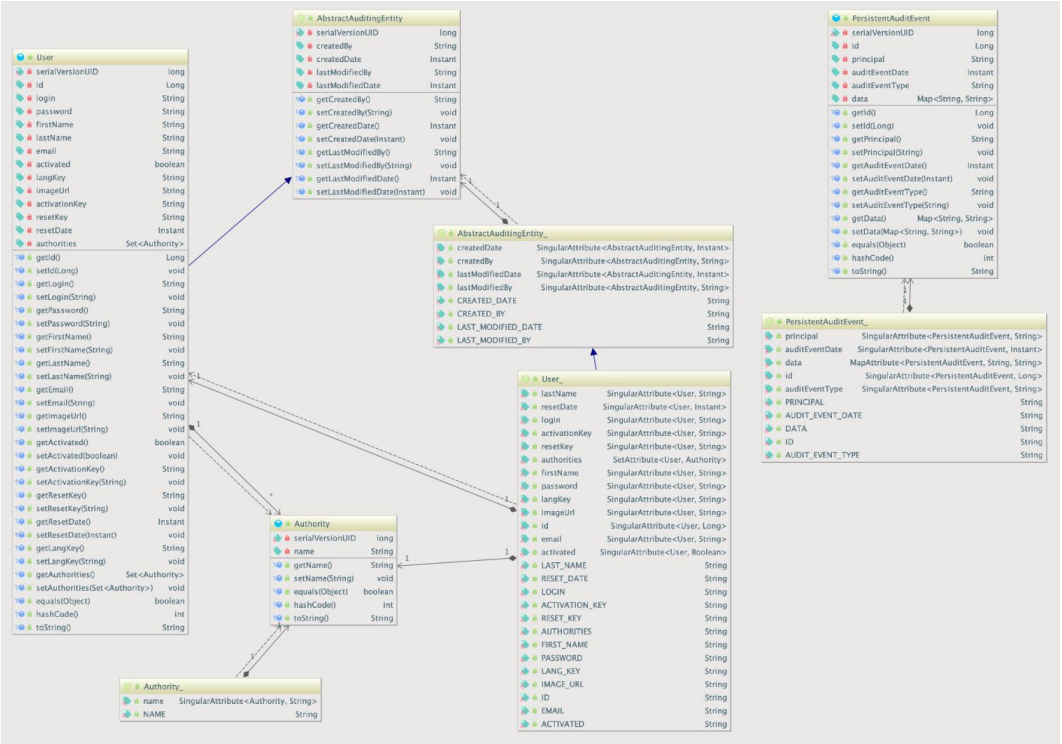

Рисунок 29 – Диаграмма классов пакета domain

Repository – пакет, содержащий классы Spring Data JPA. Этот модуль упрощает построение приложений, работающих с базами данных, и позволяет с помощью классов и конфигурационных файлов описывать сущности баз данных и работу с ними. Диаграмма классов пакета repository для проекта UAA server изображена на рисунке 30.

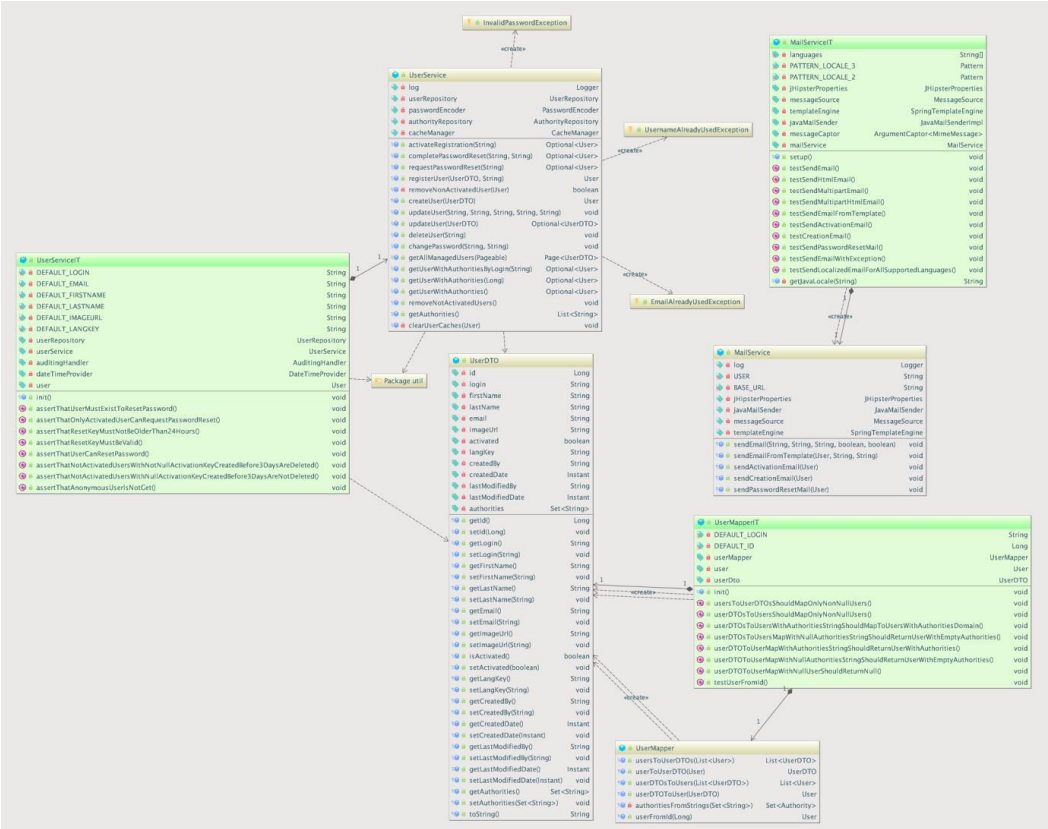

Рисунок 30 – Диаграмма классов пакета repository

Security – содержит вспомогательные классы для конфигурации в пакете config.

Service – содержит сервисы для работы с пользователем. Предоставляет такую функциональность как регистрация, авторизация, получение пользовательских данных и т.д. Диаграмма классов пакета service для проекта UAA server изображена на рисунке 31.

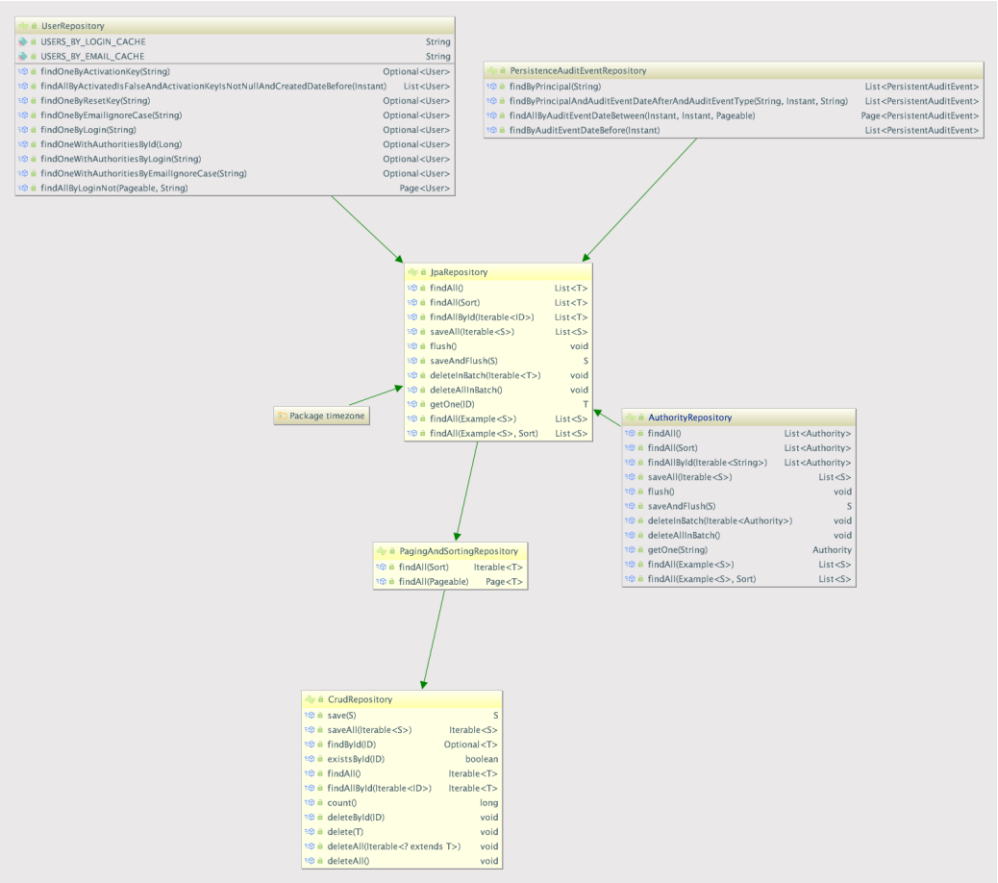

Рисунок 31 – Диаграмма классов пакета service

3.3 Registry сервер

Registry сервер это микросервис, выполняющий три основных функции:

– является Eureka сервером, выступающим в роли Discovery server. Именно благодаря Eureka приложение решает вопросы связывания, балансировки нагрузки и масштабируемости всех микросервисов;
– является сервером конфигурации Spring Configuration Server, предоставляющим конфигурационные файлы для всех микросервисов во время выполнения;
– является сервером администрирования, мониторинга и обработки услуг.

Главный класс данного проекта обозначен несколькими инструкциями. Аннотация EnableEurekaServer обозначает проект как сервер Eureka и

означает, что этот проект будет выступать как Discovery server. EnableConfigServer обозначает проект, как сервер конфигураций, то есть все микросервисы смогут считывать конфигурационные файлы из единственного хранилища, а еще существует возможность динамического изменения самих конфигурационных файлов.

3.4 Events сервис

Events сервис – это небольшой микросервис, который как раз будет предоставлять системе API для работы с мероприятиями. Для создания данного микросервиса создана модель данных для сервиса.

Основными сущностями модели являются Организация, Событие и Локация. Рассмотрим каждую из сущностей. ER-диаграмма проектируемой сущности изображена на рисунке 32.

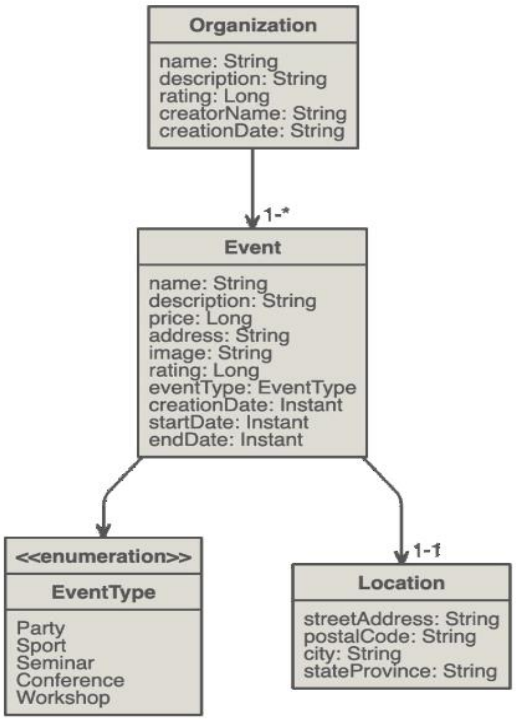

Рисунок – 32 – ER-диаграмма сервиса

Рассмотрим подробнее каждую из сущностей.

Сущность Организация имеет следующие поля.

– id – первоначальный ключ таблицы;
– name – поле, содержащее имя организации;
– description – поле, содержащее описание организации;
– rating – поле, содержащее рейтинг организации;
– creatorName – поле, содержащее название основателя организации;
– creationDate – поле, содержащее дату создания организации.

Данная сущность представляет собой компанию, проводящую определенное мероприятие.

Сущность Событие имеет следующие поля:

– id – первоначальный ключ таблицы;
– name – поле, содержащее название события;
– description – поле, содержащее описание события;
– price – поле, содержащее цену происшествия;
– address – поле, содержащее адрес события;
– image – поле, содержащее путь к картинке;
– rating – поле, содержащее оценку происшествия;
– eventType – поле, содержащее тип события;
– creationDate – поле, содержащее дату создания события;
– startDate – поле, содержащее дату начала события;
– endDate – поле, содержащее дату окончания события.

Данная сущность является основополагающей в данном микросервисе.

Последней сущностью является локация, которая хранит в себе данные о размещении события и содержит следующие поля:

– id – первичный ключ таблицы;
– streetAddress – поле, содержащее адрес улицы;
– postalCode – поле. Содержит почтовый код;
– city – поле, содержащее название города;
– stateProvince – поле, содержащее название района.

На основе этой модели данных мы можем разработать программную реализацию данной модели используя ORM (Object-relational mapping). ORM — это механизм, позволяющий обращаться и управлять объектами, не считая, как эти объекты сохраняются. Это идея возможности писать запросы в базу данных любой сложности, используя объектно-ориентированную парадигму желаемого языка программирования.

В данном случае Spring предоставляет удобный модуль для работы с ORM – Spring Data. Данный модуль позволяет посредством аннотаций определять сущности баз данных.

Как можно увидеть на рисунке, нам необходимо создать так называемый POJO класс и пометить его аннотациями @Entity и @Table, которые обозначают его как сущность и таблицу, и помогут модулю Spring Data соответствующим образом обработать данный класс и создать на его основе соответствующую таблицу в любой базе данных. Пример такого класса показан на рисунке 33.

```java
@Entity
@Table(name = "event")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Event implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "price")
    private Long price;

    @Column(name = "address")
    private String address;

    @Column(name = "image")
    private String image;

    @Column(name = "rating")
    private Long rating;

    @Enumerated(EnumType.STRING)
    @Column(name = "event_type")
    private EventType eventType;

    @Column(name = "creation_date")
    private Instant creationDate;

    @Column(name = "start_date")
    private Instant startDate;

    @Column(name = "end_date")
    private Instant endDate;

    @OneToOne(fetch = FetchType.LAZY)

    @JoinColumn(unique = true)
    private Location location;

    @ManyToOne(fetch = FetchType.LAZY)
    @JsonIgnoreProperties("events")
    private Organization organization;
```

Рисунок 33 – Класс Event

Все ORM сущности находятся в пакете domain и его общая структура имеет следующий вид (рисунок 34).

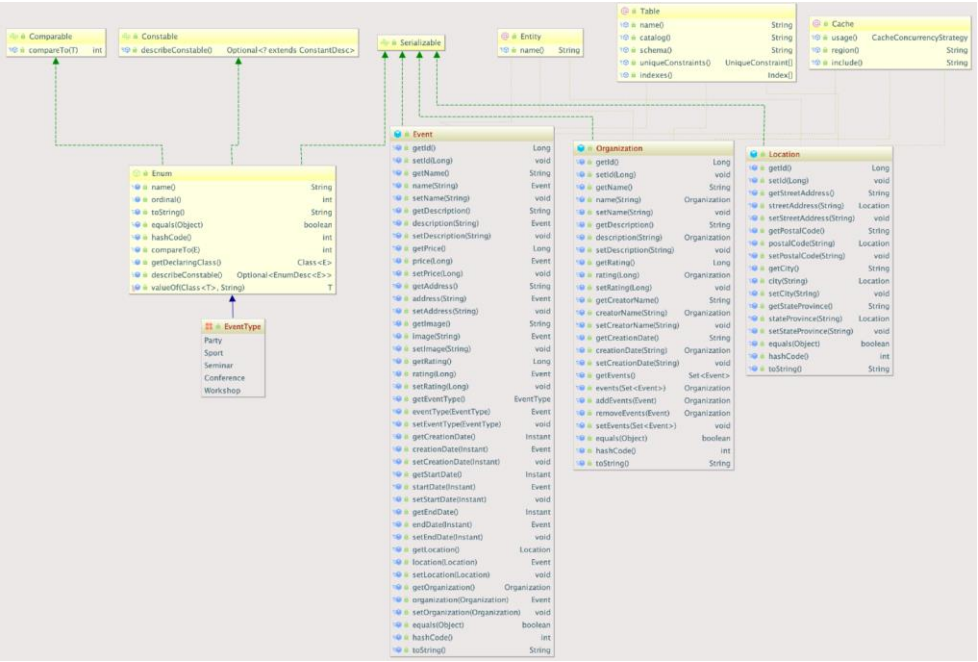

Рисунок 34 – Диаграмма классов пакета domain

Для работы с данными у нас есть модуль доступа к данным под названием repository. Для доступа к данным будем использовать Spring Data Repositories. Данный модуль позволяет создавать запросы в базы данных с помощью программного кода.

Для этого нужно воплотить интерфейс JpaRepository. После этого просто называя подходящим образом способы можно создавать запросы в базу данных. Скажем аналогом к следующему запросу в базу данных select e from Event e where e.name=?1 and e.address=?2 будет следующий программный код [30]. Пример кода изображен на рисунке 35.

```
@Repository
public interface EventRepository extends JpaRepository<Event, Long> {

    List<Event> findByNameAndAddress(final String name, final String address);

}
```

Рисунок 35 – Пример работы с интерфейсом JpaRepository

3.5 Общее описание работы системы

Диаграмма потоков данных разработанной микросервисной архитектуры представлена на рисунке 36.

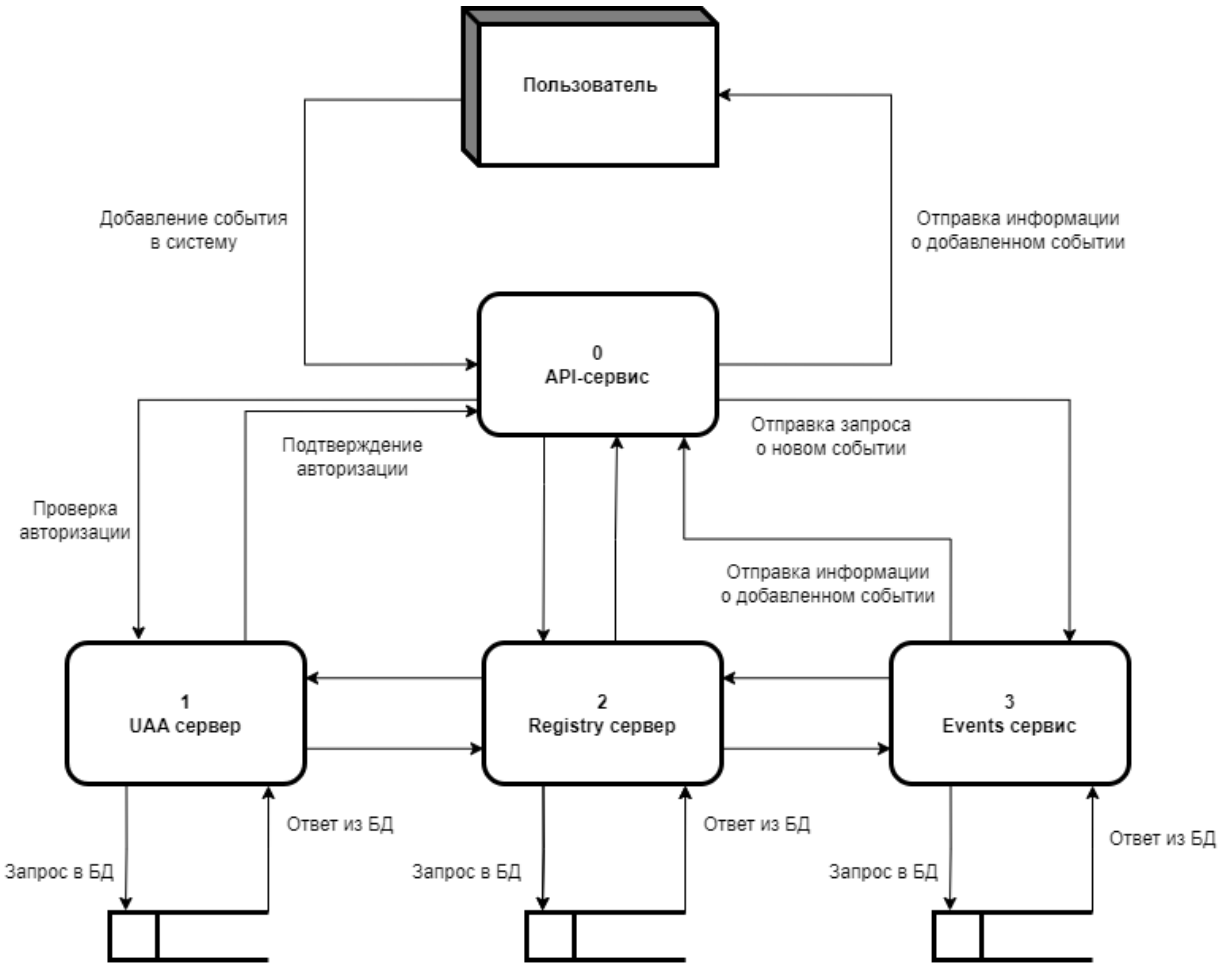

Рисунок 36 – Диаграмма потоков данных

Для запуска системы необходимо развернуть все 4 микросервиса. Первым следует развертывать Registry микросервис, ведь он является сервером конфигурации и все остальные микросервисы будут получать конфигурационные данные именно из него. Registry сервер развертывается на порте 8761. Сервер API Gateway - на порте 8080. UAA сервер развертывается на порте 9999, а микросервис Events service - на порте 8081.

Рассмотрим поближе работу с разработанной системой. Когда пользователь впервые посетит веб систему, он увидит домашнюю страницу со списком мероприятий (рисунок 37).

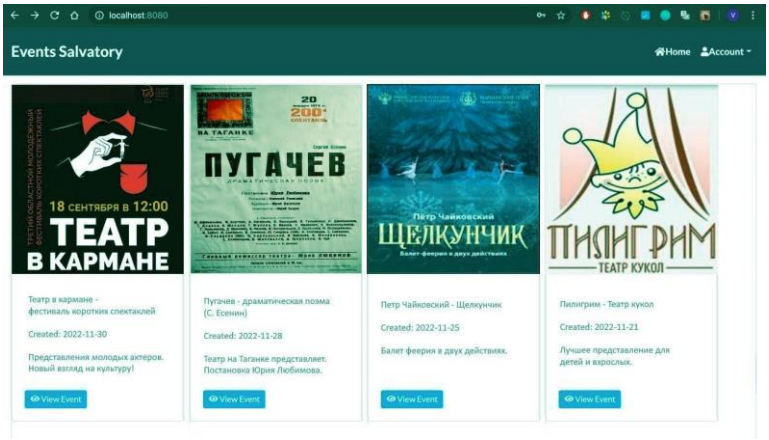

Рисунок 37 – Домашняя страница приложения

Далее пользователь может перейти зарегистрировать нового пользователя (рисунок 38).

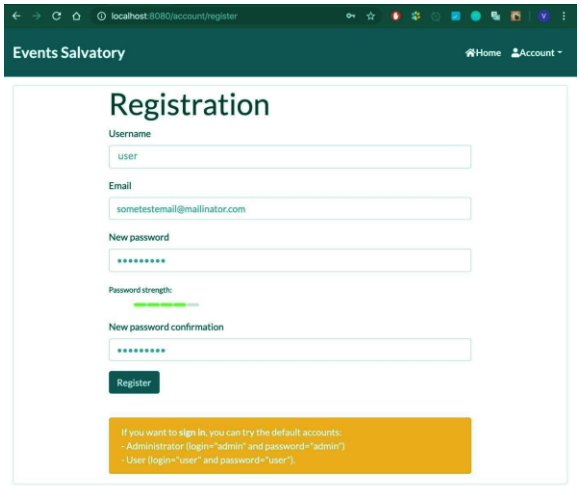

Рисунок 38 – Форма регистрации

Заполнив необходимые поля и нажав кнопку Register будет сделан POST запрос. Как можно увидеть на рисунке 39, запрос будет сделан по адресу

http://localhost:8080/services/eventsUaa/api/register. Именно здесь мы и видим, как работает API Gateway, на самом деле запрос будет сделан на UAA сервер, находящийся на порте 9999, но API Gateway с помощью Registry сервера, узнает, что существует некоторый UAA сервер, который называется eventsUaa, который предоставляет API для регистрации пользователя путем /api/register, добавит к этому всего общий суффикс /services в результате чего и получится описанный выше адрес. Соответственно, клиенту не нужно знать адрес и порт UAA сервера, а использовать порт API Gateway и путь, сгенерированный по описанному выше правилу.

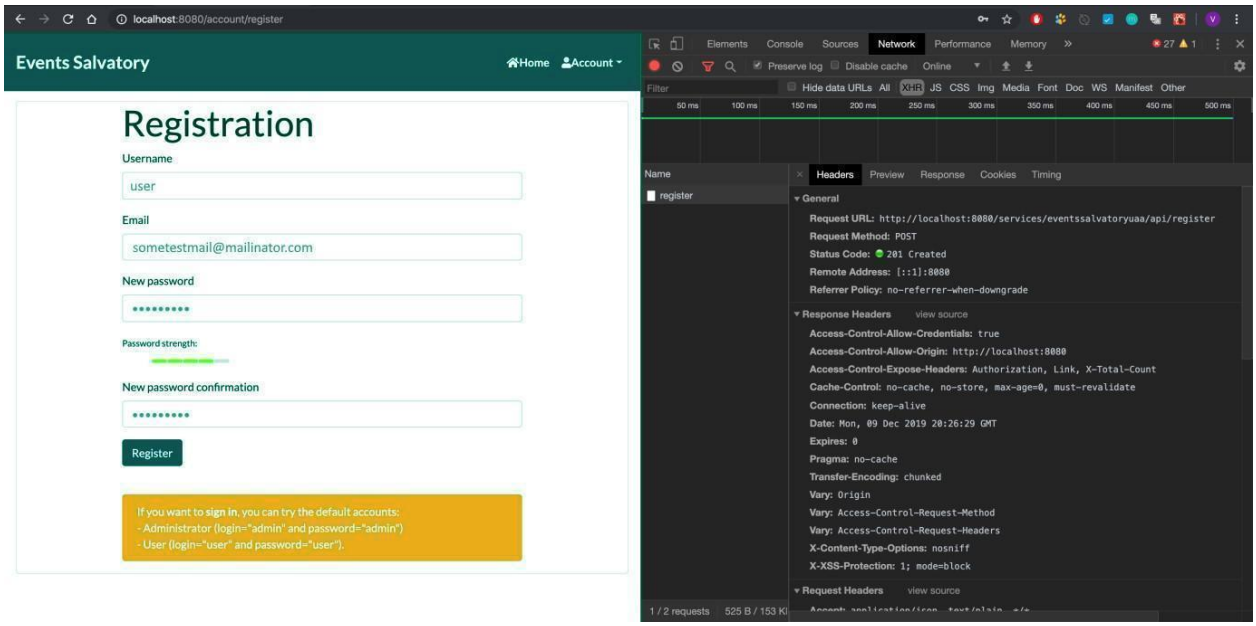

Рисунок 39 – Заполненная форма регистрации и запрос на UAA сервер

Собственно, можно выполнить запрос и направление, чтобы продемонстрировать это, используя приложение Postman, которое помогает выполнять запросы к определенным API, представленное на рисунке 40.

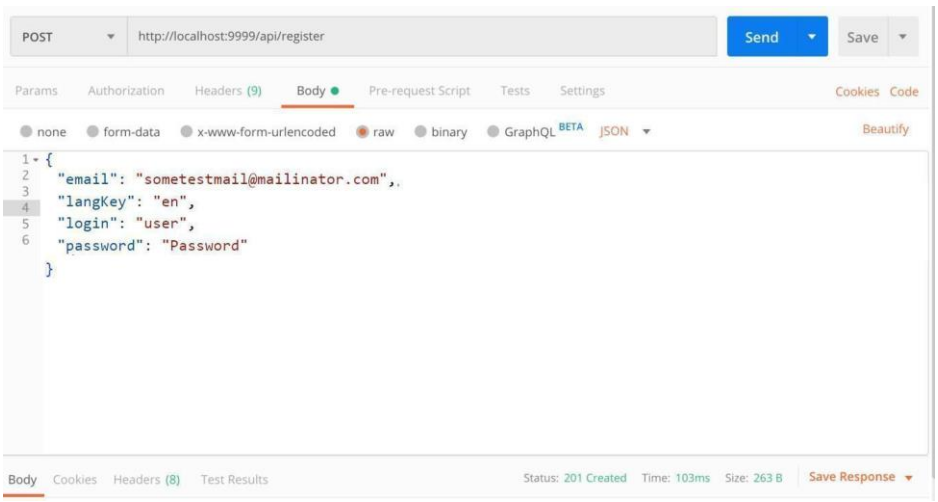

Рисунок 40 – Прямой запрос на регистрацию к UAA серверу

Postman — это API клиент, позволяющий делать запрос к разрабатываемому API. И это очень удобно, ведь для разработки и тестирования разрабатываемого API не нужно создавать клиентскую часть приложения, а достаточно всего выполнять запросы используя простой на гибкий интерфейс Postman API.

Для того чтобы выполнить запрос к UAA серверу напрямую, нам необходимо добавить в тело запроса следующие поля:

– email – email адрес пользователя;
– langKey – язык клиентской части;
– login – имя пользователя;
– password – пароль пользователя.

После этого мы можем зайти в систему как администратор и убедиться, что оба пользователя были зарегистрированы (рисунок 41).

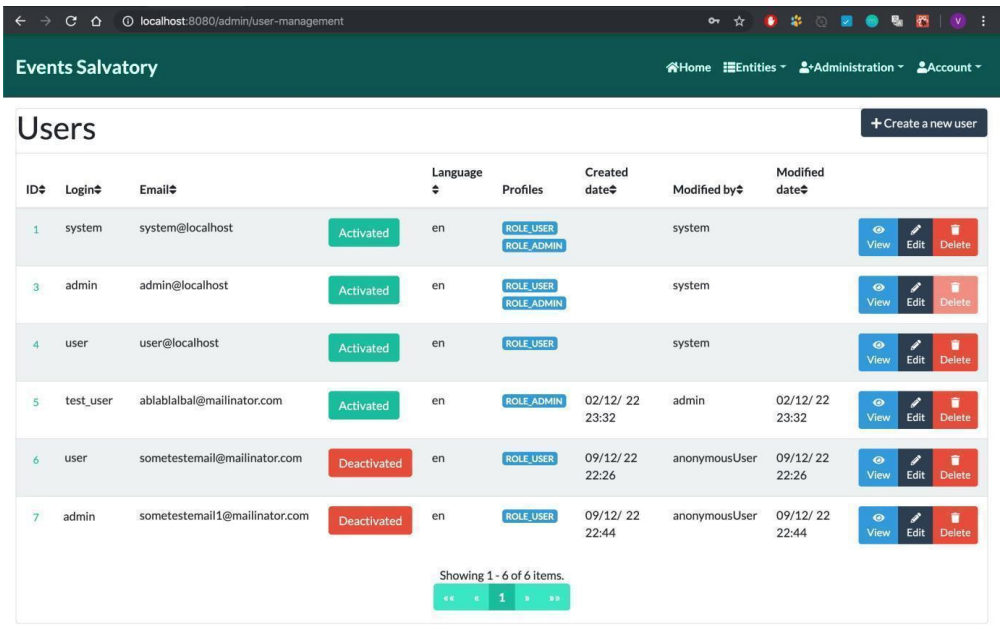

Рисунок 41 – Панель администратора для управления пользователями

После процесса регистрации рассмотрим процесс авторизации в системе. На пользовательском интерфейсе будет доступна кнопка Account. После того, как пользователь нажмет на нее, ему будет предоставлен выбор, зарегистрировать нового пользователя или войти в систему уже существующим. На этот раз необходимо избрать войти существующим.

После этих действий пользователю откроется форма логина (рисунок 42), где ему нужно будет ввести свое имя и пароль. После того как пользователь ввел свои данные в форму логина и нажал кнопку login, будет сделан запрос по уже описанной выше схеме. Если пользователь не прошел аутентификацию (заданная учетная запись не существует или пароль неверен), то пользователю будет показано окно с ошибкой.

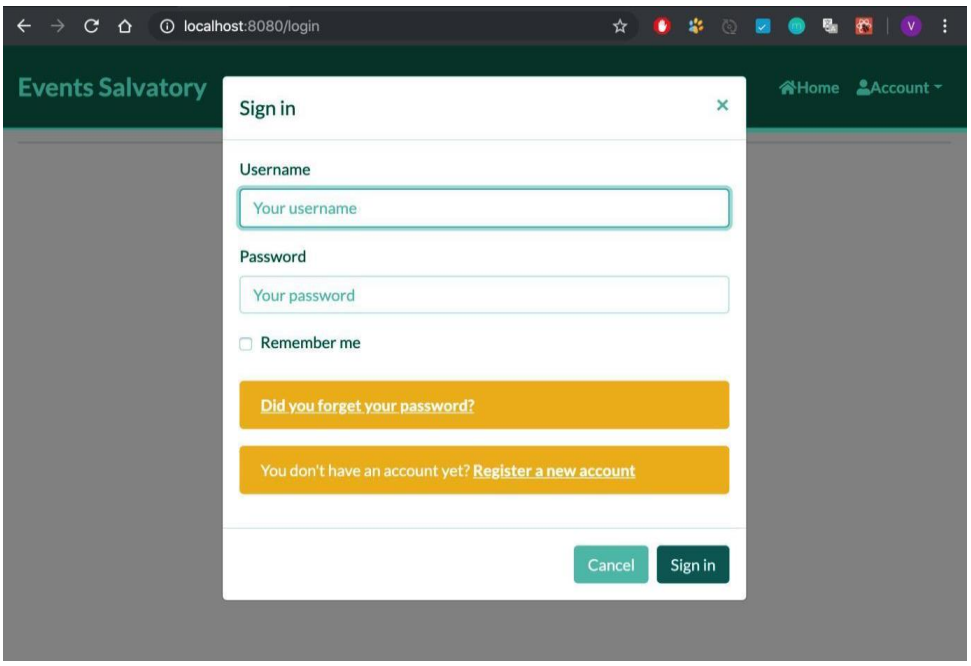

Рисунок 42 – Форма входа

Если пользователь успешно прошел аутентификацию, то UAA сервер сгенерирует JWT токен (рисунок 43) для пользователя и сохранит его в хранилище cookie. JWT токен будет хранить данные о пользователе, такие как его имя и права доступа.

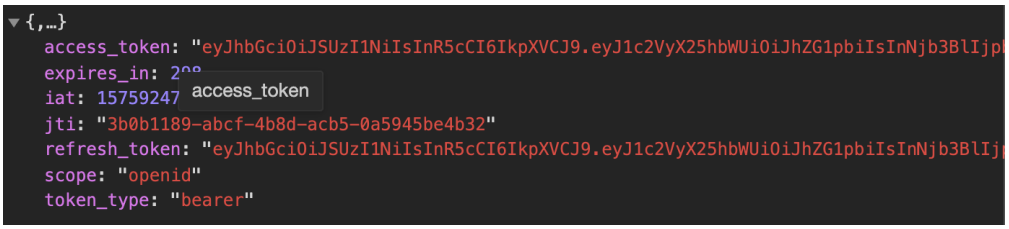

Рисунок 43 – Ответ на запрос авторизации, содержащий JWT токен

Далее, чтобы иметь возможность доступа к ресурсам, таким как список мероприятий или организаций, в запросе нужно передавать данный токен в запросе. На сервере он будет проходить валидацию, декодироваться и при наличии соответствующих прав, будет возвращен ответ клиенту, сделавшему

запрос. В качестве примера приведем запрос, который должен вернуть список всех событий (рисунок 44).

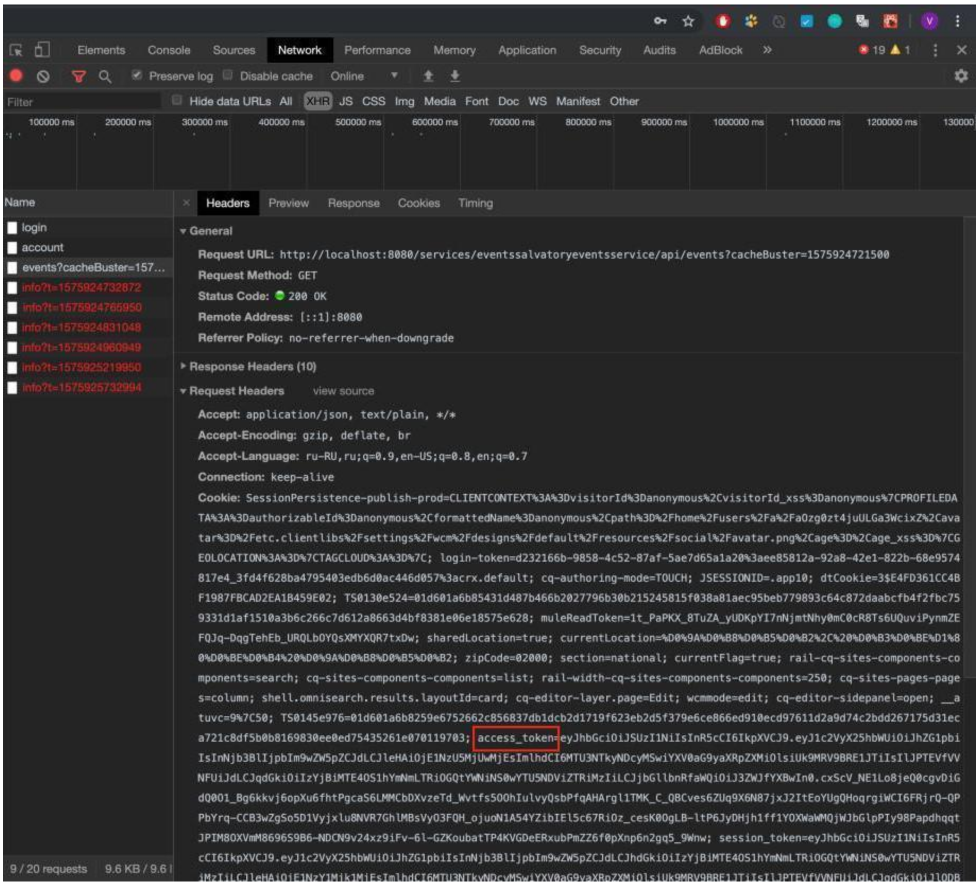

Рисунок 44 – Домашняя страница системы и запрос в Events микросервис

В ответ на запрос получим JSON массив с событиями (рисунок 45). JSON массив представляет собой упорядоченный набор объектов.

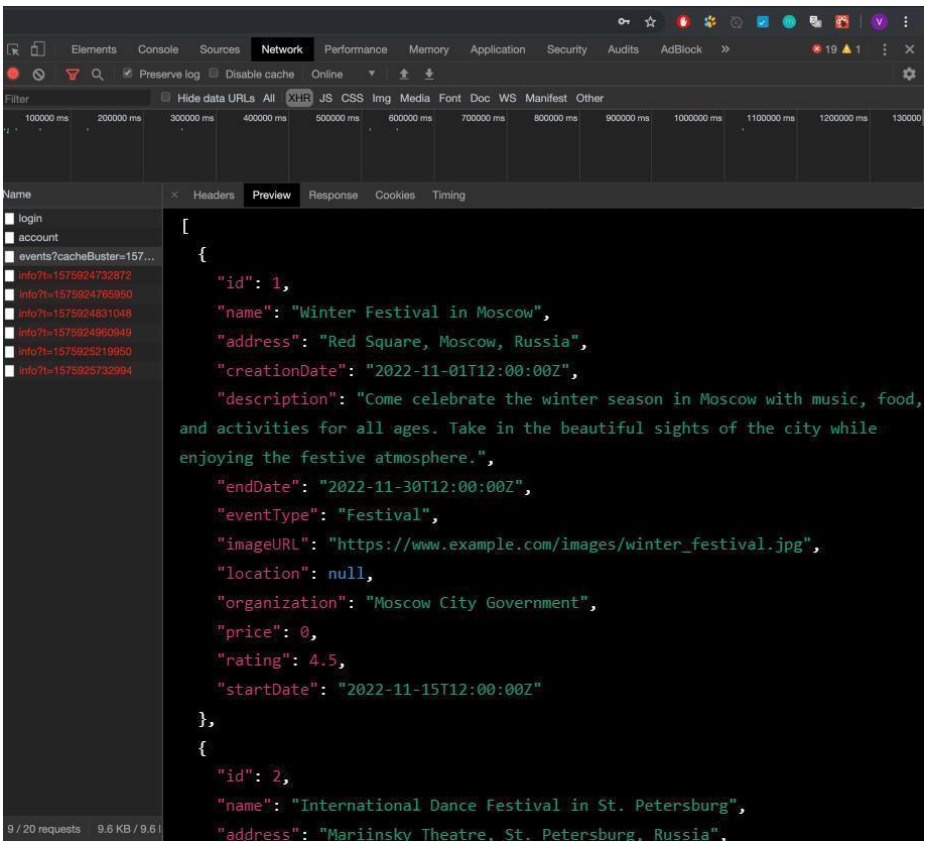

Рисунок 45 – Домашняя страница и ответ Events микросервиса по запросу

Администратор также имеет возможность просматривать (рисунок 46) и модифицировать существующие мероприятия (рисунок 47), а также создавать новые.

Рисунок 46 – Страница события

Рисунок 47 – Страница создания или модификации мероприятия

Registry сервер также имеет встроенный графический интерфейс, и мы можем с его помощью посмотреть, например, список подключенных к серверу микросервисов (рисунок 48).

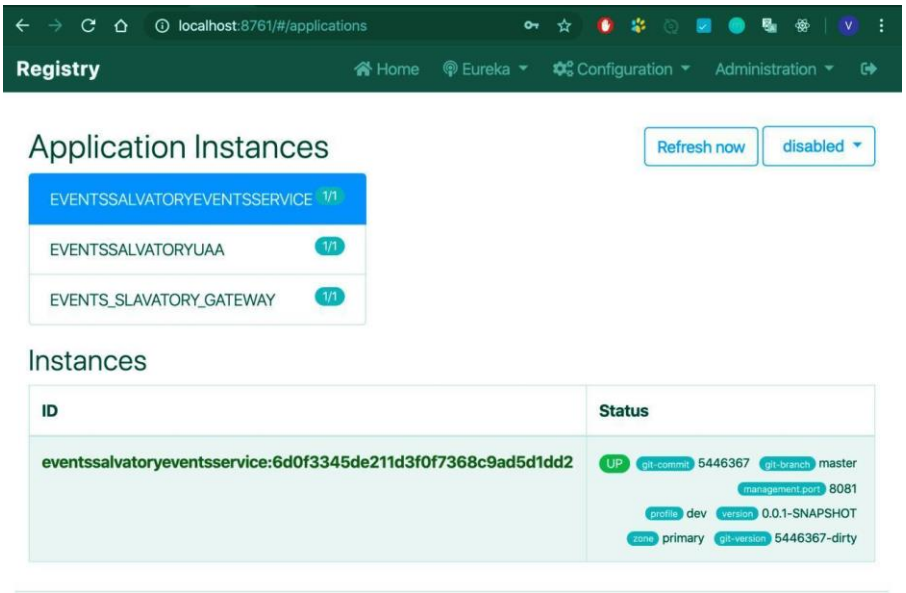

Рисунок 48 – Интерфейс Registry сервера для просмотра подключенных микросервисов

На рисунке можем увидеть, что в данный момент времени Registry сервер работает с тремя микросервисами.

3.6 Анализ эффективности разработанного решения

Чтобы продемонстрировать преимущества микросервисной архитектуры над монолитной, мы реализовали две тестовые системы и провели реальное сравнение веб-приложения, построенного на микросервисах, с аналогичным приложением, построенным на монолитной архитектуре.

Веб-приложение, построенное на микросервисах, имеет сервер шлюза API, сервер UAA и сервер реестра, а также ряд независимых микросервисов, таких как сервис Events. Каждый микросервис отвечает за определенную функцию, а приложение в целом было способно обрабатывать большое количество пользователей и трафика благодаря своей масштабируемости и отказоустойчивости.

В отличие от этого, монолитное приложение было построено как единая большая кодовая база со всеми функциями в одном приложении. Хотя это и упростило первоначальную разработку, это также затруднило масштабирование и поддержку приложения по мере роста его размера и сложности.

Тестирование проводилось на локальном окружении с замерами метрик производительности, используя программное обеспечение Netdata, что предоставляет быструю интеграцию и удобные дашборды с множеством показателей. Процесс наблюдения за двумя системами длился 1 неделю. Для создания нагрузки на систему была применена утилита имитации посещаемости Apache JMeter с равномерным распределением трафика по времени и разными пиками нагрузки. Был установлен объем трафика 200 тыс. запросов равномерно распределенный на весь срок тестирования с максимальной пиковой нагрузкой в 20 тыс. запросов одновременно. В случае

отказа системы выполнялась ручная перезагрузка и рестарт сервиса. Для обоих приложений были установлены одинаковые условия и ресурсы, чтобы тестирование было максимально объективным.

Метрики, которые собирались в процессе наблюдения:

– нагрузка на процессор (CPU);
– объем использованной оперативной памяти (RAM);
– число успешно обработанных запросов;
– число неудачных запросов к системе;
– время простоя системы, когда она не могла обрабатывать запросы;
– пиковая нагрузка на систему - максимальное число запросов перед отказом системы, которое приложение смогло обработать.

Собранные показатели статистики по монолитному приложению были внесены в таблицу 3.

Таблица 3 – Собранная статистика работы монолитного приложения за указанный период

День	Средн. CPU, %	Средн. Memory, %	Успешных запросов	Неудачных запросов	Простой системы, мин.	Макс. число запросов перед отказом
1	78	86	14290	14930	29	9544
2	71	79	13849	13482	32	9391
3	75	85	14938	15731	35	9884
4	74	84	14831	14839	30	9102
5	76	81	13483	14470	25	9938
6	73	79	13382	14483	29	9543
7	76	84	13848	13444	28	9891
Итого:	-	-	98621	101379	208	-
Среднее:	75	83	14089	14483	30	9613

Собранные показатели статистики по микросервисному приложению были внесены в таблицу 4.

Таблица 4 – Собранная статистика работы микросервисного приложения за указанный период

День	Средн. CPU, %	Средн. RAM, %	Успешных запросов	Неудачных запросов	Простой системы, мин.	Макс. число запросов перед отказом
1	44	52	22300	7391	15	16459
2	43	55	20844	7382	14	14688
3	41	51	19388	6990	9	15441
4	48	47	22013	6872	12	15970
5	51	55	20932	7644	11	14983
6	45	51	22561	7323	16	13081
7	49	46	21012	7348	10	15733
Итого:	-	-	149050	50950	87	-
Среднее:	46	51	21293	7279	12	15194

Список метрик, которые были использованы в процессе расчета эффективности на основании полученной статистики:

– время простоя, которое относится ко времени, когда система не обрабатывала никаких запросов по причине недоступности;
– частота отказов, которая показывает процент запросов, которые не были обработаны от общего числа запросов к системе;
– использование ресурсов, что описывает процент доступных ресурсов, используемых системой;
– общее количество успешно обработанных запросов за весь период тестирования и наблюдения;

– пиковая нагрузка на систему, под которой понимается максимальное количество запросов, которое система может обработать в любой момент времени.

Результаты, полученные на основании собранной аналитики, представлены в таблице 5.

Таблица 5 – Сравнение производительности микросервисной архитектуры и монолитной архитектуры в нашем реальном примере использования

Метрики производительности	Архитектура микросервисов	Монолитная архитектура	Улучшение
Время простоя	~ 87 мин.	~ 208 мин	На 58.2% меньше простоев системы
Коэффициент отказов	1.00%	4.00%	На 75% меньше отказов системы
Использование ресурсов	48.00%	79.00%	На 39.24% эффективнее использование ресурсов
Суммарное число успешно обработанных запросов за весь период	Способность обслуживать суммарно 149 050 запросов без простоев и сбоев	Способность обслуживать суммарно 98 621 запросов с периодическими простоями и отказами	Улучшение на 55.10% и отсутствие простоев
Пиковая нагрузка	15 194 запросов одновременно	9613 запросов одновременно	Улучшение на 77.78%

Результаты исследования показали, что микросервисная архитектура (MSA) на тестируемых приложениями показала себя эффективнее, чем монолитная архитектура по нескольким ключевым показателям:

– MSA имела на 58,2% меньше времени простоя системы по сравнению с монолитной архитектурой, что указывает на более надежную и отказоустойчивую систему.

– Уровень отказов для архитектуры микросервисов также был значительно ниже - 1,00% по сравнению с 4,00% для монолитной архитектуры, что указывает на лучшую изоляцию и восстановление после сбоев в архитектуре микросервисов.
– С точки зрения использования ресурсов, микросервисная архитектура была на 39,24% эффективнее в использовании ресурсов по сравнению с монолитной. Это можно объяснить тем, что каждый микросервис в архитектуре микросервисов может масштабироваться независимо в зависимости от спроса, в то время как в монолитной архитектуре масштабирование осуществляется на уровне приложения.
– MSA смогла успешно обработать 149 050 запросов без простоев и сбоев, в то время как монолитная архитектура смогла обслужить только 98 621 запрос с периодическими простоями и сбоями, что говорит о том, что архитектура микросервисов была более надежной и эффективной.
– Архитектура микросервисов также смогла справиться со значительно более высокой пиковой нагрузкой по сравнению с монолитной архитектурой, с улучшением на 77,78%. Это можно объяснить тем, что архитектура микросервисов может масштабировать отдельные сервисы в зависимости от спроса, в то время как в монолитной архитектуре для обработки возросшего трафика необходимо масштабировать все приложение.

В целом, результаты тестирования демонстрируют преимущества архитектуры микросервисов перед монолитной архитектурой с точки зрения масштабируемости, отказоустойчивости и использования ресурсов. MSA позволяет разрабатывать и развертывать независимые сервисы по отдельности, что облегчает обслуживание и масштабируемость. Кроме того, обеспечивает отказоустойчивость, при которой сбой в одном сервисе не обязательно приведет к отказу всего приложения.

В ходе тестирования мы обнаружили, что микросервисная архитектура значительно превосходит монолитную архитектуру по нескольким ключевым параметрам. Во-первых, обеспечивает легкую масштабируемость и отказоустойчивость, позволяя приложению обрабатывать большое количество пользователей и трафика без простоев и сбоев.

Кроме того, MSA была более гибкой и адаптируемой, чем монолитная, позволяя использовать различные технологии и базы данных для создания новых сервисов. Это облегчило добавление новых функций в приложение по мере необходимости и уменьшило сложность кодовой базы.

Кроме того, микросервисная архитектура была более экономичной и эффективной с точки зрения использования ресурсов, поскольку развертывать и масштабировать нужно было только необходимые сервисы. Это привело к снижению затрат и повышению производительности по сравнению с монолитной архитектурой.

В целом, наше сравнение продемонстрировало явные преимущества микросервисной архитектуры перед монолитной, особенно в плане масштабируемости, гибкости, отказоустойчивости и использования ресурсов. Микросервисная архитектура представляет собой более надежное и эффективное решение для разработки сложных, масштабируемых веб-приложений.

3.7 Анализ разработанного решения

Разработанная веб система представляет собой систему построенную на основе микросервисов (рисунок 17). К ее преимуществам можно отнести следующие особенности:

– легкая масштабируемость. Разработанная система имеет огромный потенциал в масштабировании. Разработанный прототип представляет собой необходимый каркас для работы с неограниченным количеством микросервисов, он выполняет такие

ключевые функции веб системы как авторизация и аутентификация пользователя в системе, работа с конфигурационными файлами, балансировка нагрузки, нахождение и связывание независимых сервисов и т.д. Поэтому для создания и добавления нового микросервиса в систему требуется минимум дополнительных усилий;

– гибкость. Разрабатываемая система очень гибкая и позволяет использовать различные технологии и базы данных для создания новых сервисов;
– удобство использования. Разработанные микросервисы являются автономными, то есть их можно использовать отдельно как сервисы, предоставляющие открытые API. Скажем UAA сервер возможно использовать и в других проектах, просто необходимо делать запросы к определенным в сервисе конечным точкам и сервер все так же будет генерировать JWT токен и предоставлять и контролировать информацию о пользователе;
– отказоустойчивость и беспроблемное развертывание. Разработанное решение достаточно отказоустойчивое и остановка работы Events сервиса не приведет к отказу всей системы и пользовательский интерфейс, как и авторизация и аутентификация для примера, все еще будет работать исправно. Также процесс развертывания каждого сервиса достаточно прост и удобен и не повлияет на работу всей системы, к тому же при внесении изменений в определенный сервис, нет необходимости перезагружать всю систему, а достаточно лишь развернуть необходимый микросервис.

Подводя итоги, стоит отметить, что разработанное решение является готовым шаблоном для разработки новых систем на основе микросервисов. API Gateway сервер, UAA сервер и Registry сервер является полноценным и необходимым минимумом для разработки новых микросервисных систем. Необходимо только добавлять новые микросервисы с новой логикой,

принимая за шаблон Events сервис. Все перечисленные плюсы делают бескомпромиссным выбор микросервисной архитектуры из монолитной и сервисно-ориентированной, не имеющих описанных преимуществ, а сам прототип помогает решить главный недостаток микросервисной архитектуры – ее сложность, что позволит разработчикам сразу сосредоточиться на разработке бизнес-логики и избежать неприятностей на старте.

Выводы по главе 3

Данная глава диссертации посвящена реализации микросервисной архитектуры для монолитного веб-приложения. Мы делаем обзор структуры разрабатываемой микросервисной архитектуры и описываем компоненты, которые были реализованы в проекте.

Компоненты микросервисной архитектуры включают в себя службу шлюза API, сервер UAA (учетные записи и аутентификация пользователей), сервер реестра и несколько независимых микросервисов. Служба API Gateway служит точкой входа для всех клиентских запросов и направляет их к соответствующему микросервису. Сервер UAA отвечает за аутентификацию и авторизацию пользователей. Сервер реестра используется для регистрации и управления микросервисами. Независимые микросервисы отвечают за конкретные функции, например, служба Events, которая занимается управлением событиями.

Чтобы продемонстрировать преимущества микросервисной архитектуры, мы реализовали две простые тестовые системы - одну на основе микросервисов и другую на основе монолитной архитектуры. Затем провели сравнение двух систем, чтобы определить, какая из них работает лучше.

Тестирование проводилось в локальной среде с использованием программного обеспечения Netdata для измерения таких показателей производительности, как загрузка процессора, объем используемой оперативной памяти, количество успешно обработанных запросов, количество

неуспешных запросов, время простоя системы и пиковая нагрузка на систему. Утилита моделирования посещаемости Apache JMeter использовалась для создания нагрузки на систему с равномерно распределенным по времени трафиком и различными пиками нагрузки. Объем трафика в 200 тысяч запросов был задан равномерно распределенным на весь период тестирования с максимальной пиковой нагрузкой в 20 тысяч запросов за один раз.

Статистика для монолитного приложения и микросервисного приложения собиралась в течение 1 недели и заносилась в таблицу 3 и таблицу 4 соответственно. На основе полученных статистических данных рассчитали эффективность двух систем по таким показателям, как время простоя, частота отказов, использование ресурсов, общее количество успешно обработанных запросов и пиковая нагрузка.

Результаты показали, что микросервисная архитектура превзошла монолитную архитектуру по эффективности. Микросервисная архитектура имела на 58,2% меньше времени простоя, на 75% меньше отказов системы и на 39,24% более эффективно использовала ресурсы. Кроме того, микросервисная архитектура смогла обслужить в общей сложности 149 050 запросов без простоев и сбоев, в то время как монолитная архитектура смогла обслужить только 98 621 запрос с периодическими простоями и сбоями - улучшение на 55,10%.

В целом, глава содержит подробный анализ эффективности разработанной микросервисной архитектуры и демонстрирует преимущества использования микросервисов по сравнению с монолитной архитектурой в реальном примере использования.

Глава 4 Разработка бизнес-плана проекта

4.1 Описание идеи проекта

В данном разделе будет представлена разработка проекта по теме работы, а также будет проведен анализ возможного внедрения проекта.

Идея проекта – разработать архитектуру веб-приложения по поиску мероприятий на основе микросервисной архитектуры. Основной целью является разработка эффективной архитектуры веб-приложения, которая будет эффективна и устойчива при дальнейшем развитии приложения, а также привлекательна для клиента, что могло бы привести к заказу на разработку подобной системы.

Отдельно рассмотрим направления применения идеи и ее полезность для пользователя в таблице 6 и заказчика в таблице 7.

Таблица 6 – Описание идеи проекта

Содержание идеи	Направление применения	Удобство пользователя
Создать систему поиска мероприятий с использованием микросервисной архитектуры, которая позволит пользователю быстро и удобно искать разнообразные мероприятия	Веб-приложение для поиска мероприятий	Отсутствие необходимости тратить много времени на поиск необходимой информации на большом количестве сторонних сервисов
		Удобство. Вся информация о мероприятиях находится в одном месте и имеет удобный пользовательский интерфейс.
		Интеграция с платежными системами позволяет не только просматривать информацию о мероприятиях, но и сразу приобрести билеты при необходимости

Таблица 7 – Описание идеи проекта для заказчика

Содержание идеи	Направление применения	Удобства для заказчика
Разработать систему, которая будет гибкой и легко масштабируемой. Созданная система должна иметь возможность работать с разнообразными стеками технологий.	Разработка новой высоконагруженной системы	Простота в масштабировании системы при необходимости
		Гибкость для новых разработчиков
		Возможность системы быстро адаптироваться под новые технологии/фреймворки

После обзора всех конкурентных технологий было обнаружено, что главными конкурентами являются SOA и Монолитные приложения. Список сильных и слабых сторон разрабатываемой архитектуры позволит лучше понять ее место по сравнению с конкурентами. Определение сильных и слабых сторон проекта приведено в таблице 8.

Таблица 8 – Определение сильных, слабых и нейтральных сторон идеи проекта

Технико-экономические характеристики идеи	Потенциальные товары/концепции конкурентов			W	N	S
	Мой проект (MSA)	Конкурент 1 SOA	Конкурент 2 Монолиты			
Дизайн архитектуры	Сервисы построены как отдельные, независимые компоненты целой системы, ориентированные на бизнес-нужды.	Сервисы могут быть как небольшого размера, так и быть большими сервисами, решающими множество бизнес-задач	Монолитное приложение является одним крупным сервисом и довольно часто разрастается до таких размеров, что понимать его или масштабировать становится тяжело		+	
Практическая полезность	Сервисы предоставляют определенный API, что дает доступ к реализации тех бизнес-потребностей, которые реализует каждый сервис.	К сервисам можно получить доступ через стандартный протокол передачи данных как SOAP, через SOA Bus.	Монолитное приложение является единственным сервисом, поэтому переиспользование функциональны частей монолитного приложения является лимитированным.		+	

Продолжение таблицы 8

Технико-экономические характеристики идеи	Потенциальные товары/концепции конкурентов			W	N	S
	Мой проект (MSA)	Конкурент 1 SOA	Конкурент 2 Монолиты			
Легкая масштабируемость	Сервисы существуют как независимые компоненты для развертывания, поэтому легко изменять и масштабировать их без значительного влияния на систему в целом.	Сервисы имеют определенную связанность, что приводит к проблемам масштабирования.	Поскольку монолит — это цельное приложение, его масштабирование может быть сложным вызовом.	-	-	+
Гибкость в выборе технологий	Поскольку каждый сервис является автономным, разработчики могут использовать различные стеки технологий для различных микросервисов.	Возможность использовать разные технологии присутствует, но наличие уровня агрегации SOA Bus накладывает определенные ограничения.	Монолитное приложение должно использовать только единый стек технологий.	-	-	+
Гибкость в построении команд разработчиков	Существует возможность создания различных команд, которые будут работать над разными сервисами и разными технологиями, что придает большую гибкость в найме.	+	-	-	-	+
Отказоустойчивость системы	Каждый компонент является автономным сервисом, поэтому его отказ не приведет к отказу всей системы. Тем более, что чаще всего разворачивается несколько экземпляров сервиса, поэтому при поломке одного система перераспределяет нагрузку по остальным сервисам того же типа.	Средняя	Низкая. Поломка любого компонента, скорее всего, повлечет за собой сбой всей системы.	-	-	+

Продолжение таблицы 8

Технико-экономические характеристики идеи	Потенциальные товары/концепции конкурентов			W	N	S
	Мой проект (MSA)	Конкурент 1 SOA	Конкурент 2 Монолиты			
Простота освоения	Разработка на основе микросервисной архитектуры является довольно трудным процессом, ведь нужно настроить и разработать огромное количество сервисов, но если предположить, что отдельные команды работают над отдельными сервисами, то небольшие компоненты системы просты для понимания, тогда как система в целом, все же достаточно тяжелая.	Средняя	Средняя или простая	-	+	-
Ремонтопригодность	Поскольку каждый сервис автономный, то поломка отдельного сервиса не сломает систему в целом. Также есть возможность отдельно изменять сервис, не сильно влияя на систему в целом.	Средняя	Средняя	-	-	+
Защищенность	Защита по протоколу Oauth2 и с помощью JWT токена.	Защита по протоколу Oauth2 и с помощью JWT токена.	Защита по протоколу Oauth2 и с помощью JWT токена.	-	+	-

Существенными преимуществами моего решения по сравнению с конкурентными являются легкая масштабируемость архитектуры проекта при необходимости, гибкость в выборе технологий, отказоустойчивость и ремонтопригодность

4.2 Технологический аудит проекта

Далее проведем технологический аудит проекта, в котором рассмотрим какие технологии будут использоваться при разработке данного проекта. Аудит служит для того, чтобы проанализировать список избранных

технологий и заключить целесообразность их использования. Результаты представлены в таблице 9.

Таблица 9 – Технологический аудит проекта

Содержание идеи	Технологии ее реализации	Наличие технологий	Доступность технологий
Система поиска мероприятий построена на основе микросервисной архитектуры	Spring stack Netflix OSS stack	Технологии доступны и хорошо работают вместе. Необходимо правильно их объединить между собой	Технологии доступны
	Spring Boot	+	+
	Spring Cloud	+	+
	Spring Data	+	+
	Spring Cloud Security	+	+
	Netflix Eureka	+ Открытое ПО	+
	Netflix Zuul	+ Открытое ПО	+
	Netflix Hystrix	+ Открытое ПО	+
	Netflix Ribbon	+ Открытое ПО	+
	Maven	+	+
Выбранные технологии: Spring Framework stack, Netflix OSS stack.			

Технологический аудит проекта показал, что для разработки приложения на основе микросервисной архитектуры достаточно привлекателен язык Java и его фреймворк Spring, который имеет много модулей для разработки именно микросервисов и интеграцию с библиотеками Netflix OSS.

4.3 Анализ рыночных возможностей запуска

Перед выходом на рынок нужно проанализировать будет ли спрос на данный проект. К тому же, существует достаточно большое количество угроз на рынке для успешного запуска любого проекта. Проведем предварительную характеристику рынка в таблице 10.

Таблица 10 – Предварительная характеристика потенциального рынка проекта

Показатели состояния рынка (наименование)	Характеристика
Количество главных игроков, ед.	Множество
Общий объем продаж, руб.	-
Динамика рынка (качественная оценка)	Растет
Наличие ограничений для входа (указать характер ограничений)	Большое количество систем, уже успешно работающих в конкретных сферах. Сложность реализации на старте.
Специфические требования к стандартизации и сертификации	Отсутствуют
Средняя норма рентабельности в отрасли (или по рынку), %	~80%

После анализа потенциального рынка становится ясно, что на рынке существует большое количество конкурентов, предлагающих разные решения для разработки проектов. Поэтому нужно хорошо понять, кто является

потенциальным клиентом приложения и определить целевую группу. Анализ потенциальных клиентов проекта представлен в таблице 11.

Таблица 11 – Характеристика потенциальных клиентов проекта

Потребность, формирующая рынок	Целевая аудитория (целевые рынки)	Различия в поведении разных целевых групп клиентов	Требования потребителей к товару
Потребность в легко масштабируемой и максимально гибкой системе, что позволит заказчику быстро реагировать на изменения.	Любые учреждения/ бизнесы/рынки	Система должна быть легко масштабируемой и гибкой. Система должна давать возможность разрабатывать ее на различных технологиях, быть надежной, отказоустойчивой и защищенной. Порог входа для новых разработчиков должен быть достаточно низким.	Система должна предоставлять привлекательный и удобный клиентский интерфейс. Быть удобной и быстрой.

После анализа потенциальных клиентов проекта проведем анализ факторов угроз и возможностей в таблицах 12 и 13.

Таблица 12 – Фактор угроз

Фактор	Содержание угрозы	Возможная реакция компании
Сложность реализации	Микросервисы хотя и предоставляют множество преимуществ, но это вытекает в достаточно большую сложность при разработке	Упрощение сложности системы путем комбинации и соединения сервисов в большие.

Продолжение таблицы 12

Фактор	Содержание угрозы	Возможная реакция компании
Отсутствие интереса к новым заказчикам	Некоторые заказчики могут решить, что система построенная на основе микросервисной не отвечает их требованиям	Улучшение процесса разработки системы и повышение качества разрабатываемых решений путем постоянного взаимодействия с заказчиками
Смена трендов	Микросервисная архитектура может устареть	Гибкость архитектурного подхода позволяет достаточно просто адаптировать систему к новым технологиям
Критические ошибки	Выявление ошибок в работе системы	Своевременный выпуск обновлений, предоставление исчерпывающей консультации службой поддержки

Таблица 13 – Фактор возможностей

Фактор	Содержание возможности	Возможная реакция компании
Заинтересованность инвесторов	Возможность получить инвестиции	Увеличение штата разработчиков
Интеграция с другими сервисами и платформами	Расширение потенциального количества пользователей и рекламодателей. Также гибкость в выборе технологии при разработке	Увеличение количества персонала, работающего над интеграционными вопросами
Сотрудничество с другими компаниями	Сочетание данных с различных сервисов поиска мероприятий с помощью открытых API	Расширение функционала системы
Сотрудничество с физическими лицами	Привлечение пользователей для популяризации системы	Повышение уровня удовлетворенности пользователей

Далее проведем анализ конкуренции на рынке в таблице 14.

Таблица 14 – Ступенчатый анализ конкуренции на рынке

Особенности конкурентной среды	В чем состоит данная характеристика	Воздействие на деятельность предприятия (возможны действия компании, чтобы быть конкурентоспособной)
Указать тип конкуренции – чистая	Большое количество независимых игроков в отдельных сферах на рынке	Предложение собственного подхода к разработке системы, ее упрощение, что позволит ускорить разработку и снизить затраты
По уровню конкурентной борьбы - национальный	Основными конкурентами системы являются веб-системы поиска мероприятий в рамках страны.	Предложение лучшей системы за меньшие деньги.
По отраслевому признаку – внутриотраслевая	Сотрудничество с предприятиями и учреждениями, работа которых связана с событиями	Предложить упрощенный процесс интеграции с сервисами других компаний
Конкуренция по видам товаров: - товарно-видовая.	Большое количество существующих систем в отдельных отраслях	Расширение функционала системы для решения потребностей каждой конкретной компании
По характеру конкурентных преимуществ: ценовая.	Заказчик платит за разработку и использование системы	Ориентация на потребность заказчиков

Далее следует провести анализ конкуренции в области методом пяти сил Майкла Портера, что показано в таблице 15.

Таблица 15 – Анализ конкуренции в отрасли по М. Портеру

	Прямые конкуренты	Потенциальные конкуренты	Поставщики	Клиенты	Товары-заменители
Составляющие анализа:	Компании по разработке ПО, являющихся лидерами на рынке разработки ПО и к которым уже есть доверие.	Небольшие компании по разработке программного обеспечения. Фрилансеры.	Доступ к системе поставляется за счет сети Интернет	Любые предприятия, желающие заказать разработку системы	Каждый из продуктов конкурентов является, частично, заменителем
Выводы:	Существует достаточно большое количество конкурентных компаний разных размеров.	Выход на рынок зависит от условий заказчика, конкретной ситуации на рынке труда и количества конкурентных решений.	Не влияют никак на происходящее на рынке	Поскольку существует достаточно большое количество конкурентов, основной целью является максимальное удовлетворение потребностей клиента по оптимальной цене	Каждая система, разработанная компанией-конкурентом на основе любой архитектуры, является потенциальным заменителем.

Далее следует привести факторы, делающие проект конкурентоспособным. Сделаем это в таблице 16.

Таблица 16 – Обоснование факторов конкурентоспособности

Фактор конкурентоспособности	Обоснование (наведение факторов, делающих фактор для сравнения конкурентных значимых проектов)
Гибкость архитектурного решения	Разработка системы с использованием микросервисной архитектуры делает систему гибкой и легкой для масштабирования и модификации, привлекательной для заказчиков, планирующих масштабировать свое предприятие.
Ценовая политика	Привлекательная ценовая политика для достаточно сложной системы привлечет дополнительных клиентов
Отказоустойчивость	Архитектура системы предполагает достаточно высокую отказоустойчивость, что позволит заказчику быть уверенным, что сбой системы не принесет ему никаких дополнительных затрат, так как сбой системы сам по себе фактически невозможен.
Оптимизация	Разрабатываемое приложение будет просто оптимизировать и расширять
Универсальность	Предлагаемое архитектурное решение может быть использовано для любых заказов

Таблица 17 – Сравнительный анализ сильных и слабых сторон

Фактор конкурентоспособности	Баллы 1-20	Рейтинг товаров-конкурентов по сравнению с моим продуктом						
		-3	-2	-1	0	1	2	3
Гибкость архитектурного решения	18			+				
Ценовая политика	20		+					
Отказоустойчивость	15				+			
Оптимизация	17				+			
Универсальность	18			+				

Далее можно выделить все сильные и слабые стороны проекта, это сделано в таблице 18 с помощью SWOT анализа. В таблице 19 представлены альтернативы рыночного внедрения проекта.

Таблица 18 – SWOT- анализ проекта

Сильные стороны:	Слабые стороны:
Сильными сторонами предлагаемого архитектурного решения есть гибкость, легкая масштабируемость, универсальность для использования. Большим преимуществом является цена создания ПО с использованием выбранного архитектурного решения по сравнению с рынком.	Высокая конкуренция Трудный процесс разработки Сложный процесс развертывания
Возможности:	Угрозы:
Возможностями является расширение состава команды. Создание ступени проекта, что позволит ускорить процесс развертывания проекта. Интеграция с различными сервисами, что может ускорить процесс разработки и упростить его. Есть возможность использования других архитектурных решений для привлечения новых клиентов.	Микросервисная архитектура может устареть или ее популярность может снизиться. Компании конкуренты могут предложить лучшее решение или цену, что может привести к отсутствию спроса. Компаний конкурентов может появиться больше, что придаст больше конкуренции.

Таблица 19 – Альтернативы рыночного внедрения проекта

Альтернатива (ориентировочный комплекс мер) рыночного поведения	Вероятность получения ресурсов	Сроки реализации
Использование альтернативного архитектурного подхода к построению системы	Средняя	Несколько месяцев
Реализация функционала частично с постепенным добавлением новых сервисов	0.7	Несколько месяцев
Использование альтернативного стека технологий	Средняя	Несколько месяцев
Маркетинговая компания для привлечения новых заказчиков. Реализация демо проектов для заинтересованности.	0.8	Несколько месяцев
Разработка приложения с полным функционалом в кратчайшие сроки на основе разработанного проекта-шаблона	0.8	Несколько месяцев

На основе анализа возможностей выхода на рынок проекта можно сделать несколько выводов:

– потенциальный рынок для проекта растет, но уже существует множество конкурентов, предлагающих различные решения для разработки проекта. Поэтому необходимо хорошо понимать, кто является потенциальными потребителями приложения, и определить целевую группу;
– необходимо учитывать различные факторы угроз и возможностей. К факторам угрозы относятся сложность реализации, отсутствие интереса со стороны новых клиентов, изменение тенденций и критические ошибки. С другой стороны, факторы возможностей включают интерес инвесторов, интеграцию с другими услугами и платформами, сотрудничество с другими компаниями и сотрудничество с частными лицами;
– конкуренция на рынке очень высока, и выход на рынок зависит от условий клиента. Поэтому проект должен предложить уникальный подход к разработке системы, который позволит ускорить разработку и снизить затраты;
– анализ конкуренции в данной области с помощью метода «пяти сил» Майкла Портера показал, что существует достаточно большое количество конкурентоспособных компаний разного размера, а вход на рынок зависит от условий заказчика;
– в целом, проект имеет потенциал на рынке, но для обеспечения его успеха необходимо тщательно изучить рынок и конкуренцию.

4.4 Разработка рыночной стратегии проекта

Для успешного запуска проекта необходимо знать целевые группы проекта. Приведем их в таблице 20.

Таблица 20 – Выбор целевых групп потенциальных потребителей

Описание профиля целевой группы потенциальных клиентов	Готовность потребителей воспринять продукт	Ориентировочный спрос в пределах целевой группы (сегмента)	Интенсивность конкуренции в сегменте	Простота входа в сегмент
Малый бизнес и стартапы	Готовы	Малый	Высокая	Вход в сегмент является достаточно сложным, ведь предложенный архитектурный подход является достаточно сложным и лучше всего подходит системам, которые планируют расти и масштабироваться. Разработка и поддержка таких систем является довольно долгим процессом, когда как для малого бизнеса и стартапов необходимо быстрое решение. Тем не менее, существует комплекс мер для ускорения разработки и запуска проекта, таких как разработка системы из шаблона.
Средний бизнес	Готовые	Средний	Высокая	Вход в сегмент является средним по сложности, все зависит от целей бизнеса. Если бизнес намерен расти и масштабироваться данный архитектурный подход является удачным решением, но если бизнес не планирует расти, то данное решение может быть неудачным ходом
Крупный бизнес	Готовые	Высокий	Высокая	Вход в сегмент является простым, ведь данный архитектурный подход лучше всего подходит для создания больших систем с большим количеством сервисов.
Государственные учреждения	Готовые	Средний/Высокий	Высокая	Данный сегмент чаще всего представляет собой большие и сложные системы поэтому простота входа в сегмент является средней/большой
Физические лица	Готовые	Средний	Высокая	Вход в сегмент является средней простоты ведь для физических лиц скорее всего нужно разрабатывать несложные системы, но данное архитектурное решение можно адаптировать и для малых систем
Целевые группы: №2, №3, №4				

Следующим шагом является определение базовой стратегии развития продукта, приведенная в таблице 21.

Таблица 21 – Определение базовой стратегии развития.

Выбранная альтернатива развития проекта	Стратегия охвата рынка	Ключевые конкурентоспособные позиции в соответствии с выбранной альтернативой	Базовая стратегия развития
Выход на рынок с полным проведением маркетинговой компании. Создание шаблона проекта, который позволит быстрее начать разработку приложения и сможет привлечь малый и средний бизнес. Создание финансовой политики, что сделает компанию привлекательнее конкурентов.	Проведение маркетинговой кампании. Предложение разнообразных ценовых планов для бизнеса разного размера.	Использование альтернативного стека технологий. Использование альтернативного архитектурного подхода.	Концентрированный маркетинг

Далее определим базовую стратегию конкурентного поведения, и добавим результаты в таблицу 22. В таблице 23 представлены стратегии позиционирования.

Таблица 22 – Определение базовой стратегии конкурентного поведения

Является ли проект «первопроходцем» на рынке?	Будет ли компания искать новых потребителей, или забирать существующих у конкурентов?	Будет ли компания копировать основные характеристики товара конкурента, и какие?	Стратегия конкурентного поведения
Нет	Компания будет как искать новых пользователей, так и отбирать существующих у конкурентов.	Копирование выгодных характеристик является обычной практикой, ведь это очевидное следствие конкуренции.	Стратегия занятия конкурентной ниши

Таблица 23 – Определение стратегии позиционирования

Требования к товару целевой аудитории	Базовая стратегия развития	Ключевые конкурентоспособные позиции собственного проекта	Выбор ассоциаций, которые должны сформировать комплексную позицию собственного проекта (три ключевых)
Универсальность	Рост	Позиционирование системы как той, которую можно использовать в различных ситуациях	1. Удобство 2. Универсальность 3. Гибкость
Удобство	Рост	Позиционирование системы как той, которая предоставляет чрезвычайное удобство клиентам	
Функциональность	Стабилизация	Позиционирование системы как той, которая предоставляет большое количество функциональных возможностей клиенту	
Гибкость	Рост	Позиционирование системы как той, которая является чрезвычайно гибкой и легкой для изменений	

В результате проведенного выше анализа выбора стратегии развития проекта мы определили, что целевыми потребителями данного продукта могут

быть средний и крупный бизнес, а также государственные учреждения, для продуктов которых ключевыми параметрами являются удобство, универсальность и гибкость.

4.5 Разработка маркетинговой программы

В таблице 24 представлены ключевые преимущества концепции потенциального товара.

Таблица 24 – Определение ключевых преимуществ концепции потенциального товара

Потребность	Выгода, которую предлагает товар	Ключевые преимущества перед конкурентами (существующие или такие, которые нужно создать)
Универсальность	Архитектурное решение может использоваться для решения большого количества разнообразных задач	Гибкость, Масштабируемость, Отказоустойчивость.
Гибкость	Система является гибкой для модификации и масштабирования	Систему можно быстро масштабировать, модифицировать и интегрировать с другими системами
Дешевизна	Несмотря на то, что разрабатываемая система будет довольно сложной с точки зрения архитектуры, ценовая политика будет оставаться достаточно привлекательной.	Компания сможет предложить свои услуги достаточно широкой аудитории

В таблице 25 представлено описание трех уровней модели товара. В таблице 26 определены границы установления цены. В таблице 27 сформирована система сбыта.

Таблица 25 – Описание трех уровней модели товара

Уровни товара	Сущность и составляющие	
I. Сущность товара	В сущности, товар представляет собой систему по поиску мероприятий построенную на основе микросервисной архитектуры	
II. Товар	Свойства / характеристики:	Размер исходных файлов
	Серверный модуль Клиентский модуль	2,7 Мб 1.8 Мб
	Качество: тестирование с помощью юнит-тестов во время разработки	
	Марка: название разработчика и название системы	
III. Товар с дополнением	До продажи: Стандартная система	
	После продажи: дополнительная поддержка и возможность расширения	

Таблица 26 – Определение границ установления цены

Уровень цен на товары-заменители	Уровень цен на товары аналоги	Уровень доходов целевой группы потребителей	Верхняя и нижняя границы установления цены на товар/услугу
$4000-$100000	$4000-$100000	> $200000	$0-$10000

Таблица 27 – Формирование системы сбыта

Специфика закупочного поведения целевых клиентов	Функции сбыта, которые должен выполнять поставщик товара	Глубина канала сбыта	Оптимальная система сбыта
Оплата услуг по разработке и поддержке программного обеспечения	Подписание контрактов и продажа по указанной цене	Двухуровневый	Через сайт компании производителя

В таблице 28 определена концепция маркетинговых коммуникаций.

Таблица 28 – Концепция маркетинговых коммуникаций

Специфика поведения целевых клиентов	Каналы коммуникации, которыми пользуются целевые клиенты	Ключевые позиции, выбранные для позиционирования	Задачи рекламного сообщения	Концепция рекламного обращения
Заключение контракта на определенный срок	Директ-маркетинг, интернет-сайт	Архитектурный подход позиционирует себя как альтернатива в подходах к разработке программного обеспечения.	Сообщить потенциальным клиентам о возможных вариантах разработки ПО, показать все положительные черты предлагаемого решения и проинформировать о привлекательной ценовой политике.	Рекламное обращение направлено к потенциальным клиентам, где показываются плюсы пользования системой

Для того, чтобы привлечь новых клиентов и увеличить объем продаж продукта, маркетинговая кампания должна строиться на правильном позиционировании товара на рынке. Так как разработанная микросервисная архитектура может масштабироваться, изменяться и адаптироваться к различным проектам - ее можно популяризировать среди потенциальных клиентов в различных видах бизнеса.

Выводы по главе 4

В данном разделе был выполнен анализ проекта, описана его основная идея, проведен технологический аудит и анализ угроз и возможностей, а также выделены слабые и сильные стороны.

В процессе исследования было выявлено, что существует достаточно много конкурентов на рынке, поэтому для повышения конкурентоспособности, необходимо предоставлять лучшие услуги, по меньшим ценам чем у конкурентов.

Были оценены целевые группы и клиенты проекта, а также был сделан вывод, что выбранная стратегия развития является оптимальной. Факторами, которые могут повлиять на внедрение проекта являются возможность появления нового тренда на рынке разработки ПО и сложность построения такой системы. Также были предложены различные решения данных проблем и варианты реакций на них.

В качестве краткого вывода можно добавить, что несмотря на большое количество конкурентов и сложность реализации данного проекта, при достаточной подготовке и грамотной маркетинговой компании, проект является довольно перспективным.

Заключение

В данной диссертации мы разработали новую архитектуру для монолитного веб-приложения, изучив преимущества и недостатки различных архитектурных подходов в разработке программного обеспечения и остановившись на архитектуре микросервисов из-за ее потенциала масштабируемости, гибкости, удобства использования и отказоустойчивости. Мы также разработали и реализовали прототип веб-системы на основе микросервисов, подчеркнув ее преимущества и предоставив шаблон для разработки новых микросервисных систем.

В разделе 3 мы провели анализ и тестирование разработанного решения и выявили преимущества нашей веб-системы на основе микросервисов, включая легкую масштабируемость, гибкость, удобство использования и отказоустойчивость для бесшовного развертывания. Мы показали, что разработанный прототип является необходимым каркасом для работы с неограниченным количеством микросервисов и выполняет такие ключевые функции, как авторизация и аутентификация, работа с конфигурационными файлами, балансировка нагрузки, обнаружение и связывание независимых сервисов и т.д. Наши микросервисы автономны и могут использоваться отдельно как сервисы, предоставляя открытый API. Разработанное решение достаточно отказоустойчивой, и остановка одного сервиса не приведет к краху всей системы.

В разделе 4 мы разработали бизнес-план нашего проекта, оценив его основную идею, проведя технологический аудит и анализ угроз и возможностей, а также выделив слабые и сильные стороны. Мы предложили решения для устранения возможных проблем, таких как конкуренция на рынке и сложность создания такой системы. Мы также оценили целевые группы и потребителей проекта и пришли к выводу, что выбранная нами стратегия развития является оптимальной.

В целом, наше исследование показало, что архитектура микросервисов является перспективным подходом для разработки веб-систем, обеспечивая значительные преимущества в плане масштабируемости, гибкости, удобства использования и отказоустойчивости. Наш прототип служит шаблоном для разработки новых микросервисных систем, позволяя разработчикам сосредоточиться на бизнес-логике и избежать проблем на начальном этапе. Бизнес-план, разработанный в разделе 4, демонстрирует потенциал нашего проекта на рынке, и при достаточной подготовке и грамотной маркетинговой кампании, наш проект может стать конкурентоспособным и успешным.

Научная и практическая значимость нашего исследования заключается в том, что мы разработали микросервисную архитектуру для монолитного веб-приложения, которая потенциально может решить многие проблемы масштабируемости, гибкости и отказоустойчивости, с которыми сталкиваются традиционные монолитные и сервис-ориентированные архитектуры. Наш прототип может послужить полезным руководством для разработчиков, стремящихся реализовать веб-системы на основе микросервисов, а наш бизнес-план может помочь предпринимателям оценить потенциал такого проекта на рынке.

Список используемой литературы и используемых источников

1. Глибина М.Д. Сравнительный анализ монолитной и микросервисной архитектуры разработки и проектирования программного обеспечения // «Современные инновации в науке и технике», сборник научных трудов 10-й Всероссийской научно-технической конференции с международным участием. 2020. С. 74-78.
2. Гольчевский Ю.В., Ермоленко А.В. Актуальность использования микросервисов при разработке информационных систем // журнал «Вестник Сыктывкарского университета». Серия 1: «Математика. Механика. Информатика». 2020. № 2 (35) с. 25-36.
3. Жантлеуова А.К. Введение в микросервисы: характеристики, преимущества и недостатки // журнал «Наука третьего тысячелетия», материалы Международной (заочной) научно-практической конференции. Нефтекамск, 2021. С. 41-46
4. Кабарухин А.П. Выгоды перехода от монолитной к микросервисной архитектуре приложения // журнал «Проблемы современной науки и образования». 2022. № 1 (170). с. 18-23.
5. Караханова А.А. Анализ микросервисной архитектуры, монолитных приложений, архитектуры SOA // журнал «Синергия наук». 2020. № 46. с. 255-262.
6. Малюга К.В., Перл И.А. Особенности коммуникации микросервисов при использовании шаблона сага // журнал «Инфокоммуникационные технологии». 2021. № 4. с. 425-435.
7. Осипова Н.Д. Разработка микросервиса интеграции системы самообслуживания абонентов сотовой связи и центра нотификаций в рамках перехода от монолитной архитектуры приложения к микросервисной // журнал «Естественные и математические науки в современном мире». 2017. № 4-5 (51). с. 9-13.

8. Паттерн Circuit Breaker [Электронный ресурс] // Microsoft - Microsoft - [Электронный ресурс]. URL: https://docs.microsoft.com/enus/azure/architecture/patterns/circuit-breaker (дата обращения: 10.09.2022).

9. Потапенко А.С. Сравнение монолитной и микросервисной архитектур // Журнал «Научные горизонты». 2020. № 5 (33). с. 297-303.

10. Рудольф Мачадо «Монолитная архитектура» / Рудольф Мачадо - Рудольф Мачадо. К.: 1995.

11. Танатканова А.К. Применение микросервисной архитектуры при разработке корпоративных веб-приложений // журнал «Вестник науки». 2019. № 5 (14). с. 149-153.

12. Тхуан Л. Тай/Хоан К. Лам «.NET Framework Essentials: Introducing the .NET Framework» / Тхуан Л. Тай/Хоан К. Лам - К.: «O'REILLY», 2003.

13. Шилдт Г. «Java 8. Полное руководство, 9-е издание» / Шилдт Г. - К.: «Вильямс», 2015.

14. Щелбанин А.В. Особенности применения микросервисной архитектуры при разработке программного обеспечения // «Фундаментальные и прикладные научные исследования: актуальные вопросы, достижения и инновации», сборник статей победителей V международной научно-практической конференции: в 4 частях. Том 1, часть 4. 2017. с. 112-117.

15. Bharathan Raghuram «Apache Maven Cookbook» / Bharathan Raghuram - К.: «PACKT», 2015.

16. David A. Black «The Well-Grounded Rubyist» / David A. Black, К.: «MANNING», 2009.

17. Express API [Электронный ресурс] // Express - [Электронный ресурс]. URL: https://expressjs.com/en/4x/api.html (дата обращения: 11.10.2022).

18. Fowler, M. (2018). Monolith First. // Fowler, M. [Электронный ресурс]. URL: https://martinfowler.com/bliki/MonolithFirst.html (дата обращения: 12.11.2022).

19. JPA [Электронный ресурс] // Spring - [Электронный ресурс]. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (дата обращения: 14.11.2022).

20. JSON Web Token // Auth0 - [Электронный ресурс]. URL: https://jwt.io/introduction/ (дата обращения: 17.11.2022).

21. John Cornell «Spring Microservices in Action» / John Cornell - К: «MANNING», 2017.

22. Laurentiu Spilca «Spring Security in Action» / Laurentiu Spilca - К.: «MANNING», 2019.

23. Fowler M. Microservices // Martin Fowler - [Электронный ресурс]. URL: https://martinfowler.com/articles/microservices.html (дата обращения: 27.11.2022).

24. Netflix OSS // Netflix - [Электронный ресурс]. URL: https://netflix.github.io/ (дата обращения: 17.01.2023).

25. Nodejs Documentation // Nodejs - [Электронный ресурс]. URL: https://nodejs.org/api/ (дата обращения: 25.10.2022).

26. Pattern: API Gateway/Backend for Frontends [Электронный ресурс]. URL: https://microservices.io/patterns/apigateway.html (дата обращения: 20.10.2022).

27. Spring Cloud Netflix [Электронный ресурс] // Spring - [Электронный ресурс]. URL: https://cloud.spring.io/spring-cloud-netflix/multi/multi__service_discovery_eureka_clients.html (дата обращения: 10.01.2023).

28. Spring Cloud [Электронный ресурс] // Spring - [Электронный ресурс]. URL: https://spring.io/projects/spring-cloud#overview (дата обращения: 17.01.2023).

29. Spring Data JPA [Электронный ресурс] // Spring - [Электронный ресурс]. URL: https://spring.io/projects/spring-data-jpa (дата обращения: 12.01.2023).

30. The Java Tutorials // Oracle - [Электронный ресурс]. URL: https://docs.oracle.com/javase/tutorial/ (дата обращения: 09.02.2023).

31. Thomas Earle «SOA Principles of Service Design» / Thomas Earle - К.: «Prentice», 2008 г., 115

32. Williams, J. (2019). The Evolution of Monolithic Architecture. // [Электронный ресурс]. URL: https://blog.heroku.com/the-evolution-of-monolithic-architecture. (дата обращения: 04.01.2023).