

Федеральное государственное автономное образовательное учреждение
высшего образования «Омский государственный университет им. Ф.М.
Достоевского»

Гресь Владимир Игоревич

**Создание инфраструктуры обслуживания
микросервисов для информационно-аналитического
онлайн-ресурса «Православный ландшафт таежной
Сибири»: проектирование, реализация и оптимизация**

Направление 09.03.03 —
«Прикладная информатика»

Научный руководитель:
уч. степень, уч. звание
Фамилия Имя Отчество

Омск — 2024

Оглавление

	Стр.
Введение	4
Глава 1. Теоритическая часть	6
1.1 Системы автоматической сборки	6
1.1.1 Обоснование выбора Gradle	7
1.2 API Gateway. Маршрутизация и фильтрация	9
1.2.1 Прямое взаимодействие клиента и сервиса	9
1.2.2 Api Gateway	11
1.2.3 Сравнение популярных решений	14
1.2.4 Spring Cloud Gateway	18
1.3 Сервер конфигураций	20
1.3.1 Выбор сервера конфигурации	21
1.3.2 Заключение	25
Глава 2. Практическая часть	26
2.1 Интеграция системы автоматической сборки Gradle	26
2.1.1 Организация проекта Gradle для проекта аналитики исторических данных	26
2.1.2 Управление зависимостями и плагинами в Gradle	29
2.1.3 Создание пользовательских задач и скриптов в Gradle	30
2.1.4 Оптимизация процесса сборки с использованием Gradle	31
2.2 Внедрение API Gateway	32
2.2.1 Анализ требований и постановка задач	32
2.2.2 Основные компоненты архитектуры	32
2.2.3 Описание архитектуры проекта до внедрения Spring Cloud Gateway	33
2.2.4 Проектирование архитектуры с использованием Spring Cloud Gateway	34
2.2.5 Реализация в проекте	35
2.2.6 Внедрение SSL сертификатов	36
2.3 Проектирование архитектуры проекта с учетом сервера конфигурации	37
2.3.1 Схема взаимодействия компонентов	38

	Стр.
2.3.2 Безопасность и контроль доступа	38
2.3.3 Репликация и отказоустойчивость	39
2.3.4 Интеграция сервера конфигурации с микросервисами .	39
2.3.5 Настройка подключения микросервисов к серверу конфигурации	39
Заключение	43
Список литературы	44

Введение

Из множества источников исторической информации, наиболее полные сведения о культурном наследии мировой истории содержатся в письменных источниках. Во многом благодаря письменным документам можно восстановить наиболее точную картину прошлого. Однако, исследования показали, что сохранение таких источников в надлежащем состоянии представляет собой значительную проблему [1]. Естественное старение, ненадлежащее хранение и небрежное обращение с письменными документами могут привести к их безвозвратной утрате, что наносит огромный ущерб культурному наследию. Более того, отсутствие единого стандарта при их создании приводит к значительному разнообразию видов и типов документов, что усложняет работу с ними в ходе исследований.

Информационно-аналитический онлайн-ресурс «Православный ландшафт таежной Сибири» разрабатывается с целью решения, по крайней мере, нескольких этих проблем. Все исторические данные, загруженные в систему, будут оцифрованы и приведены к единому стандарту, что ускорит исторические исследования и упростит работу с данными. Пользователь сможет просматривать данные в удобном формате, будь то текстовый или графический вариант представления. Оцифровка также позволит сократить использование оригинальных письменных экземпляров, поскольку станет возможной работа с множеством источников в одном месте. Это способствует более длительному сохранению физических оригиналов в хорошем состоянии. Кроме того, оцифрованные копии исторических данных будут храниться в нескольких местах, что обеспечит их доступность и предотвратит безвозвратную утрату ценной информации.

Целью данной работы является проектирование и разработка оптимальной инфраструктуры для работы микросервисов в проекте «Православный ландшафт таежной Сибири».

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Исследовать актуальные методологии и подходы к проектированию и внедрению микросервисов.

2. Разработать архитектуру системы, включающую все необходимые компоненты и сервисы.
3. Обеспечить интеграцию разработанной инфраструктуры с существующими системами и базами данных.

Глава 1. Теоритическая часть

1.1 Системы автоматической сборки

В свете быстрого развития современных информационных технологий и повышенных требований к эффективности программных систем, внедрение интегрированных средств автоматизации сборки является неотъемлемым этапом в процессе разработки программного обеспечения. В данной главе рассматривается важная составляющая инфраструктуры разработки - интеграция системы автоматической сборки в контексте платформы аналитики исторических данных с использованием микросервисной архитектуры.

Gradle представляет собой мощный инструмент для автоматизации процессов сборки и управления зависимостями в проектах различной сложности. Его гибкость и расширяемость позволяют эффективно управлять процессом сборки как небольших приложений, так и крупных программных комплексов. В контексте платформы аналитики исторических данных, где требуется обеспечить высокую производительность и надежность при обработке больших объемов информации, внедрение Gradle представляется важным шагом для оптимизации процесса разработки и поддержки программного обеспечения.

Данная глава направлена на анализ методов интеграции системы автоматической сборки Gradle в существующую инфраструктуру обслуживания микросервисов платформы аналитики исторических данных. В частности, рассматриваются вопросы настройки Gradle для удовлетворения требований по сборке, тестированию и развертыванию микросервисов, а также оптимизации процесса разработки и управления зависимостями. Предполагается, что результаты данного исследования смогут служить основой для разработки эффективной и масштабируемой инфраструктуры сборки в рамках платформы аналитики исторических данных, способствуя повышению производительности и надежности разрабатываемых приложений.

1.1.1 Обоснование выбора Gradle

В данном разделе обосновывается выбор системы автоматической сборки Gradle в качестве основного инструмента для интеграции в инфраструктуру разработки на платформе аналитики исторических данных. Рассматриваются основные преимущества и функциональные возможности Gradle, а также адаптация данной системы к требованиям и особенностям разрабатываемого программного комплекса. Рассматриваются факторы, влияющие на принятие данного решения.

Системы автоматизированных сборок

Gradle — это система автоматизации сборки с открытым исходным кодом, в которой используются те же идеи, что и в Apache Maven и Apache Ant. Он использует предметно-ориентированный язык, основанный на компьютерном языке Groovy или Kotlin, в отличие от Apache Maven, настройка проекта которого использует XML [2].

Основные различия

Существуют существенные различия в подходах к построению двух систем. Gradle основан на модели сети зависимостей задач, где каждая задача представляет собой исполняемый объект, выполняющий определенные действия. В отличие от этого, в Maven стадии проекта связаны с целями, которые функционируют аналогично задачам в Gradle как объекты, осуществляющие выполнение работы.

Производительность

Оба фреймворка обеспечивают возможность одновременного выполнения множества сборок модулей с учетом аспектов производительности. Gradle, в частности, обладает возможностью проведения инкрементных сборок, так как он способен определить, какие задачи были изменены. Кроме того, Gradle обладает рядом выдающихся характеристик производительности, среди которых следует выделить:

1. Инкрементальная компиляция классов Java, что позволяет значительно сократить время сборки за счет перекомпиляции только тех классов, которые были изменены.
2. Компиляция предотвращения для Java, предоставляющая возможность быстрого обнаружения ошибок в коде на этапе компиляции.
3. API для дополнительных подзадач, что обеспечивает гибкость в настройке и расширении процесса сборки.
4. Демон компилятора, который значительно ускоряет процесс компиляции за счет сохранения в памяти некоторых результатов предыдущих компиляций.

На рисунке 1 представлен сравнительный тест производительности автоматизированной сборки одинаковых проектов и прохождение тестов [3].

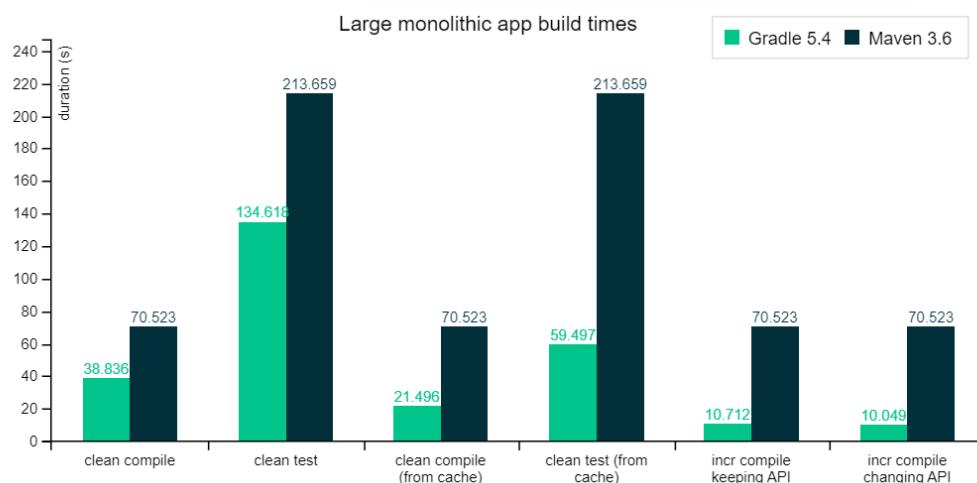


Рисунок 1 — Сравнительный тест производительности

1.2 API Gateway. Маршрутизация и фильтрация

В нынешних стандартах информационных технологий существует понятие цифровой экосистемы, требующей эффективной обработки и управления потоком данных. В контексте цифровой экосистемы появляется надобность в инструментарии, способном на рационализацию и организацию потока данных между клиентами и целевыми микросервисами. API Gateway становится одним из важных решений в данном контексте.

API Gateway – инструмент, предоставляющий функциональность маршрутизации, фильтрации и авторизации трафика. Он является одним из важных компонентов в рамках микросервисной архитектуры, способствующий организации внутренних и внешних взаимодействий между компонентами системы.

1.2.1 Прямое взаимодействие клиента и сервиса

В контексте современных подходов к разработке микросервисных платформ в некоторых случаях рассматривается возможность прямого взаимодействия клиента с микросервисом. Этот подход предполагает, что клиентское приложение способно напрямую отправлять запросы к отдельным микросервисам без посредника между ними. В данном сценарии, каждый микросервис предоставляет общедоступный эндпоинт, часто с индивидуальным TCP-портом, что обеспечивает прямой доступ к конкретному сервису [4].

При таком подходе первостепенно возникает необходимость в прозрачном механизме для управления трафиком и обеспечения безопасности взаимодействия между клиентом и микросервисами. В данном контексте, могут быть задействованы дополнительные инструменты, например балансирущики нагрузки или ADC, которые помогают обеспечить не только равномерное распределение запросов между микросервисами, но и обеспечивают уровень базовый уровень защиты, к примеру использование при запросах протокола SSL для шифрования соединения.

Однако, при создании масштабных приложений на основе микросервисов, особенно при взаимодействии с удаленными мобильными приложениями или веб-приложениями SPA, сталкиваются с рядом множественных вызовов. При увеличении системы вызовы требуют минимизации обращений к серверной части для сокращений задержки обращения и расходов на эксплуатацию, управление сквозными задачами, такими как авторизация и безопасность, а также создание специализированных дополнительных фасадов для оптимизации работы и разделение бизнес-логики для различных типов клиентских приложений [5].

Следовательно, в контексте архитектуры прямого взаимодействия клиент-микросервис, важно учитывать не только технологические аспекты, но и сложившиеся практики и вызовы, с которыми сталкиваются разработчики при проектировании и развертывании современных цифровых приложений.

Пример прямой связи клиента и сервиса представлен на рисунке 2

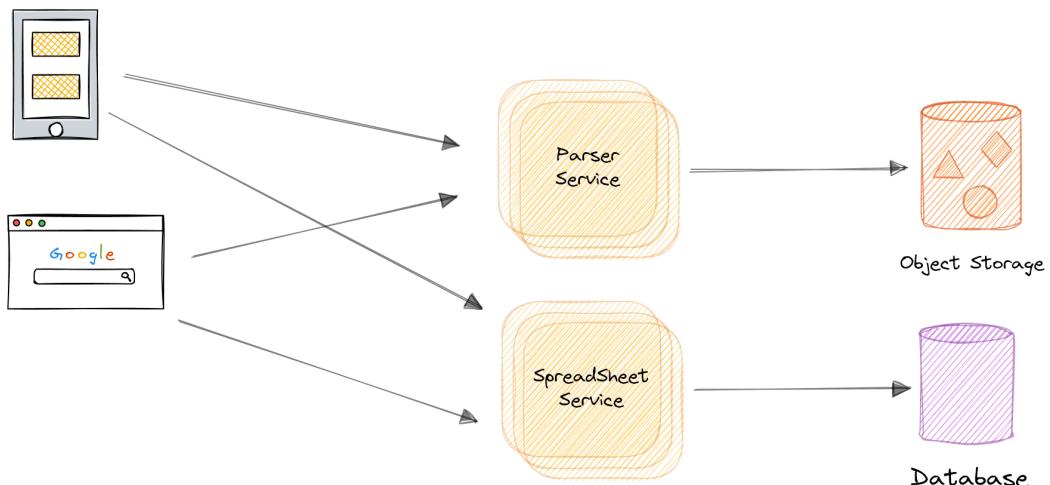


Рисунок 2 – Прямая связь клиента и сервиса

Проблемы подхода

Следует учитывать, что при прямом соединении клиента и отдельных сервисов нелинейно увеличивается сложность взаимодействия и управления. Связано это с необходимостью поддержания соединения с клиентом

на протяжении всего запроса, что потенциально увеличивает нагрузку на ресурсы сети.

Ко всему такой подход несет серьезные проблемы с безопасностью, так как отсутствует централизованный механизм контроля аутентификации. Эта проблема несет за собой, в том числе и проблемы поддержания единого соглашения политики безопасности сервиса из-за необходимости каждого отдельного сервиса обеспечивать собственную безопасность.

Наконец, прямое взаимодействие усложняет обновление и масштабирование системы. При внесении изменений в микросервисы или добавлении новых компонентов необходимо обновлять каждое клиентское приложение, что может быть трудоемким и время затратным процессом.

1.2.2 Api Gateway

Шаблон API Gateway представляет собой концепцию, используемую при разработке и архитектуре масштабных или комплексных приложений, основанных на архитектуре микросервисов и включающих несколько клиентских приложений. В контексте данной практики API Gateway представляет собой сервис, который функционирует как централизованная точка входа для определенных кластеров микросервисов. Этот шаблон, аналогичный концепции фасада в объектно-ориентированном проектировании, является составной частью децентрализованных систем [6].

Следовательно, API Gateway размещается между клиентскими приложениями и микросервисами, действуя в качестве обратного прокси-сервера, который маршрутизирует запросы от клиентов к соответствующим сервисам. Тем не менее API Gateway обеспечивает дополнительную функциональность, такую как аутентификация, SSL и кэширование.

На рисунке 3 представлена диаграмма, иллюстрирующая интеграцию пользовательского API Gateway в упрощенную схему архитектуры микросервисов.

Важно подчеркнуть, что в представленной схеме используется единый пользовательский сервис API Gateway, обслуживающий несколько различных клиентских приложений. Это может представлять риск в плане

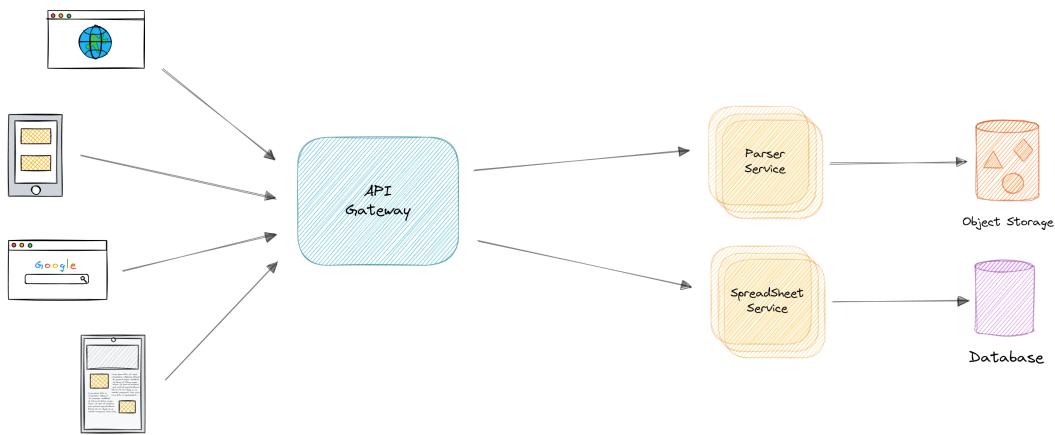


Рисунок 3 — Связь через API Gateway

масштабирования и поддержки, так как сервис API Gateway становится весьма сложным из-за разнообразных требований клиентских приложений. Это может привести к его увеличению и, в конечном итоге, превращению в монолитное приложение или сервис. Следовательно, рекомендуется разделение API Gateway на несколько служб или отдельных API Gateways в соответствии с бизнес-границами и потребностями клиентских приложений.

Разработка и реализация API Gateway требует осторожного подхода. Объединение всех внутренних микросервисов приложения в единый API Gateway может привести к нарушению автономии микросервисов и созданию архитектурного оркестратора, что не рекомендуется.

BFF

При разделении уровня API Gateway на несколько компонентов, например, при наличии нескольких клиентских приложений, основным руководством является создание различных эндпоинтов API Gateways в соответствии с типами клиентов. Это может быть реализовано через применение шаблона «Серверная часть для интерфейса» (BFF), где API Gateway предоставляет специализированный API для конкретного типа клиентского приложения [4](#).

Для платформ, использующих множество сервисов, важность применения агрегации возрастает, так как требуется объединение нескольких низкоуровневых вызовов для обеспечения пользовательских функций [\[7\]](#).

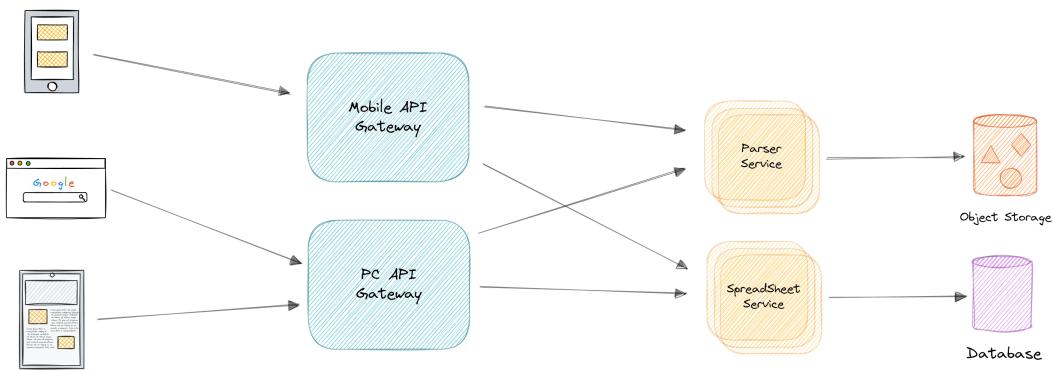


Рисунок 4 – BFF

В таких сценариях один вызов к шлюзу для фронтендов (BFF) обычно порождает ряд низкоуровневых вызовов к разным микросервисам. Примером может послужить приложение для аналитики исторических данных, в котором пользователь требует получения списка координат церквей из определенного уезда для визуализации движения рукописи за определенный период времени.

С точки зрения оптимизации ресурсов было бы целесообразно осуществлять как можно больше вызовов параллельно. После завершения начального вызова к сервису «Поиска координат» оптимально выполнять вызовы других сервисов параллельно, с целью сократить общее время обработки [8]. 5

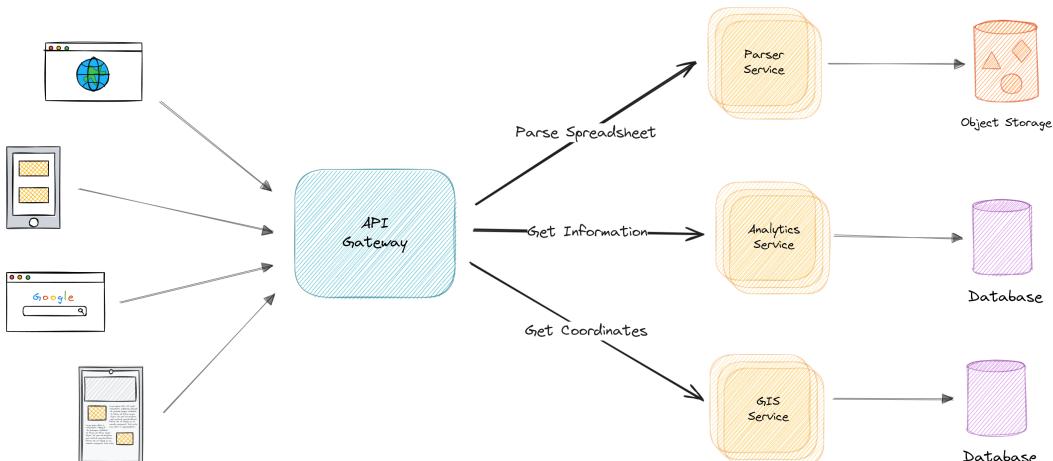


Рисунок 5 – BFF

1.2.3 Сравнение популярных решений

Для сравнения API Gateway необходимо определить критерии, по которым будут оцениваться различные решения. Основные критерии включают производительность, масштабируемость, безопасность, интеграцию с другими сервисами, удобство настройки и использования, а также сообщество и поддержку.

Сравниваемые решения включают:

- Spring Cloud Gateway
- Kong
- Amazon API Gateway
- Apigee
- NGINX

Производительность

Spring Cloud Gateway демонстрирует высокую производительность благодаря реактивной архитектуре на основе проекта Spring WebFlux. Это обеспечивает асинхронную обработку запросов и высокую пропускную способность.

Kong также имеет высокую производительность благодаря использованию NGINX и LuaJIT, что позволяет обрабатывать тысячи запросов в секунду с низкой задержкой.

Amazon API Gateway интегрирован с AWS инфраструктурой, обеспечивая надежную и масштабируемую производительность. Однако, для достижения высокой производительности могут потребоваться дополнительные настройки и использование AWS Lambda.

Apigee обладает хорошей производительностью, однако может уступать в сценариях с высокими нагрузками по сравнению с решениями на основе NGINX.

NGINX известен своей высокой производительностью и низкой задержкой, что делает его популярным выбором для API Gateway.

Масштабируемость

Spring Cloud Gateway легко масштабируется в рамках экосистемы Spring Cloud, поддерживая кластеризацию и интеграцию с различными системами оркестрации, такими как Kubernetes [9].

Kong поддерживает горизонтальное масштабирование и интеграцию с различными оркестраторами контейнеров, что делает его подходящим для крупных распределённых систем.

Amazon API Gateway предлагает автоматическое масштабирование в рамках AWS, что позволяет обрабатывать любые объемы трафика без необходимости управления инфраструктурой.

Apigee предоставляет масштабируемость на уровне предприятия, поддерживая гибкость и высокую доступность.

NGINX, как и Kong, легко масштабируется горизонтально, что позволяет его использовать в крупных распределённых системах.

Безопасность

Spring Cloud Gateway поддерживает интеграцию с различными решениями для аутентификации и авторизации, такими как OAuth2 и JWT, а также предоставляет возможности для настройки правил безопасности.

Kong обладает мощными функциями безопасности, включая аутентификацию, авторизацию, SSL/TLS шифрование и контроль доступа на уровне API.

Amazon API Gateway интегрируется с AWS IAM для управления доступом, а также поддерживает API ключи, OAuth и пользовательские авторизационные механизмы.

Apigee предлагает широкий набор функций безопасности, включая шифрование, управление доступом и интеграцию с системами аутентификации.

NGINX обеспечивает базовые функции безопасности и поддерживает расширенные настройки через модули и интеграции с другими решениями безопасности.

Интеграция с другими сервисами

Spring Cloud Gateway тесно интегрирован с экосистемой Spring, что обеспечивает простую интеграцию с микросервисами и другими компонентами на базе Spring.

Kong поддерживает интеграцию с различными базами данных, системами мониторинга и логирования, а также предоставляет множество плагинов для расширения функциональности.

Amazon API Gateway интегрируется с различными сервисами AWS, такими как Lambda, DynamoDB, S3 и другие, что делает его мощным инструментом для построения серверлесс-приложений.

Apigee предлагает интеграцию с различными облачными платформами и сервисами, а также поддерживает гибкие возможности расширения через API и SDK.

NGINX может интегрироваться с различными сервисами через модули и конфигурации, обеспечивая гибкость в построении сложных инфраструктур.

Удобство настройки и использования

Spring Cloud Gateway предоставляет декларативный способ настройки через YAML или Java-код, что делает его удобным для разработчиков, знакомых с экосистемой Spring.

Kong предлагает интуитивно понятный интерфейс и мощный API для управления конфигурациями, что упрощает его настройку и использование.

Amazon API Gateway предоставляет удобный веб-интерфейс и API для управления, но может потребовать времени на освоение всех возможностей и особенностей настройки в контексте AWS.

Apigee предлагает мощный интерфейс для управления API и инструментами аналитики, что облегчает процесс настройки и мониторинга.

NGINX требует глубокой настройки через конфигурационные файлы, что может потребовать значительных усилий для оптимальной настройки и управления.

Сообщество и поддержка

Spring Cloud Gateway пользуется поддержкой большого сообщества разработчиков Spring, что обеспечивает доступ к обширной документации, форумам и ресурсам.

Kong имеет активное сообщество и предоставляет коммерческую поддержку через Kong Enterprise.

Amazon API Gateway получает поддержку от AWS, включая обширную документацию, обучающие материалы и техподдержку.

Apigee обладает сильной поддержкой от Google, включая документацию, обучающие материалы и коммерческую поддержку.

NGINX имеет большое сообщество пользователей и разработчиков, а также коммерческую поддержку через NGINX Plus.

Вывод

Spring Cloud Gateway, благодаря своей высокой производительности, простоте интеграции с экосистемой Spring, мощным возможностям настройки и широкому сообществу, является предпочтительным выбором для проектов, использующих стек технологий Spring. Он обеспечивает гибкость, масштабируемость и безопасность, необходимые для современных

микросервисных архитектур, и, таким образом, заслуженно выходит победителем в этом сравнении.

1.2.4 Spring Cloud Gateway

Для проекта платформы аналитики исторических данных перед API Gateway выдвигаются определенные требования:

1. Должен маршрутизировать запросы в различные модификации сервисов в зависимости от клиента.
2. Поддержка OAuth 2.0 для передачи информации о пользователи в API сервисы
3. Метрики для оценки загрузки сервиса и задержки ответов.
4. Логирование запросов и ответов.

Spring Cloud Gateway успешно справляется со всеми выдвигаемыми требованиями. Тем не менее его важное преимущество – работа в режиме non-blocking.

Non-blocking Spring Cloud Gateway

Неблокирующая схема, реализованная вне контекста сервлетов и функционирующая на основе Netty, основанного на проекте Reactor, представляет собой асинхронный фреймворк, ориентированный на event loop'ы, а не на статически определенное количество потоков. В данной схеме может присутствовать только один поток. Каждый процессорный поток управляет одной петлей, в которой сосредоточены каналы, принимающие входящие TCP-соединения, декодирующие HTTP-запросы (это происходит быстро) и передающие их дальше.

Запрос поступает через HTTP в канал, затем через петлю событий и далее в приложение WebFlux (реактивный подход к написанию приложений Spring, основанный на Project Reactor) в виде объекта ServerHttpRequest – нового объекта, заменяющего HttpServletRequest. Это происходит небло-

кирующим образом: петля событий получает событие из канала, обрабатывает его, создает флаг ожидания ответа, отправляет его дальше по цепочке и готова снова принимать запросы. WebFlux обворачивает его в ServerWebExchange, который содержит в себе и запрос, и ответ, и передает их дальше по цепочке до WebClient - того же HttpClient, но способного работать асинхронно.

Таким образом, получается аналог контекста сервлетов, но уже работающий в неблокирующем режиме, с использованием реактивных операторов. Ко всему же, все происходит асинхронно, так что даже при наличии только одного процессорного потока все будет функционировать [10].

Circuit breaker pattern

В процессе обработки запросов микросервисы часто взаимодействуют друг с другом. Когда один сервис синхронно вызывает другой, существует вероятность того, что второй сервис будет недоступна или его задержка будет настолько велика, что пользование им станет практически невозможным. Это может привести к неэффективному использованию ресурсов, например, потоков, которые могут быть заблокированы вызывающей стороной в ожидании ответа от другой службы. Такие ситуации могут привести к истощению ресурсов и, в конечном итоге, вызвать неспособность обработки других запросов вызывающей службой. Кроме того, сбой одной службы может потенциально повлиять на работу других служб в рамках всего приложения [11].

Встает вопрос о необходимости предотвращения распространения сбоев одних сервисов на другие.

Для обеспечения взаимодействия клиента службы с удаленной службой необходимо использовать прокси-сервер, функционирующий по аналогии с электрическим выключателем. При достижении определенного порогового значения последовательных сбоев происходит автоматическое срабатывание выключателя. В течение установленного периода ожидания все попытки обращения к удаленной службе завершаются неудачей. По

истечении данного времени автоматический выключатель пропускает ограниченное количество тестовых запросов. В случае успешного завершения данных запросов выключатель восстанавливает нормальный режим работы. Однако, если тестовые запросы завершаются неудачно, происходит повторный запуск периода ожидания, и цикл возобновляется [12].

Такой подход имеет и минусы из-за неправильного подбора значения тайм-аута из-за чего в следствии создаются ложные срабатывания и дополнительные задержки.

1.3 Сервер конфигураций

Внедрение сервера конфигурации в инфраструктуру микросервисов обладает значительной важностью, что обусловлено рядом факторов, влияющих на стабильность, управляемость и масштабируемость систем [9].

Во-первых, сервер конфигурации позволяет централизованно управлять настройками множества микросервисов. В условиях, когда каждый микросервис может иметь свои уникальные параметры, централизованное управление конфигурациями обеспечивает консистентность и согласованность между различными компонентами системы. Это особенно критично в распределенных системах, где конфигурационные изменения должны оперативно и синхронно применяться ко всем компонентам.

Во-вторых, централизованное хранение конфигураций облегчает процесс обновления и развертывания микросервисов. Внедрение сервера конфигурации позволяет динамически изменять параметры конфигурации без необходимости перезапуска самих микросервисов. Это сокращает время простоя системы и повышает её доступность, что важно для бизнес-критичных приложений.

В-третьих, сервер конфигурации улучшает безопасность системы. Конфиденциальные данные, такие как ключи API и пароли, могут храниться в зашифрованном виде и предоставляться только авторизованным микросервисам. Это снижает риск утечек данных и упрощает управление доступом к чувствительной информации.

Четвертое значительное преимущество - улучшение масштабируемости. С ростом количества микросервисов и их экземпляров, управление конфигурацией становится всё более сложной задачей. Сервер конфигурации позволяет легко масштабировать системы, обеспечивая быстрый и гибкий доступ к необходимым параметрам конфигурации независимо от числа микросервисов.

В заключении, внедрение сервера конфигурации способствует лучшей отслеживаемости и управляемости изменений конфигураций. Введение версионирования конфигурационных файлов позволяет отслеживать историю изменений, что облегчает диагностику и устранение неполадок. Это обеспечивает прозрачность и контроль над изменениями в системе, что важно для поддержания её стабильности и надежности.

Таким образом, сервер конфигурации является одним из ключевых компонентов современной архитектуры микросервисов, способствующим повышению эффективности управления, безопасности, масштабируемости и стабильности системы.

1.3.1 Выбор сервера конфигурации

Критерии выбора сервера конфигурации

При выборе сервера конфигурации для платформы анализа исторических данных необходимо учитывать ряд ключевых критериев, которые помогут обеспечить надежную и эффективную работу всей инфраструктуры.

Первым критерием можно выделить необходимость в высокой доступности, то есть сервис должен обладать механизмами автоматического восстановления и способностью к горизонтальному масштабированию, чтобы минимизировать время простоя. Следующим фактором при выборе должна быть способность обеспечения минимальных задержек при доступе к конфиг файлам, а также возможность поддержки различных форматов файлов, например ‘json’, ‘yaml’ и другие.

Не стоит забывать и про лицензию для использования. Необходимо перед использованием ознакомиться с условиями лицензирования и возможностью использования сервера конфигурации в коммерческих и некоммерческих проектах.

Также должны поддерживаться методы шифрования конфигураций как в состоянии потока, так и при передаче.

Опциональным условием будет наличие активного сообщества разработчиков у проекта, потому наличие такого сообщества, способно предоставить обновление или улучшение продукта.

Сравнение популярных решений

Для сравнения популярных решений для управления конфигурацией и сервисами в распределённых системах рассмотрим три наиболее часто используемых инструмента: Consul, etcd и Spring Cloud Config. Каждый из этих инструментов имеет свои особенности, преимущества и недостатки, которые будут проанализированы в данной теме.

Consul

Описание: Consul — это распределённая система управления конфигурациями, обнаружения сервисов и организации сервисной сети (service mesh), разработанная компанией HashiCorp. Consul предоставляет функционал для регистрации сервисов, проверки их состояния, распределённого хранения конфигураций и автоматизации сетевого взаимодействия между сервисами.

Преимущества:

1. **Обнаружение сервисов:** Consul автоматически обнаруживает и регистрирует сервисы, что облегчает управление сложными инфраструктурами.
2. **Проверка состояния:** Встроенные механизмы health-check позволяют отслеживать состояние сервисов в реальном времени.
3. **Ключ-значение хранилище:** Consul использует KV-хранилище для распределённого хранения конфигураций.

4. **Service Mesh:** Consul поддерживает функции сервисной сети, включая управление сетевыми политиками и мидлварное шифрование.

Недостатки:

1. **Сложность настройки:** Конфигурация и интеграция Consul может быть сложной, особенно в больших кластерах.
2. **Ресурсоёмкость:** Требования к ресурсам для поддержания Consul-кластера могут быть высокими, что ограничивает его использование в небольших проектах.

etcd

Описание: etcd — это распределённое хранилище конфигураций, ориентированное на консенсус, разработанное CoreOS. Оно используется в различных системах, таких как Kubernetes, для хранения конфигурационных данных и метаданных.

Преимущества:

1. **Согласованность данных:** etcd обеспечивает строгое согласование данных благодаря алгоритму Raft.
2. **Высокая производительность:** Высокая производительность и низкая задержка доступа к данным.
3. **Простота:** Простая и лёгкая в использовании API для операций с данными.
4. **Интеграция с Kubernetes:** Является основным хранилищем конфигураций для Kubernetes, что обеспечивает глубокую интеграцию и стабильность.

Недостатки:

1. **Ограниченный функционал:** В отличие от Consul, etcd не предоставляет встроенные механизмы для обнаружения сервисов и проверки их состояния.
2. **Сложность управления:** Управление кластером etcd требует тщательного планирования и мониторинга, особенно в больших масштабах.

Spring Cloud Config

Описание: Spring Cloud Config — это инструмент для управления конфигурациями в облачных приложениях, входящий в экосистему Spring.

Он позволяет централизованно управлять конфигурационными файлами и изменениями конфигураций для различных приложений.

Преимущества:

1. **Интеграция с Spring:** Глубокая интеграция с экосистемой Spring облегчает использование в Spring-приложениях.
2. **Централизованное управление:** Позволяет централизованно управлять конфигурациями для множества приложений.
3. **Поддержка версионирования:** Интеграция с системами контроля версий (такими как Git) позволяет отслеживать изменения конфигураций и возвращаться к предыдущим версиям.

Недостатки:

1. **Ограниченнная функциональность:** Основной фокус на управление конфигурациями, без дополнительных функций, таких как обнаружение сервисов или управление сетями.
2. **Зависимость от Spring:** Оптимизирован для использования в Spring-приложениях, что может ограничивать его применение в других технологиях.

Сравнительный анализ

Функциональные возможности:

- Consul предлагает наиболее широкий спектр возможностей, включая обнаружение сервисов, проверку их состояния и управление сетевой политикой.
- etcd специализируется на высокопроизводительном, согласованном хранении данных.
- Spring Cloud Config фокусируется на управлении конфигурациями в приложениях, интегрированных с Spring.

Простота использования:

- Spring Cloud Config является наиболее простым в использовании для разработчиков, работающих с экосистемой Spring.
- Consul и etcd требуют более сложной настройки и управления, особенно в крупных кластерах.

Производительность и масштабируемость:

- etcd демонстрирует высокую производительность и низкую задержку, что делает его предпочтительным для хранения критически важных данных.
- Consul и Spring Cloud Config могут иметь более высокие накладные расходы в зависимости от используемых функций и архитектуры.

1.3.2 Заключение

Выбор инструмента для управления конфигурациями и сервисами в распределённых системах зависит от конкретных требований проекта и архитектуры системы. Consul подходит для комплексных систем с необходимостью управления сетевой политикой и обнаружения сервисов. etcd является оптимальным выбором для высокопроизводительного и надёжного хранения данных. Spring Cloud Config идеален для централизованного управления конфигурациями в Spring-приложениях.

Глава 2. Практическая часть

2.1 Интеграция системы автоматической сборки Gradle

2.1.1 Организация проекта Gradle для проекта аналитики исторических данных

Структурирование исходного кода и процесса сборки программных проектов играют важную роль в обеспечении их читаемости и управляемости. На основе накопленного опыта были выработаны оптимальные методики, способствующие достижению этих целей. В данном разделе рассматриваются эффективные практики организации проектов, направленные на обеспечение их структурной ясности и удобства сопровождения. Также анализируются распространенные проблемы, с которыми сталкиваются разработчики, и предлагаются методы их предотвращения.

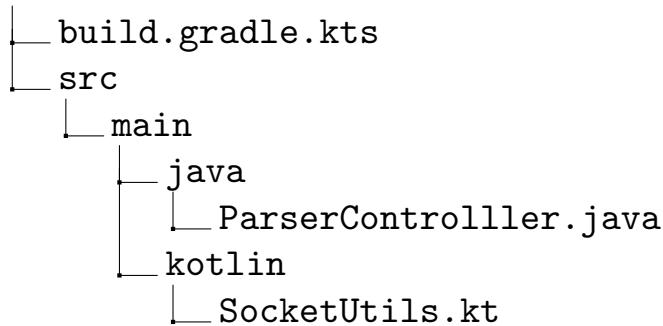
Разделение файлов исходного кода для определенных языков

Плагины для языка Gradle устанавливают определенные соглашения относительно организации и обработки исходного кода проекта. Например, при использовании языка программирования Java проект автоматически компилирует свой исходный код, расположенный в каталоге ‘src/main/java’. Аналогично, плагины для других языков следуют подобной схеме, определяя местоположение исходных файлов в соответствии с соответствующими соглашениями [13].

Некоторые компиляторы способны осуществлять кросс-компиляцию для нескольких языков, используя единый каталог для хранения исходных файлов. В контексте оптимизации производительности сборочных процессов, Gradle рекомендует строго соблюдать разделение исходных файлов по

языкам программирования, что облегчает как процесс сборки, так и обеспечивает четкость в предполагаемых структурах проекта [14].

Представленная структура исходного кода включает в себя отдельные каталоги для файлов на Java и Kotlin. Файлы на Java размещаются в каталоге `src/main/java`, в то время как файлы на Kotlin хранятся в каталоге `src/main/kotlin`.



Тестирование

В рамках программного проекта часто требуется проведение разнообразных видов тестирования, таких как модульные, интеграционные, функциональные и smoke тесты. Для эффективного управления исходным кодом каждого типа теста рекомендуется его хранение в специально выделенных каталогах. Практика размещения тестового кода в отдельных каталогах положительно сказывается на удобстве сопровождения проекта и разделении обязанностей, поскольку позволяет запускать различные типы тестов независимо друг от друга. Такой подход способствует повышению гибкости и эффективности процесса разработки, облегчая выявление и устранение дефектов в приложении [15].

В частности, мы добавляем плагин соглашения, чтобы ‘`buildSrc`’ разделить настройку интеграционного теста между несколькими подпроектами:

```

//...
val integrationTestTask = tasks.register<Test>("integrationTest")
{
    description = "Runs integration tests."
    group = "verification"
    useJUnitPlatform()
  
```

```

    testClassesDirs = integrationTest.output.classesDirs
    classpath = configurations[integrationTest.
        runtimeClasspathConfigurationName] + integrationTest.output

10     shouldRunAfter(tasks.test)
}
//...

```

Применение в проекте конкретного приложения:

```

dependencies {
    implementation(project(":list"))
}

```

Использование buildSrc для инкапсуляции императивной логики

Для абстрагирования императивной логики в процессе сборки рекомендуется использовать механизм ‘buildSrc‘. Комплексная логика сборки часто является целесообразным кандидатом для инкапсуляции в виде пользовательской задачи или двоичного плагина. Реализации таких пользовательских задач и плагинов не следует размещать внутри сценариев сборки. Использование ‘buildSrc‘ для этой цели является предпочтительным, поскольку это позволяет избежать повторного использования кода в различных независимых проектах.

Каталог ‘buildSrc‘ рассматривается как встроенный проект сборки. При обнаружении этого каталога Gradle автоматически компилирует и тестирует его содержимое, а затем включает скомпилированные классы в путь вашего сценария сборки. Для многопроектных сценариев сборки может быть только один каталог ‘buildSrc‘, который должен располагаться в корневом каталоге проекта. Важно отдавать предпочтение использованию плагинов сценариев, так как они обеспечивают более простое обслуживание, рефакторинг и тестирование кода [16].

2.1.2 Управление зависимостями и плагинами в Gradle

В проекте дополнительно используется механизм централизованного управления расширением зависимостей. Модуль позволяет указать источник версий зависимостей для каталога `libs`. В данном случае он указывает на файл `libs.versions.toml`, содержащий информацию о версиях зависимостей.

```

dependencyResolutionManagement {
    versionCatalogs {
        create("libs") {
            from(files("configuration/libs.versions.toml"))
        }
    }
}

include("conventions")

```

В качестве основного языка для Gradle был выбран Kotlin из-за схожести синтаксиса с Java. В данном случае, плагин `kotlin-dsl` включается для использования Kotlin DSL в качестве основного языка для написания сборочных скриптов в Gradle во всем проекте.

Плагин Spring Boot Gradle обеспечивает поддержку Spring Boot в Gradle. Он позволяет упаковывать исполняемые jar, запускать приложения Spring Boot и использовать управление зависимостями, предоставляемое `spring-boot-dependencies` [17].

```

plugins {
    kotlin("jvm") version "1.5.31"
    id("org.springframework.boot") version "2.6.4"
}
dependencies {
    implementation(libs.spring.boot.gradle.plugin)
}

```

В представленном кодовом отрывке, переменная `versionCatalog` получает доступ к расширению `VersionCatalogsExtension` и идентифицирует именованный каталог версий с названием «`libs`».

Метод `api(platform(project(":platform")))` добавляет зависимость от платформы проекта с именем «`:platform`». Это указывает на то, что весь проект будет использовать версии зависимостей, определенные в данной платформе.

```

val versionCatalog = extensions.getByName<VersionCatalogsExtension>()
    .named("libs")
dependencies {
    api(platform(project(":platform")))
5    versionCatalog.findLibrary("lombok").ifPresent {
        compileOnly(it)
        annotationProcessor(it)
    }
}

```

Такой подход способствует стандартизации и упрощает управление зависимостями, особенно в масштабируемых и многоуровневых проектах.

2.1.3 Создание пользовательских задач и скриптов в Gradle

В проекте используется несколько пользовательских задач, переопределяющих стандартное поведение Gradle.

Директива `manifest {...}` определяет настройки манифеста, который содержит метаданные о проекте. В данном контексте устанавливаются атрибуты манифеста, такие как ‘`Implementation-Title`’ (название реализации) и ‘`Implementation-Version`’ (версия реализации), которые присваиваются соответственно имени и версии проекта.

```

tasks.jar {
    manifest {
        attributes(
            mapOf(
5                "Implementation-Title" to project.name,
                "Implementation-Version" to project.
        )
    }
}

```

10 | }

2.1.4 Оптимизация процесса сборки с использованием Gradle

Эффективность процесса сборки играет решающую роль в производительности разработки. Увеличение времени сборки повышает риск прерывания рабочего процесса. Учитывая, что сборки запускаются многократно в течение дня, даже небольшие задержки могут существенно накапливаться [18].

Параллельное выполнение сборки

Так как проект состоит более чем из одного подпроекта линейная сборка занимает значительное время из-за не равносильности подпроектов. Так как из-за микросервисной архитектуры подпроекты независимы друг от друга, то они не имеют общего состояния. В таком случае есть смысл заставить Gradle выполнять сборку подпроектов параллельно.

Для этого в проекте используется флаг:

```
| org.gradle.parallel = true
```

Результаты оптимизации представлены на рисунке 6:

Прирост производительности автоматизированной сборки составил 21%.

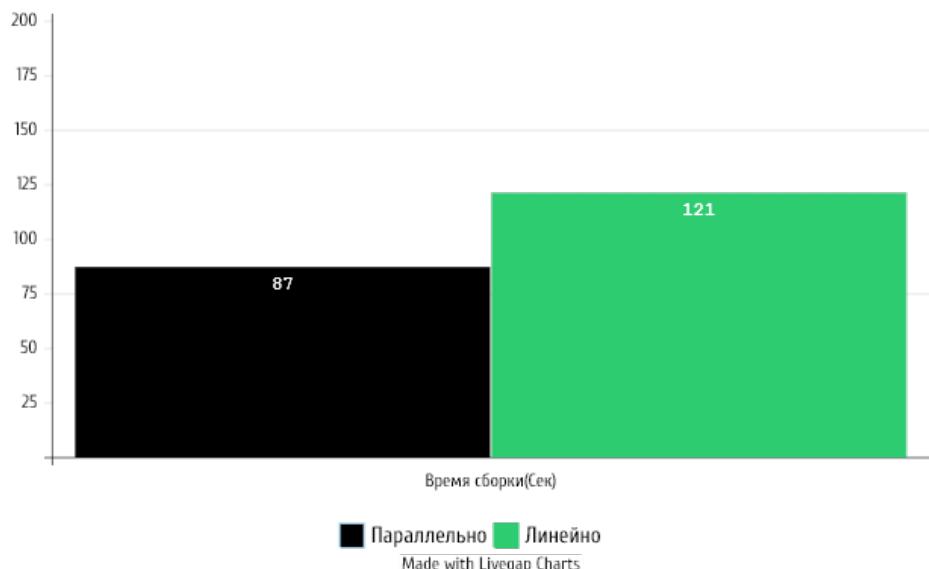


Рисунок 6 — Результат оптимизации

2.2 Внедрение API Gateway

2.2.1 Анализ требований и постановка задач

Проект начинается с определения основных требований к системе, включая:

- Обеспечение высокодоступного и отказоустойчивого доступа к данным.
- Масштабируемость для обработки большого объема запросов.
- Безопасность и защита данных.
- Гибкость маршрутизации и обработки запросов.

2.2.2 Основные компоненты архитектуры

Архитектура системы включает следующие основные компоненты:

- Spring Cloud Gateway: Центральный элемент для маршрутизации запросов и применения политик безопасности.

- Микросервисы: Каждый микросервис отвечает за конкретные функции, такие как обработка данных, управление пользователями, генерация отчетов и т.д.
- Базы данных: Хранение структурированных и неструктурированных данных о православных объектах, исторической информации и т.д.
- Интерфейсы пользователя: Веб-приложения и мобильные приложения для взаимодействия с пользователями.

2.2.3 Описание архитектуры проекта до внедрения Spring Cloud Gateway

До внедрения API gateway в проект аналитики исторических данных все запросы маршрутизировались напрямую каждому микросервису.

За собой это влекло ряд трудностей, которые требовали решения. Первостепенно - это вопрос безопасности. Прямой доступ значительно повышает вероятность потенциальных угроз. Дополнительно это усложняет соблюдение единой политики безопасности.

Следующей трудностью оказалось версионирование API. У каждого микросервиса есть свой эндоинт, в том числе даже в рамках одного сервиса существуют разные версии конечной точки. Возникала проблема доступа к разным версиям API.

Таким образом, отсутствие API Gateway приводило к множеству организационных и технических проблем, которые отрицательно сказывались на эффективности разработки, поддержке и масштабируемости системы.

Архитектурная схема работы проекта до внедрения выглядела следующим образом [7](#).

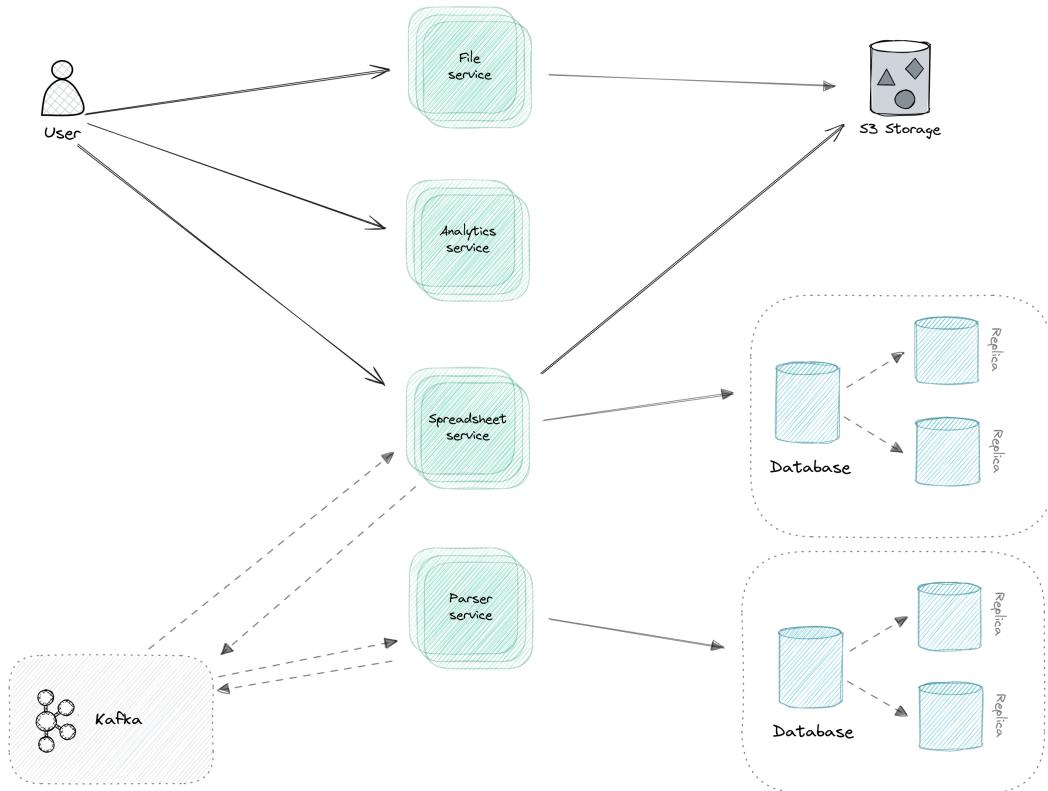


Рисунок 7 — Архитектура без фильтра

2.2.4 Проектирование архитектуры с использованием Spring Cloud Gateway

В данном разделе представлена концепция проектирования архитектуры информационно-аналитического онлайн-ресурса «Православный ландшафт таежной Сибири» с использованием Spring Cloud Gateway. Основная цель заключается в создании масштабируемой, надежной и безопасной системы для анализа и представления данных, связанных с православными объектами в таежных регионах Сибири, которая решает проблемы озвученные ранее.

В теоретической части описывались различные подходы к реализации. Исходя из этого была выбрана модель API Gateway.

Была спроектирована схема, которая представлена на рисунке 8.

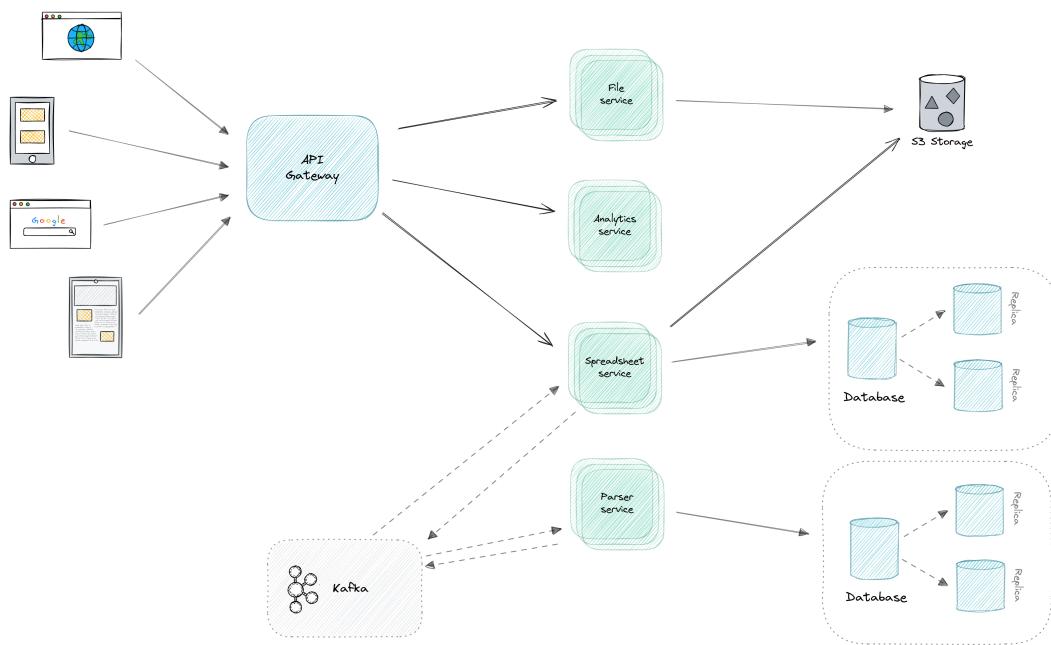


Рисунок 8 — Архитектура с фильтром

2.2.5 Реализация в проекте

Для начала необходимо сконфигурировать балансировщик нагрузки встроенный в Spring Cloud Gateway.

```

public class LoadBalancerConfiguration {

    @Bean
    ReactorLoadBalancer<ServiceInstance> roundRobinLoadBalancer(
5        Environment environment,
        LoadBalancerClientFactory loadBalancerClientFactory
    ) {
        String name = environment.getProperty(
LoadBalancerClientFactory.PROPERTY_NAME);
        return new RoundRobinLoadBalancer(
            loadBalancerClientFactory.getLazyProvider(name,
10           ServiceInstanceListSupplier.class),
            name
        );
    }
}

```

Балансировщик нагрузки RoundRobinLoadBalancer реализует алгоритм кругового перебора (Round Robin), распределяя входящие запросы равномерно между доступными экземплярами сервиса.

Таким образом, данный метод конфигурирует и возвращает балансировщик нагрузки, который использует алгоритм кругового перебора для распределения нагрузки между экземплярами сервиса в зависимости от имени сервиса, указанного в свойствах окружения.

Далее идет настройка маршрутизации и фильтрации.

```
//.....
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder
) {
    return builder.routes()
        .route(SIMPLE_RES_ID, r -> r
            .path(SIMPLE_RES_PATH)
            .filters(f -> f
                .rewritePath(API_PATH, ""))
        )
        .uri(loadBalancerUriFor(serviceConfig .
getSimple()))
    .
//.....
}
```

2.2.6 Внедрение SSL сертификатов

Подготовка SSL сертификата. Необходимо приобрести SSL сертификат у доверенного центра сертификации (СА) через Госуслуги можно бесплатно получить электронный сертификат безопасности — стандартный OV или базовый DV

Сертификат от Минцифры заменит иностранный в случае его отзыва или окончания срока действия, или создать самоподписанный сертификат. Для создания самоподписанного сертификата можно воспользоваться инструментами, такими как ‘keytool‘ или ‘openssl‘. Например, для создания сертификата с использованием ‘keytool‘:

```
keytool -genkeypair -alias myalias -keyalg RSA -keysize 2048 -storetype F
```

Этот процесс создаст файл хранилища ключей ‘keystore.p12‘, который будет использоваться в приложении.

Настройка Spring Boot для использования SSL сертификата. В проекте Spring Boot необходимо настроить файл конфигурации ‘application.yml‘ для использования SSL. Пример настройки в файле ‘application.yml‘:

```
server:
  port: 8443
  ssl:
    key-store: classpath:keystore.p12
    5      key-store-password: your_password
    key-store-type: PKCS12
    key-alias: myalias
```

В данной конфигурации указывается путь к хранилищу ключей, его пароль, тип хранилища и алиас ключа.

Обновление конфигураций клиента. Все клиентские приложения, обращающиеся к вашему API Gateway, должны быть обновлены для использования HTTPS вместо HTTP. Это включает изменение URL-адресов в конфигурационных файлах или коде приложений, а также проведение тестирования для проверки корректности работы.

Таким образом, выполнение вышеуказанных шагов позволяет успешно внедрить SSL конфигурации в Spring Cloud API Gateway, обеспечивая безопасность данных и защищенные соединения между клиентами и сервером.

2.3 Проектирование архитектуры проекта с учетом сервера конфигурации

Проектирование архитектуры сервера конфигурации позволяет централизованно управлять конфигурационными данными, обеспечивая их консистентность и безопасность. Это упрощает администрирование, повышает надежность системы и позволяет быстро адаптироваться к изменениям без необходимости перезапуска приложений.

2.3.1 Схема взаимодействия компонентов

В ходе работы была спроектированная схема, представленная на рисунке 9

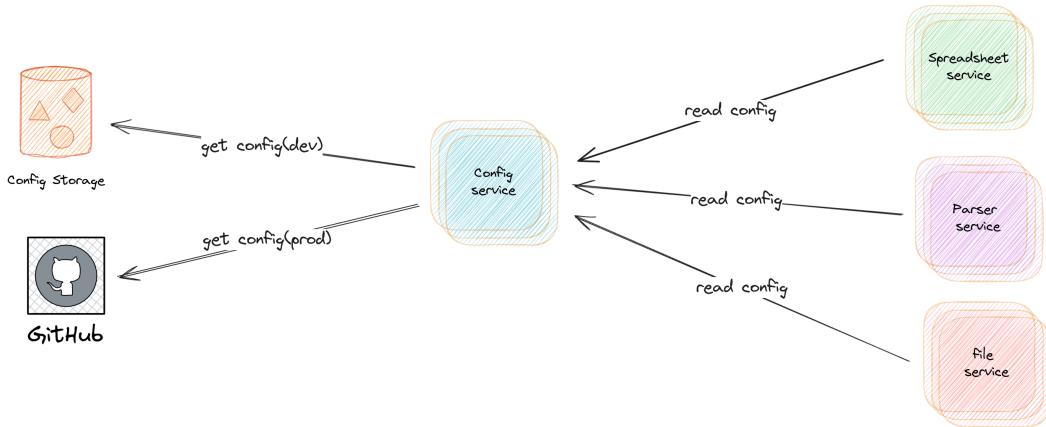


Рисунок 9 – Config service

Благодаря такому подходу мы можем разделить конфигурации для среды разработки приложения и настоящей.

2.3.2 Безопасность и контроль доступа

Spring Cloud Config предоставляет возможности для реализации различных механизмов безопасности и контроля доступа, которые обеспечивают защиту конфигурационных данных от несанкционированного доступа и изменений. Аутентификация пользователей и сервисов, получающих доступ к Spring Cloud Config, является первым шагом в обеспечении безопасности. Spring Cloud Config поддерживает различные методы аутентификации. Он из «коробки» интегрируется с различными провайдерами OAuth2, такими как Auth0, Okta, Keycloak.

Защита конфиденциальных конфигурационных данных, таких как учетные данные и ключи API, осуществляется с помощью шифрования. Spring Cloud Config поддерживает шифрование конфиденциальных значений в конфигурационных файлах. Для этого используются такие инструменты, как JCE (Java Cryptography Extension) и внешние сервисы

шифрования (например, HashiCorp Vault). Использование TLS (Transport Layer Security) для шифрования данных при передаче между клиентами и сервером Spring Cloud Config, что защищает данные от перехвата и манипуляций.

2.3.3 Репликация и отказоустойчивость

Для обеспечения отказоустойчивости Spring Cloud Config Service следует применять несколько стратегий: Разворачивание нескольких экземпляров Spring Cloud Config Server в кластерной конфигурации с использованием балансировщика нагрузки, который будет распределять запросы между доступными экземплярами. Это обеспечивает устойчивость к сбоям отдельных узлов. Использование нескольких зеркальных копий конфигурационных хранилищ (Git-репозиториев или баз данных) с автоматическим переключением на доступный репозиторий в случае сбоя основного хранилища. Локальное кеширование конфигурационных данных на уровне клиентских приложений с использованием механизма Spring Cloud Config Client. В случае недоступности конфигурационного сервера клиент может временно использовать закешированные данные.

2.3.4 Интеграция сервера конфигурации с микросервисами

2.3.5 Настройка подключения микросервисов к серверу конфигурации

Интеграция сервиса конфигурации прошла успешно. Для этого были приняты несколько шагов. В целях безопасности далее реальные наименования и данные будут заменены на условные.

Реализация EnvironmentRepository по умолчанию использует серверную часть Git, что чрезвычайно удобно для управления обновлениями

и физическими средами, а также для аудита изменений. Чтобы изменить расположение репозитория, можно настроить свойство конфигурации `spring.cloud.config.server.git.uri` на сервере конфигурации, например, в файле `application.yml`. При установке этого свойства с префиксом `file:` репозиторий будет работать из локального хранилища, что позволяет быстро и легко начать работу без необходимости запуска сервера. В данном случае сервер будет работать непосредственно с локальным репозиторием, не клонируя его, что не зависит от его публичного статуса, поскольку сервер конфигурации никогда не изменяет ‘удалённый’ репозиторий.

SSL сертификация

В зависимости от назначения развертывания используются разные параметры для SSL сертификатов.

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://www.ru/my/git_repo
          skipSslValidation: true/false
```

Spring Cloud Config Server поддерживает URL-адрес репозитория `git` с заполнителями для параметров `{application}` и `{profile}`. Это позволяет реализовать политику «один репозиторий на приложение», используя структуру, аналогичную следующей.

```
uri: https://github.com/myorg/{application}
```

Безопасность

Для использования базовой аутентификации HTTP в удалённом репозитории необходимо отдельно указать свойства имени пользователя и пароля, а не включать их в URL. Пример конфигурации представлен ниже:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/config-repo
          username: user
          password: user
```

Если не используется HTTPS и учётные данные пользователя, можно использовать SSH, при условии, что ключи хранятся в стандартных директориях (`~/.ssh`), а URI указывает на SSH-адрес, например, `<git@github.com>: configuration/cloud-configuration`. Важно, чтобы в файле `~/.ssh/known_hosts` содержалась запись для Git-сервера в формате ssh-rsa, так как другие форматы (например, ecdsa-sha2-nistp256) не поддерживаются. Для избежания непредвиденных проблем следует убедиться, что в файле `known_hosts` присутствует только одна запись для Git-сервера, соответствующая указанному в конфигурационном сервере URL. Если в URL используется имя хоста, оно должно точно совпадать с записью в `known_hosts`, а не с IP-адресом. Репозиторий доступен через JGit. Настройки HTTPS-прокси могут быть заданы в файле `~/.git/config` или с помощью системных свойств JVM (`-Dhttps.proxyHost` и `-Dhttps.proxyPort`).

Если удалённые источники свойств содержат зашифрованное содержимое (значения, начинающиеся с `{cipher}`), они расшифровываются перед отправкой клиентам по HTTP. Основным преимуществом данной настройки является то, что значения свойств не обязательно должны быть в открытом виде, когда они находятся «в покое» (например, в репозитории Git). Если значение не может быть расшифровано, оно удаляется из источника свойств, и добавляется дополнительное свойство с тем же ключом,

но с префиксом `invalid` и значением, означающим «не применимо» (обычно `<н/а>`). Это делается в основном для предотвращения случайного использования зашифрованного текста в качестве пароля и его утечки.

Если настроен удалённый репозиторий конфигурации для клиентских приложений конфигурации, он может содержать файл `application.yml`, аналогичный следующему:

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}123123ASDASDQWEQWE'
```

Эти значения можно безопасно отправить в общий репозиторий Git, и секретный пароль останется защищённым.

Сам же доступ к конфигурации можно получить только имея необходимые права и скопы для протокола OAuth 2.0.

Перспективы развития

В перспективе развития данной технологии возможно углубление интеграции с современными инструментами управления конфигурациями, такими как Vault, а также расширение функциональности для обеспечения более гибкого управления версиями конфигураций и автоматизации процессов развертывания и обновления приложений. Не исключается вариант перехода на Etcd.

Заключение

Основные результаты работы заключаются в следующем.

1. Проанализированы подходы к проектированию и внедрению микросервисов.
2. Разработаны архитектурные схемы, с учетом требуемых компонентов.
3. Проведена интеграция нововведений в существующую систему

В заключение автор выражает благодарность и большую признательность научному руководителю Болдовской Т. Е. за поддержку, помощь, обсуждение результатов и научное руководство. Проект существует при поддержки Российского научного фонда Грант номер «23-78-10119».

Список литературы

1. *Болдовская, Т. ИСТОРИЧЕСКАЯ ИММОРТАЛИЗАЦИЯ: ЭФФЕКТИВНЫЕ МЕТОДЫ ДОЛГОСРОЧНОГО ХРАНЕНИЯ ИСТОРИЧЕСКИХ ДАННЫХ* [Текст] / Т. Болдовская, В. Гресь, А. Ветров // Математические структуры и моделирование. — 2023. — 4 (68). — С. 85—92.
2. *Prakash, M. Software Build Automation Tools a Comparative Study between Maven, Gradle, Bazel and Ant* [Текст] / M. Prakash // International Journal of Software Engineering & Applications. DOI: <https://doi.org/10.5121/ijsea>. — 2022.
3. *Antony, A. A Detailed Comparison of Maven vs Gradle* [Текст] / A. Antony. — URL: [urhttps://medium.com/javarevisited/a-detailed-comparison-of-maven-vs-gradle-9c8c444c057c](https://medium.com/javarevisited/a-detailed-comparison-of-maven-vs-gradle-9c8c444c057c).
4. *Trebichavský, R. API Gateways and Microservice Architectures* [Текст] / R. Trebichavský. — 2021.
5. *Microsoft. The API gateway pattern versus the Direct client-to-microservice communication* [Текст] / Microsoft. — URL: [urhttps://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice - container - applications / direct - client - to - microservice - communication-versus-the-api-gateway-pattern](https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice - container - applications / direct - client - to - microservice - communication-versus-the-api-gateway-pattern).
6. *Zhao, J. Management of API gateway based on micro-service architecture* [Текст] / J. Zhao, S. Jing, L. Jiang // Journal of Physics: Conference Series. T. 1087. — IOP Publishing. 2018. — C. 032032.
7. *Alkhodary, S. The Evaluation of Using Backend-For-Frontend in a Microservices Environment* [Текст] / S. Alkhodary // Accessed: Oct. — 2023. — T. 6.
8. *Newman, S. Building Mircoservices* [Текст] / S. Newman. — O'reilly, 2015. — C. 600.
9. *Carnell, J. Spring microservices in action* [Текст] / J. Carnell, I. H. Sánchez. — Simon, Schuster, 2021.

10. *Spring.* Spring Cloud Gateway [Текст] / Spring. — URL: <https://cloud.spring.io/spring-cloud-gateway>.
11. *Richardson, C.* Pattern: Circuit Breaker [Текст] / C. Richardson. — URL: <https://microservices.io/patterns/reliability/circuit-breaker.html>.
12. *Richardson, C.* Microservices patterns: with examples in Java [Текст] / C. Richardson. — Simon, Schuster, 2018.
13. *Gradle.* java plugin [Текст] / Gradle. — URL: https://docs.gradle.org/current/userguide/java_plugin.html#java_plugin.
14. *Gradle.* Gradle project [Текст] / Gradle. — URL: https://docs.gradle.org/current/userguide/organizing_gradle_projects.html.
15. *Gradle.* Testing gradle project [Текст] / Gradle. — URL: https://docs.gradle.org/current/samples/sample_jvm_multi_project_with_additional_test_types.html.
16. *Gradle.* Sharing build logic between subprojects Sample [Текст] / Gradle. — URL: https://docs.gradle.org/current/samples/sample_convention_plugins.html.
17. *Spring.* Справочное руководство по плагину Spring Boot Gradle [Текст] / Spring. — URL: <https://docs.spring.io/spring-boot/docs/current/gradle-plugin/reference/htmlsingle/>.
18. *Gradle.* Gradle Perfomance [Текст] / Gradle. — URL: <https://docs.gradle.org/current/userguide/performance.html>.