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)

Информационные системы и технологии корпоративного управления
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему Исследование методов и инструментов организации управления прикладным программным интерфейсом микросервисов

Студент Д. А. Ольхов
(И.О. Фамилия) (личная подпись)

Научный руководитель канд. пед. наук Е.В. Панюкова
(ученая степень, звание, И.О. Фамилия)

Тольятти 2021

Оглавление

Введение

В данной работе поднимается проблема управления API микросервисов в облачных средах, а также выдвигается гипотеза о возможности его значительного упрощения при правильно спроектированном архитектурном подходе. Данная проблема поднимается с точки зрения не только разработчика ПО, но и человека, управляющего проектом и отвечающего за его архитектуру.

Проблема управления API является частью проблем, возникающих при использовании микросервисной архитектуры. Эта архитектура на данный момент является передовой технологией разработки корпоративных приложений. При проектировании программного продукта с использованием микросервисной архитектуры остро возникает вопрос публикации интерфейса, с целью скрыть конечную реализацию и предоставить возможность замены реализации и гибкость в распределении ролей и функций микросервисов. Эту проблему решают специальные инструменты управления API. В дополнение к этому эти инструменты могут брать на себя обязательства по обеспечению безопасности коммуникации, предоставлять возможность отслеживать и собирать статистику использования API. Перед разработчиками и архитекторами стоят проблемы, решение которых ещё только предстоит найти, это обуславливает **актуальность** темы исследования.

Объектом исследования является внешняя коммуникация с облачным решением, а также внутренняя микросервисная коммуникация.

Предметом исследования является методы и инструменты организации коммуникации с облачным решением и микросервисной коммуникации внутри облачной среды.

Целью исследования является теоретическое обоснование и практическая реализация методов и инструментов управления API.

Для достижения поставленной цели требуется решить следующие задачи:

– проанализировать подходы к организации управления API и микросервисной коммуникации;
– проанализировать способы реализации инструментов организации управления API и микросервисной коммуникации;
– разработать модель инструмента организации управления API;
– подтвердить эффективность предлагаемой модели на практике.

Гипотеза исследования: предположение, что разработанная модель инструмента управления API и организацию микросервисной коммуникации, которая обеспечит понижение стоимости разработки приложений с применением принципов микросервисной архитектуры.

На защиту выносятся:

– модель инструмента организации управления API облачного приложения;
– результаты апробации инструмента организации управления API облачного приложения.

Научная новизна исследования заключается в разработке модели инструмента, предоставляющего возможности простого управления API микросервисов и производящий автоматическую настройку микросервисного взаимодействия.

Практическая значимость работы заключается в разработке инструмента управления API в облачных средах, который снизит стоимость и время разработки облачных приложений.

Методы исследования: методы и модели развертывания компонентов инструмента управления API, объектно-ориентированных подход к анализу и проектированию инструмента.

Основные этапы исследования: исследование проводилось с 2018 по 2021 год в несколько этапов:

– анализ подходов к организации управления API и микросервисной коммуникации;
– анализ способов реализации инструментов организации управления API и микросервисной коммуникации;
– разработка модели инструмента организации управления API;
– разработка компонентов инструмента;
– анализ результатов внедрения инструмента.

Первая глава посвящена анализу архитектуры облачных программных решений, а также рассматриваются разные подходы к управлению API в облачной среде, поднимается проблема управления API, возникающая в условиях использования облачных сред и микросервисных архитектур.

Во второй главе рассмотрены существующие методы управления API с их преимуществами и недостатками. Рассмотрены методы развертывания микросервисов в облачной среде. А также в этой главе описывается концептуальная архитектура инструмента управления API и микросервисной коммуникации.

Третья глава содержит этапы проектирования и разработки инструмента управления API и микросервисной коммуникации.

Четвертая глава описывает экспериментальную апробацию инструмента в облачной среде, а также приведен расчет надежности инструмента.

В заключении подводятся итоги выполненной работы.

Работа изложена на 88 страницах и включает 38 рисунков, 3 таблицы.

Глава 1 Анализ методов разработки микросервисов

1.1 Виртуализация

Облачные вычисления построены на виртуализации и благодаря ей имеют преимущества в виде гибкости, доступности и изолированности. В свою очередь виртуализация берет своё начало с 60-х годов прошлого века с первых виртуальных машин, созданных компанией IBM как способ обеспечить большому количеству пользователей совместный доступ к большим дорогостоящим мейнфреймам. В то время IBM занималась развитием виртуализации платформ. Пристальное внимание и интерес к виртуализации возник в начале XXI веке благодаря росту возможностей вычислительной техники. Производительность компьютеров стала гораздо больше и дешевле и сделало выгодным и практичным использование технологий виртуализации.

Виртуализация – это технология создания виртуальных ресурсов, абстрагированных от аппаратной части компьютера специальным слоем. В качестве виртуального ресурса могут выступать сервера, персональный компьютер, операционная система, файлы, хранилища или сетевая инфраструктура. Основная цель виртуализации – управление вычислительной нагрузкой за счет трансформации аппаратных ресурсов в виртуальные, что открывает возможности для ее распределения между большим количеством пользователей. Технологии виртуализации достаточно часто используется для создания виртуальных машин, в дальнейшем ВМ. ВМ имеет преимущества и недостатки в сравнении с традиционным подходом к организации вычислительных ресурсов.

Преимущества:

– возможность работать с устаревшими программными решениями и операционными системами;

– возможность создать защищённые пользовательские окружения для работы с сетью интернет и не защищёнными сетями, в этом случае внешние атаки могут нанести вред виртуальной машине, а не операционной системе;
– несколько виртуальных машин, развернутых на аппаратных ресурсах одного компьютера, изолированы друг от друга, таким образом, сбой одной из виртуальных машин не повлияет на доступность и работоспособность сервисов и приложений всех остальных;
– благодаря тому, что каждая виртуальная машина представляет собой программный контейнер, она может быть перенесена или скопирована, как и любой иной файл;
– виртуальные машины не зависят от аппаратного обеспечения, на котором они работают в том смысле, что в качестве значений параметров виртуальной машины, таких как оперативная память, процессор и т.п., можно указать значения и типы, отличающиеся от реальной аппаратной конфигурации компьютера;
– виртуальные машины идеально подходят для процессов обучения и переподготовки, поскольку позволяют развернуть требуемую платформу вне зависимости от параметров и программного обеспечения хост-машины (физического компьютера, на котором функционирует виртуальная машина);
– возможность сохранения состояния виртуальной машины (так называемый слепок или snapshot) позволяет быстро вернуться к точке до внесения изменений в систему, а также создавать резервные копии;
– в рамках одной хост-машины может быть развернуто несколько гостевых операционных систем, объединенных в сеть и взаимодействующих между собой;

– виртуальные машины могут создавать представления устройств, которых физически нет (эмуляция устройств).

Недостатки:

– обеспечение единовременной работы нескольких виртуальных машин требует достаточного количества аппаратных мощностей;
– в зависимости от используемого решения, операционная система виртуальной машины может работать медленнее, чем на «чистом» аналогичном аппаратном обеспечении;
– различные платформы виртуализации не поддерживают виртуализацию всего аппаратного обеспечения и интерфейсов.

Виртуализация бывает разных типов, здесь мы рассмотрим основные.

Виртуализация операционной системы.

Наиболее распространённый тип виртуализации на данный момент. Представляет собой одну или более гостевых ОС, запущенных на хост-машине с определенной хост-ОС.

Виртуализация приложений.

Под этим типом виртуализации понимается использование программных решений в рамках изолированной виртуальной среды.

Виртуализация сети.

Этот тип виртуализации представляет собой процесс объединения аппаратных и программных ресурсов в единую виртуальную сеть. Такая виртуализация разделяется на внешнюю – объединяющую множество сетей в одну виртуальную и внутреннюю – создающую виртуальную сеть между виртуальными машинами или контейнерами внутри одной системы.

Виртуализация аппаратного обеспечения.

Разбиение компонент аппаратного обеспечения на сегменты, управляемые отдельно друг от друга. А также полная виртуализация аппаратной архитектуры всей платформы.

Виртуализация систем хранения.

Делится на два типа: виртуализацию блоков и виртуализацию файлов. Виртуализация файлов как правило используется в системах хранения, при этом ведутся записи о том, какие файлы и каталоги на каких носителях находятся. Виртуализация блоков используется в сетях распределенного хранения данных на примере технологий RAID и iSCSI.

Рассматривая виртуализацию операционной системы, стоит выделить основные способы виртуализации.

Программная виртуализация, характеризующая полностью программной эмуляцией аппаратных ресурсов, драйверов и т. п. В свою очередь делится на динамическую трансляцию и паравиртуализацию. Суть динамической трансляции в преобразовании команд непосредственно во время работы гостевой ОС, при этом проблемные команды гостевой операционной системы перехватываются программным гипервизором. Паравиртуализация характеризуется взаимодействием гостевой ОС с программным гипервизором посредством специального API. Минусом такого подхода является необходимая адаптация ядра гостевой ОС, что не всегда возможно ввиду закрытости кода или других лицензионных ограничений.

Аппаратная виртуализация. В этой технологии гипервизор реализован в аппаратной части вычислительной машине, а именно в центральном процессоре. Это позволяет гостевым ОС быть управляемыми напрямую гипервизором, а также возможна их полная изоляция друг от друга.

Контейнеризация или виртуализация на уровне операционной системы. Самая современная технология на данный момент в отличии от описанных выше суть её в том, что ядро операционной системы берет на себя обязательства по изоляции. Другими словами, ядро поддерживает несколько изолированных экземпляров пространства пользователя вместо одного. Эти экземпляры называются контейнерами (иногда зонами) полностью соответствуют отдельному экземпляру операционной системы. Явным преимуществом данного подхода является отсутствие дополнительных

расходов на эмуляцию виртуального оборудования и запуск полноценного экземпляра ОС. Однако недостаток данной технологии в ограничении, при котором в контейнере может быть запущен экземпляр операционной систем только с точно таким же ядром что и у хост-машины из-за того, что используется одно ядро ОС на все контейнеры.

1.2 Модели обслуживания облачных вычислений. PaaS решения

Облачные вычисления – это способ предоставления системных ресурсов компьютера по требованию без активного участия пользователя в их управлении. Частные случаи таких ресурсов – хранилище данных (облачное) и вычислительные мощности. Этот термин в общем смысле подходит для описания дата-центров, доступных множеству пользователям посредством сети интернет. Облачные вычисления позволили рынку информационных технологий перейти на модель «плати сколько можешь и пользуйся» за счет удешевления стоимости часа используемой вычислительной мощности, а также стоимости одного гигабайта хранилища.

Национальным Институт стандартов и технологий США определены следующие пять обязательных характеристик облачных вычислений.

Самообслуживание по требованию (On-demand self-service).

Потребитель может самостоятельно определять свои вычислительные возможности, такие как: серверное время, скорости доступа и обработки данных, объём хранимых данных – без непосредственного взаимодействия с представителем поставщика услуг.

Широкий сетевой доступ (Broad network access).

Услуги доступны потребителям по сети передачи данных вне зависимости от используемого терминального устройства.

Объединение ресурсов (Resource pooling).

Поставщик услуг объединяет ресурсы в единый пул для обслуживания большого числа потребителей для динамического перераспределения мощностей между потребителями в условиях постоянного изменения спроса на мощности; при этом потребители контролируют только основные параметры услуги (например, объём данных, скорость доступа), но фактическое распределение ресурсов, предоставляемых потребителю, осуществляет поставщик (в некоторых случаях потребители всё-таки могут управлять некоторыми физическими параметрами перераспределения, например, указывать желаемый центр обработки данных из соображений географической близости).

Высокий уровень эластичности (Rapid elasticity).

Возможность быстрого расширения, или уменьшения ресурсных возможностей потребителем в любой момент времени, без дополнительных издержек на взаимодействие с поставщиком, как правило, в автоматическом режиме;

Учёт потребления (Measured service).

Поставщик услуг автоматически исчисляет потреблённые ресурсы на определённом уровне абстракции (например, объём хранимых данных, пропускная способность, количество пользователей, количество транзакций) и на основе этих данных оценивает объём предоставленных потребителям услуг.

Облачные вычисления предоставляются потребителям в разных формах. Моделей обслуживания всего три основных, хотя в рамках конкретных бизнес-потребностей модели могут принимать подвиды.

Первой и часто используемой заинтересованным в конкретном решении проблемы бизнесом моделью является «Программное обеспечение как услуга» (SaaS, Software-as-a-Service).

Это модель, в которой потребителю предоставляется возможность использования прикладного программного обеспечения провайдера,

работающего в облачной инфраструктуре и доступного из различных клиентских устройств или посредством тонкого клиента, например, из браузера (например, веб-почта) или посредством интерфейса программы.

Контроль и управление основной физической и виртуальной инфраструктурой облака, в том числе сети, серверов, операционных систем, хранения, или даже индивидуальных возможностей приложения (за исключением ограниченного набора пользовательских настроек конфигурации приложения) осуществляется облачным провайдером.

Второй и популярной среди разработчиков решений корпоративного уровня является модель «Платформа как услуга» (PaaS, Platform-as-a-Service).

Это модель, когда потребителю предоставляется возможность использования облачной инфраструктуры для размещения базового программного обеспечения для последующего размещения на нём новых или существующих приложений (собственных, разработанных на заказ или приобретённых тиражируемых приложений).

В состав таких платформ входят инструментальные средства создания, тестирования и выполнения прикладного программного обеспечения — системы управления базами данных, связующее программное обеспечение, среды исполнения языков программирования — предоставляемые облачным провайдером.

Контроль и управление основной физической и виртуальной инфраструктурой облака, в том числе сети, серверов, операционных систем, хранения осуществляется облачным провайдером, за исключением разработанных или установленных приложений, а также, по возможности, параметров конфигурации среды (платформы).

Последней моделью предоставления облачных вычислений является модель «Инфраструктура как услуга» (IaaS, Infrastructure-as-a-Service).

Эта модель предоставляется как возможность использования облачной инфраструктуры для самостоятельного управления ресурсами обработки,

хранения, сетями и другими фундаментальными вычислительными ресурсами, например, потребитель может устанавливать и запускать произвольное программное обеспечение, которое может включать в себя операционные системы, платформенное и прикладное программное обеспечение.

Потребитель может контролировать операционные системы, виртуальные системы хранения данных и установленные приложения, а также обладать ограниченным контролем за набором доступных сетевых сервисов (например, межсетевым экраном, DNS).

Контроль и управление основной физической и виртуальной инфраструктурой облака, в том числе сети, серверов, типов используемых операционных систем, систем хранения осуществляется облачным провайдером.

Рисунок 1 – Модели компьютерных вычислений в виде слоев

Все эти модели выстраиваются в стройный ряд и являются слоями над физическими вычислительными ресурсами, так модель SaaS включает в себя все слои ниже, с той лишь разницей что для клиентов облака остальные слои

скрыты и взаимодействие с ними происходит только лишь через слой SaaS (см. Рисунок 1).

В рамках научно-исследовательской работы наше внимание будет приковано больше к модели PaaS несмотря на то, что наше конечное решение будет применяться именно в SaaS слое. Именно на PaaS слое разрабатываются микросервисы и формируется бизнес логика. Другими словами, именно на PaaS слое формируется слой выше (SaaS), который предоставляет уже конкретные услуги пользователям. То, что наше решение будет предоставлять новые возможности непосредственно разработчикам микросервисов указывает на его принадлежность продуктовым решениям. И если попробовать расположить результат научно-исследовательской работы на рис. 1, то он будет располагаться где-то между PaaS и SaaS. Именно поэтому в научно-исследовательской работе будут использоваться широкие возможности, предоставляемые именно PaaS моделью.

К более конкретным программным продуктам, реализующим модель PaaS является: Openshift, Kubernetes, Docker, Docker Swarm. Разработанный инструмент будет частью платформы и должен иметь возможность легко разворачиваться в любой из этих сред. Этого легко добиться, так как Openshift платформа строиться на Kubernetess, а Kubernetess в свою очередь построен на основе Docker. Docker Swarm является аналогом Kubernetess, только с более ограниченными возможностями управления запущенным приложением в виртуальной среде. Выбор на эти продукты пал во многом благодаря их популярности. За счёт удачности решения эти программные продукты используются во многих компаниях, разрабатывающих облачные программные продукты и на данный момент, являются практически стандартом облачной разработки. Такие крупные провайдеры облачных платформ как Amazon Web Services, Google Cloud, Microsoft Azure, Alibaba Cloud, IBM Cloud предоставляют возможность развернуть любой из приведенных выше продуктов, что также говорит в их пользу. Благодаря

осознанному выбору адаптировать продукт под эти платформы шанс на успешность, популярность и частоту использования этого продукта повышается. Опишем вкратце каждый из продуктов.

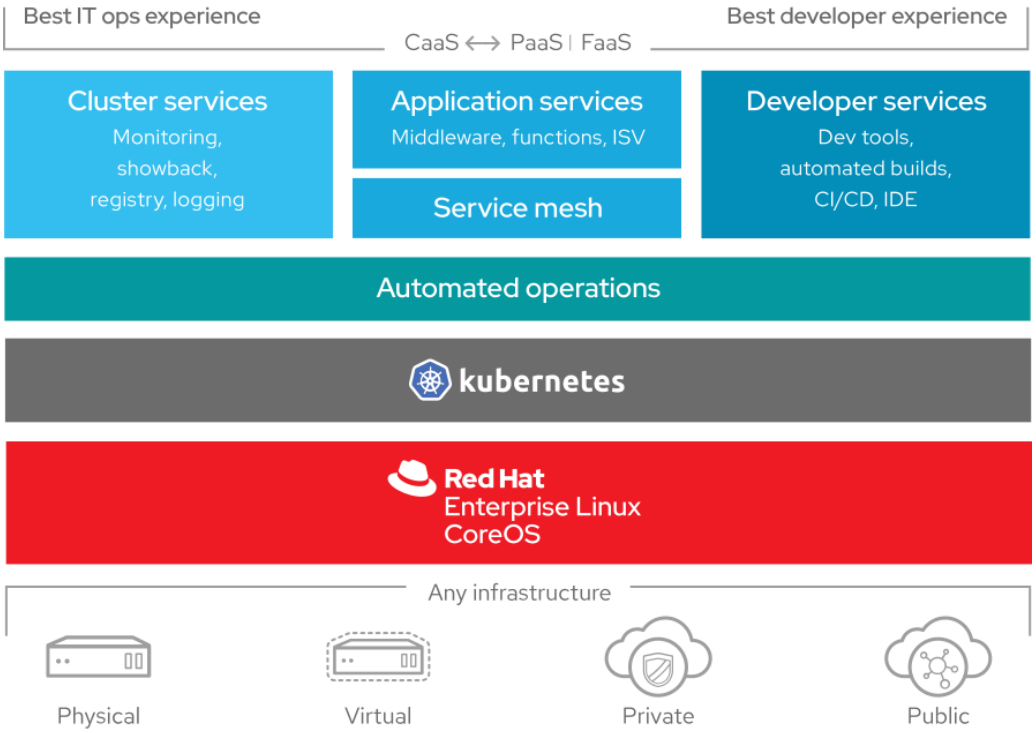

Рисунок 2 – Структура Openshift

Docker – это программное обеспечение, созданное с целью сделать процессы создания, развертывания и запуска приложений проще, используя контейнеризацию. Контейнеры позволяют разработчикам «запаковать» приложение со всеми необходимыми частями, такими как зависимости, библиотеки, среды выполнения и с лёгкостью развернуть его как самостоятельный пакет.

Kubernetes – это портативная, расширяемая платформа с открытым исходным кодом для управления контейнерами с рабочей нагрузкой и сервисами, что облегчает декларативную конфигурация и автоматизацию. Kubernetes предоставляет широкий набор функций: балансировка нагрузки и сервис обнаружений, управление хранилищами, автоматический откат и выкат изменений, бинарная авто-упаковка контейнеров, сервис

самовосстановления, управление конфигурациями и секретами. Стоит отметить, что сам по себе Kubernetes не является полноценным PaaS решением, а только лишь предоставляет набор строительных блоков (платформу) для построения таких решений.

Openshift – это программный продукт реализующий модель PaaS облачных вычислений. Вся экосистема продукта построена на основе оркестрации контейнерами при помощи Docker, управление сервисами и приложениями за счет Kubernetes и операционной системы Red Hat Enterprise Linux.

1.3 Микросервисы. Микросервисная архитектура

Так как целью научно-исследовательской работы является теоретическое обоснование, и практическая реализация модели автоматизированного управления прикладным программным интерфейсом и конфигурации сети микросервисов то стоит остановится на том, что такое микросервис, в чем заключается микросервисная архитектура приложений.

1.3.1 Монолитная архитектура

Чтобы лучше понять, чем являются микросервисы стоит рассмотреть архитектурные стили, используемые до появления микросервисов и то, как архитектура корпоративных приложений эволюционировала в микросервисную. Лучший способ понять ключевые моменты и характеристики микросервисов это детально рассмотреть эволюцию архитектур корпоративных систем из монолитных приложений в микросервисы.

Программные приложения уровня предприятия проектируются чтобы облегчить реализацию многочисленных бизнес требований.

В архитектуре монолита все бизнес функциональности собраны внутри одного монолитного приложения и упакованы в один единственный модуль. Чтобы улучшить понимание приведем реальный пример монолитного приложения, выберем в этом качестве интернет-магазин (см. Рисунок 3)

В целом интернет-магазин представляет собой набор нескольких компонентов, таких как управление заказами, платежи, управление продуктом и так далее.

Каждый из этих компонентов предлагает широкий набор бизнес-функций.

Добавление или модификация функциональных возможностей в компонент очень дорого из-за его монолитной природы.

Также, чтобы покрыть все бизнес требования эти компоненты должны обмениваться между собой данными.

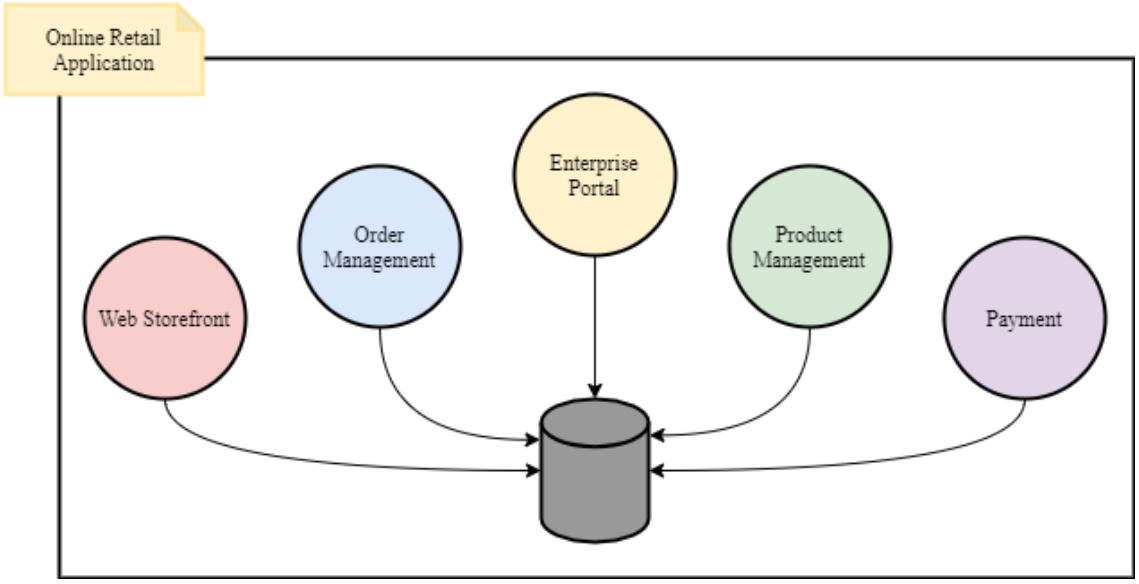

Рисунок 3 – Приложение интернет-магазина, разработанное в монолитной архитектуре

Коммуникация между компонента часто строится на проприетарных протоколах и стандартах и основаны на схеме взаимодействия точка-точка.

Поэтому изменение или замена данного компонента также связано с большими трудностями.

Для примера если магазин хотел бы использовать другую систему управления заказами, то осуществление такого требования понесло за собой огромное количество изменений в нескольких существующих компонентах. Подытожим основные характеристики монолитных приложений:

- проектируются, реализуются и разворачиваются как единственный модуль/пакет;
- подавляющая сложность реализаций большинства реальных бизнес сценариев, что превращает обслуживание, обновление и добавление новых функций в ночной кошмар;
- с трудом подходит для практики методологий гибкой разработки и доставки приложений заказчику. Так как приложение собирается в один большой модуль, то большинство бизнес-функций не имеет своего выделенного жизненного цикла;
- с ростом монолитного приложения растет время его запуска и стоимость такого запуска;
- один нестабильный сервис может вывести из строя все приложение;
- в случае необходимость может быть масштабирован только как полноценное приложение (нет возможности сделать это по частям), что сделать сложно в случаях неоднородного требования компонентов к ресурсам. Например, одна бизнес функциональность требует больше времени CPU, в то время как другой необходимо больше памяти. В таком случае очень сложно удовлетворить индивидуальные потребности каждого компонента;
- миграция на новые технологии и библиотеки затруднительна, так как монолитная структура подразумевает использование единой технологии.

В качестве решения некоторых из ограничений монолитной архитектуры приложений появилась «Сервисно-ориентированная архитектура» (Service Oriented Architecture, SOA) и «Сервисная шина предприятия» (Enterprise Service Bus, ESB).

1.3.2 Сервисно-ориентированная архитектура и сервисная шина предприятия

Сервисно-ориентированная архитектура была создана чтобы справиться с проблемами монолитных приложений путем разделения функциональностей монолитного приложения на переиспользуемые и слабо связанные сущности, называемые сервисами.

Доступ к каждому из сервисов осуществляется посредством вызовов через сеть.

- сервис — это автономная реализация четко описанного бизнес функционала, которая доступна по сети. Приложения в сервисно-ориентированной архитектуре основаны на сервисах;
- сервисы являются частью приложения с детерминированными интерфейсами, не зависящими от реализации. Четкое разделение интерфейсов сервисов от их реализации является важным аспектом сервисно-ориентированной архитектуры;
- потребителям важен только интерфейс сервиса, но не его реализация;
- сервисы автономны (и выполняют заранее определенные задачи) и слабо связаны;
- сервисы могут быть обнаружены динамически. Потребителям часто не нужно знать местоположение и другие детали о сервисе. Метаданные о сервисе могут быть обнаружены в специальном репозитории и реестре сервисов. Когда метаданные сервиса изменились, сервис может обновить свои данные в реестре сервисов;

– композитные сервисы могут быть построены из набора других сервисов.

С сервисно-ориентированной парадигмой каждая бизнес функциональность строится на сервисах, которые реализуют несколько под-функциональностей. Такие сервисы разворачиваются внутри сервера приложений.

Когда дело доходит до потребления бизнес функциональностей нам часто нужно интегрировать множество таких сервисов и других систем. Этой цели служит сервисная шина предприятия.

Потребитель использует составные сервисы, открытые и доступные посредство слоя сервисной шины предприятия.

Следовательно, она используется как централизованная шина, к которой подсоединены все эти сервисы и системы.

Для примера вернемся назад к нашему интернет-магазину (см. Рисунок 4).

На рисунке ниже изображена схема реализации приложения интернет-магазин используя сервисно-ориентированную парадигму.

Мы объявили множество веб-сервисов, которые обслуживают различные потребности бизнеса, такие как продукты, заказчики, портал продаж, заказы, платежи и т. д.

На уровне шины ESB мы можем интегрировать эти бизнес-потребности и создавать объединенные сервисы, которые будут доступны потребителю. Или же ESB слой может быть использован чтобы открыть функциональные возможности пользователю как есть, с дополнительными функциями, например, такими как безопасность.

Очевидно, что ESB слой также содержит значительную часть бизнес-логики всего приложения.

Другие сопутствующие проблемы, такие как безопасность, мониторинг, аналитика могут быть также расположены на уровне шины. ESB слой является

по своей сути монолитом, где все разработчики используют одну и туже среду выполнения для разработки и развертывания их сервисных интеграций.

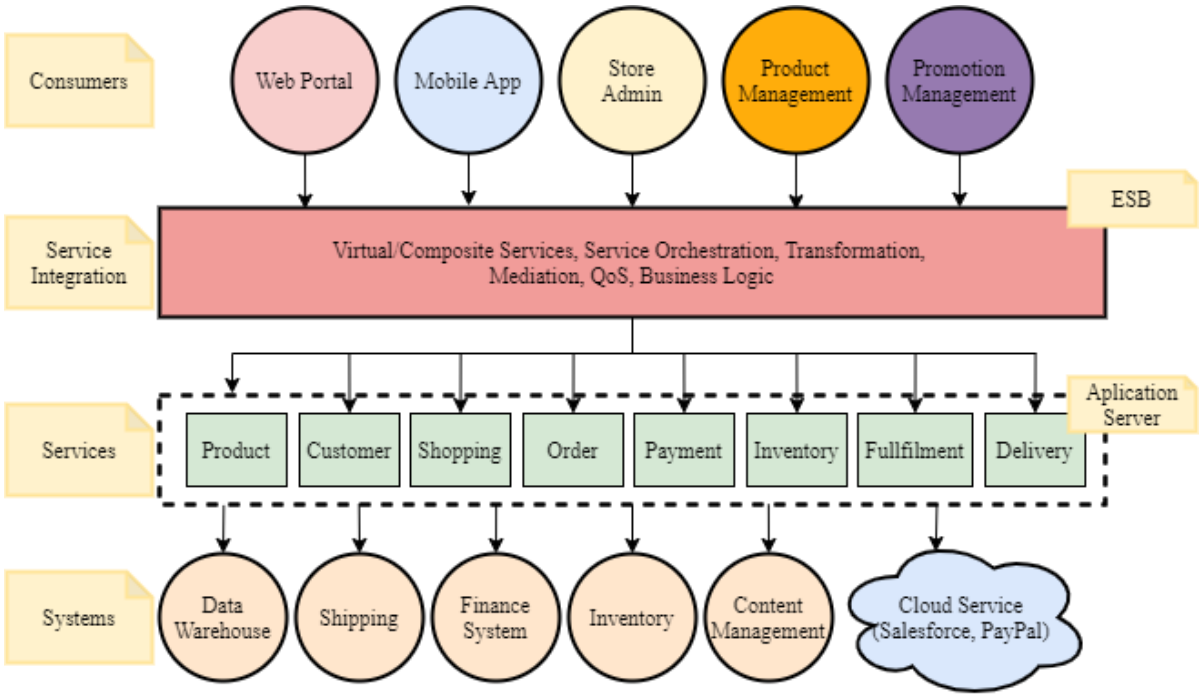

Рисунок 4 – Интернет-магазин, построенный на принципах SOA/ESB

Публикация бизнес функциональностей как сервисы, которыми можно управлять или внешних интерфейсов приложения является ключевым требованием современных архитектур корпоративных приложений.

Тем не менее веб-сервисы, представленные выше, как и сервисно-ориентированная архитектура не являются идеальным решениями таких требований. Связано это с сложностью ориентированных на веб-сервисы технологий, таких как SOAP, WS-Security, WSDL и т. д.

Из-за этого большинство организаций помещают новый уровень управления API (Application programming interface) на вершине сервис-ориентированной схемы реализации.

Этот слой также известен как API-фасад (API facade) или API Gateway. Он открывает простой интерфейс для потребителя и скрывает всю внутреннюю сложность слоя ESB.

Также API-фасад забирает на себя ответственность за безопасность, пропускную способность, кеширование и т. д.

Для примера на рисунке 5 представлен API фасад, расположенный перед ESB слоем.

Все функциональные возможности нашего интернет-магазина теперь раскрыты потребителю через управляемый слой (см. Рисунок 5).

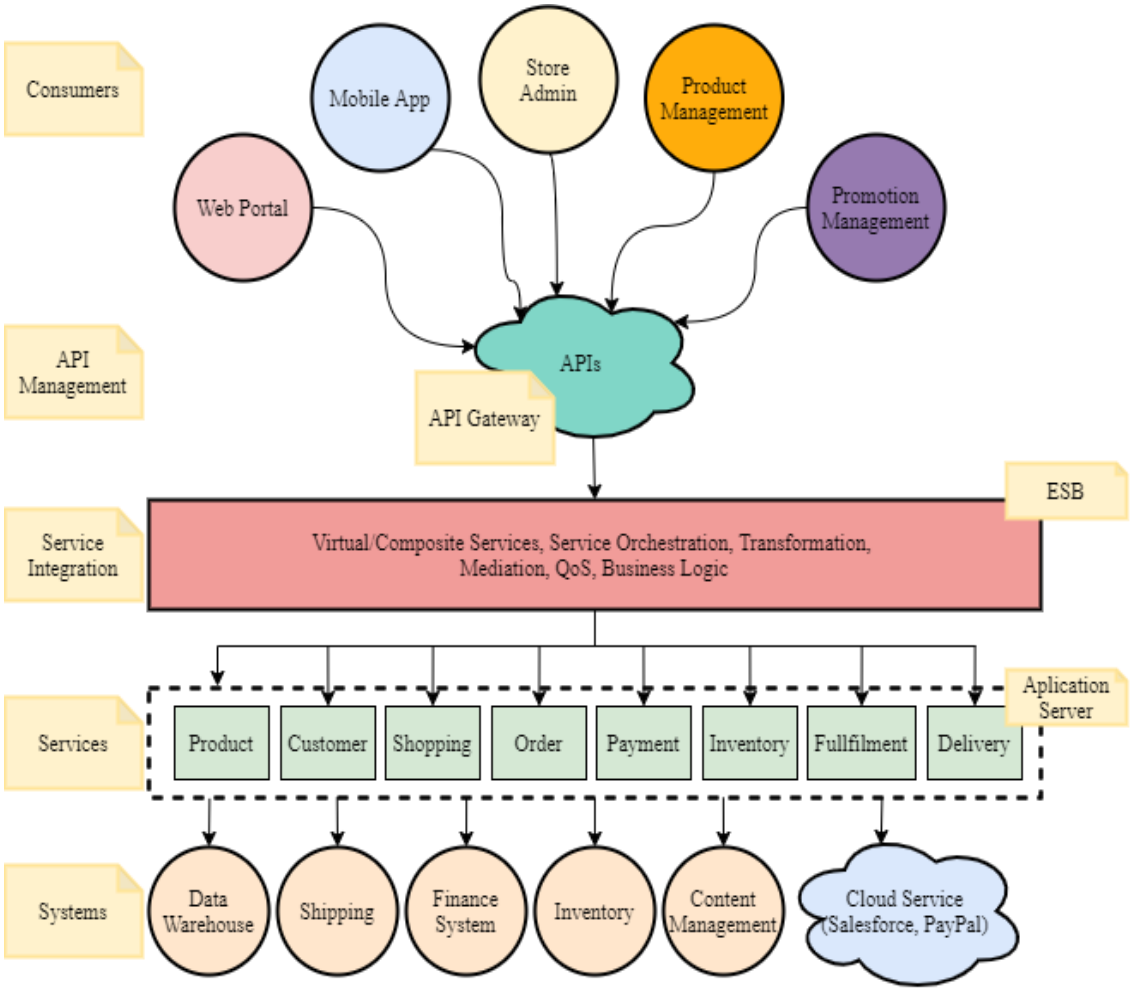

Рисунок 5 – Раскрытие бизнес возможностей приложения через слой API Gateway

С ростом спроса на сложные бизнес возможности, монолитная архитектура больше не может обслуживать разработку современного корпоративного приложения.

Ключевая суть монолитных приложений приводит к отсутствию возможности масштабировать приложения независимо, затруднению независимой разработки приложений, обнажает проблемы с надежностью. Чтобы преодолеть большинство из этих ограничений и удовлетворить современные, сложные и децентрализованные потребности приложений, должна быть придумана новая идеология.

Микросервисная архитектура стала лучшей парадигмой, решивший недостатки подходов с использованием ESB и SOA архитектур, а также традиционных монолитных приложений.

1.3.3 Микросервис

Основой архитектуры микросервисов является разработка единого приложение как набор небольших и независимых сервисов, которые работают в отдельном процессе, а также разработаны и развернуты независимо от остальных.

Рисунок ниже иллюстрирует как интернет-магазин может быть переделан в микросервисную архитектуру посредством разрушения монолитного слоя приложения на отдельные независимые, ориентированные на потребности бизнеса сервисы (см. Рисунок 6).

Также эта архитектура избавляет нас от центральной шины, таким образом сами сервисы заботятся о кросс-сервисном взаимодействии и композитной логике.

Каждый микросервис предлагает четко задекларированные функции для бизнеса, которые проектируются, разрабатываются, разворачиваются и администрируются независимо.

Уровень управления API остается во многом таким же, как и раньше не смотря на отсутствие шины и слоя ее коммуникации с сервисами.

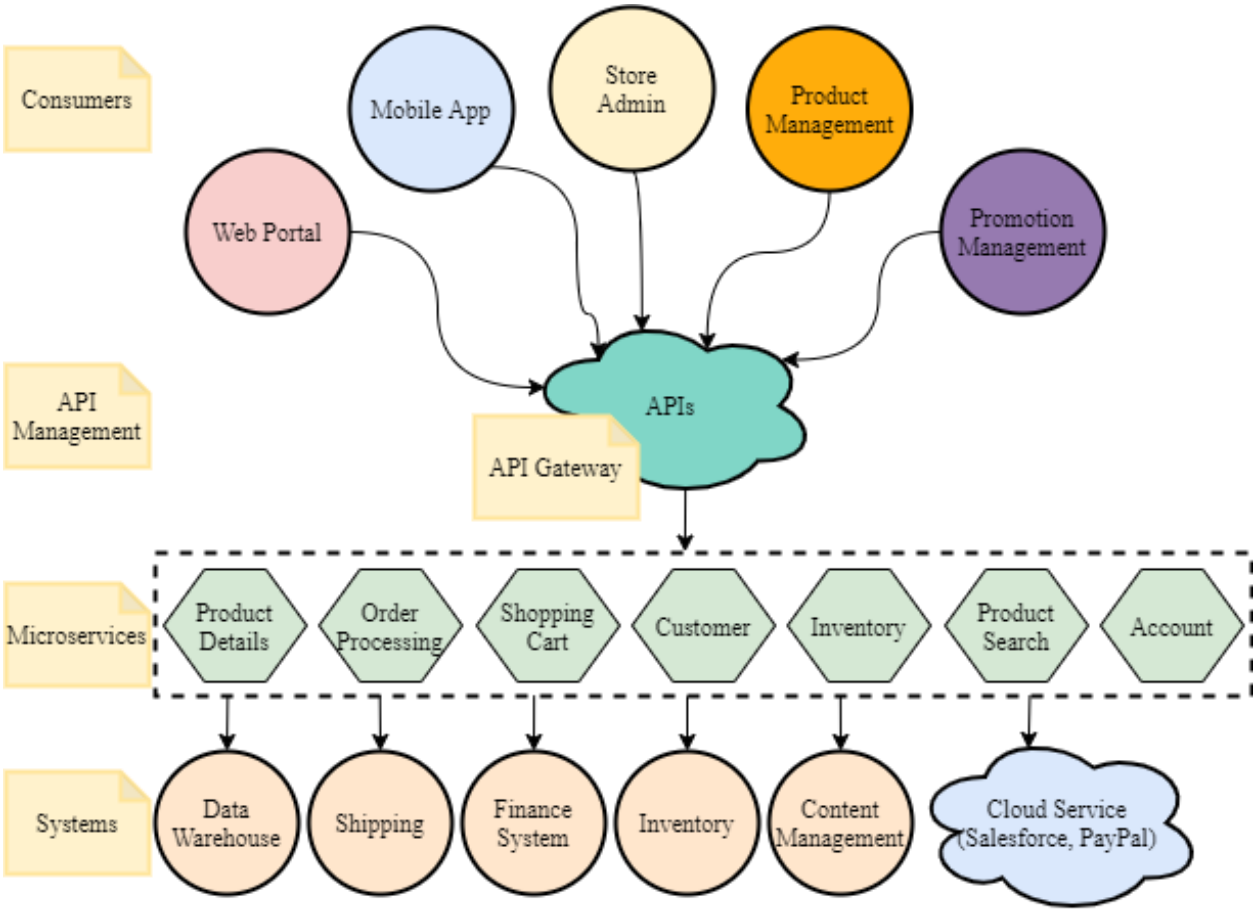

Рисунок 6 – Схема интернет-магазина, построенного по принципам микросервисной архитектуры

API gateway слой открывает потребителю задекларированные бизнес функции как управляемое API.

Также открывается возможность разделить фасад API на отдельные и независимые среды исполнения, разделенные по какому-то признаку.

1.4 Анализ подходов к управлению прикладным программным интерфейсом микросервисов

Любые разрабатываемые микросервисные приложения должны представить бизнес-возможности для потребителей таким образом, чтобы эти возможности можно было легко создавать, управлять ими, защищать, анализировать и масштабировать. Такие возможности предоставляются

потребителям как API, которые регулируются процессом широко известный как управление API (API management). Аналогично, микросервисы могут использовать внешние API в процессе своей работы. В большинстве случае API представляют собой синхронный обмен сообщениями типа запрос-ответ.

Любые бизнес-возможности, которые предоставляются потребителю (внутренние или внешние) можно считать API. Раскрывать такие бизнес-возможностей стоит таким образом, чтобы была возможность создавать, управлять, защищать, анализировать и масштабировать это API. Эта возможность называется управлением API. Используя решение по управлению API, вы можно включить проверку политик и ключей, управление версиями сервисов, управление квотами, примитивные преобразования, авторизация и контроль доступа, наблюдаемость, возможности самообслуживания, рейтинги и т. д. для каждого микросервиса.

Когда дело доходит до управления API, есть несколько основных компонентов, которые должны быть частью любого решения по управлению API:

- API Publisher;
- API Developer Portal / Store;
- API Gateway;
- API Analytics / Observability;
- API Monetization;
- API QoS (качество обслуживание, такие как безопасность, регулирование, кэширование и т. д.).

Остановимся подробнее на описание сервиса публикации API (API Publisher) и так называемом шлюзе API (API Gateway). Сервис публикации может имеет большую зону ответственности чем просто расположения задекларированных API в шлюзе. Он может предоставлять все необходимые возможности разработчикам API для проектирования, разработки,

публикации, версионирования, управления, мониторинга доступности и измерения производительности.

API-шлюз представляет собой единую точку вхождения для клиента, расположенную над сервис-ориентированным бэкендом. При публикации API помещается в API-шлюз. API-шлюз защищает, управляет и ответственен за масштабирование вызовов API. Все запросы, отправляемые пользователями, перехватываются шлюзом или направляются непосредственно в него. Это позволяет применять разные политики для целой группы API, такие как безопасность, фильтры, статистика использования API. Кроме того, в некоторых случаях есть возможность объединить множество бизнес-функций на уровне шлюза API.

Описанные выше принципы управления API открывают очень хорошо накладываются на методологию гибкой разработки и позволяют реализовывать конкретный бизнес функционал отдельно от остальных. Единственная проблема, которая стоит перед разработчиками на микросервисной архитектура – это реализации этих шаблонов. Например, существует великолепная платформа для проектирования API - Swagger, но она хороша развита именно для этого и не предоставляет открытой возможности или какой-либо интеграции с реализациями на API Gateway. Такие интеграции на данный момент разработаны только для облачных решений на Amazon. Сама же платформа Swagger является коммерческим продуктом, это поднимает вопрос об о реальной полезности и выгоды продукта для конкретных случаев решения бизнес проблем. В реалиях новизны технологии готовые решения часто свежи и не могут точно соответствовать тем требованиям, которые стоят перед разработчиками из-за этого реализацию сервисов управлением API. Или же такие реализации существуют, но имеют свою специфику и совершенной другой взгляд разработчиков такого продукта на реализацию таких шаблонов и стоимость такой реализации очень дорого, как дорого и внедрение в свой продукт.

1.5 Проблема управления API

Перед рассмотрением реализаций стоит определиться с проблемой, перед которой мы стоим.

От компании к компании, занимающейся проектированием и реализаций разного рода программных продуктов для бизнеса, осуществляются разные подходы к разработке. Мы будем рассматривать модель разработки, которая применяется в компании NetCracker, как пример достаточно крупной компании по разработке ПО. Все технологии и подразделения, а также ограничения будут накладываться условиями работы среднего разработчика программного обеспечения.

В данной работе мы будем рассматривать модель, когда общий набор задач, которые нужно решить делятся на общие и задачи бизнеса. Реализация общих задач, к которым относятся архитектурные сервисы, и приложения обслуживания называется платформой. На реализацию платформы сосредоточены силы нескольких команд разработки. Платформа должна быть одинаково применима для разных бизнес задач и разных проектов, которые реализуют эти бизнес задачи.

Рассмотренные выше архитектурные шаблоны проектируются и реализуются также командами, отвечающими за платформу.

В подавляющем большинстве именно проекты, реализующие непосредственно бизнес задачи, формируют API и реализуют его в своих микросервисах, в то время как задача по управлению этого API кладется на плечи команд платформы. Это кардинально отличается от процессов проектирования, разработки и публикации API, которые используют в небольших компаниях, в которых одна большая команда трудится над цельным проектом и использует совокупности готовых решений как платформу. Таким небольшим проектам действительно достаточно тех реализаций, которое может предложить IT сообщество.

Платформа подстраивается под нужды проекта, из-за чего за платформой не всегда стоит выбор технологии, особенно если технологию достаточно сложно внедрить.

Однако платформа вправе разрабатывать свои собственные технологии на основе архитектурных шаблонов и внедрять, если они являются достаточно простыми.

В случае управления API всё что есть у разработчиков проектов – это их разработанные на Java микросервисы, в которых средствами Spring framework четко задекларированы API (REST API). Разработчики знают какое API должно быть управляемым. Проекту необходимо решить задачу «общения» микросервисов друг с другом и как было описано выше эта задача решается при помощи шаблона API Gateway. Платформа предоставляет реализацию API Gateway с использованием «Envoy Proxy» в качестве прокси-сервера и маршрутизатора API запросов.

Как итог задача управлением API кладется на проект и на отдельного человека не из числа разработчиков. Другими словами, сотрудник с навыками разработчика и бизнес-аналитика должен сформировать API и самостоятельно настроить «Envoy Proxy» под нужды проекта.

Выдвинем гипотезу, что с теми условиями, что имеются в нашем распоряжении, а именно: готовые микросервисы на Java с четко задекларированным API, готовый сервис для проксирования и маршрутизации Envoy Proxy, облачная среда Openshift или Kubernetes, мы сможем осуществить умную публикацию API и предоставить разработчикам простой инструмент, на основе которого будет формироваться полноценная конфигурация для API-шлюза и это приведет к отсутствие человеческих ошибок, упрощению и надежности тестирования.

1.6 Анализ инструментов управления прикладным программным интерфейсом микросервисов

Развитие технологии виртуализации, а в частности контейнеризации дало толчок и микросервисной архитектуре.

На данный момент благодаря популярности этого подхода к построению архитектуры корпоративных приложений реализовано множество инструментов для управления API.

Несмотря на это реализация большей часть сформированных архитектурой сервисов остается на плечах разработчиков конкретного приложения или платформы для разработки решений.

Так, например, не существует эталонной реализации сервиса публикаций API в его явном виде и в свободном доступе, в то время как разных реализаций API-шлюзов большое разнообразие.

Перед рассмотрением реализаций стоит определиться с проблемой, перед которой мы стоим.

От компании к компании, занимающейся проектированием и реализаций разного рода программных продуктов для бизнеса, осуществляются разные подходы к разработке. В данной работе мы будем рассматривать модель, когда общий набор задач, которые нужно решить делятся на общие и задачи бизнеса. Реализация общих задач, к которым относятся архитектурные сервисы, и приложения обслуживания называется платформой. На реализацию платформы сосредоточены силы нескольких команд разработки. Платформа должна быть одинаково применима для разных бизнес задач и разных проектов, которые реализуют эти бизнес задачи.

Рассмотренные выше архитектурные шаблоны проектируются и реализуются также командами, отвечающими за платформу. В подавляющем большинстве именно проекты, реализующие непосредственно бизнес задачи, формируют API и реализуют его в своих микросервисах, в то время как задача

по управлению этого API кладется на плечи команд платформы. Это кардинально отличается от процессов проектирования, разработки и публикации API, которые используют в небольших компаниях, в которых одна большая команда трудится над цельным проектом и использует совокупности готовых решений как платформу. Таким небольшим проектам действительно достаточно тех реализаций, которое может предложить IT сообщество.

Рассмотрим самые популярные реализации API-шлюза и реализации управления API. В список таких реализаций входит:

– Spring Cloud Gateway,
– Kong Gateway,
– Swagger,
– NGINX,
– Envoy Proxy.

Рассматривать эти реализации стоит со стороны процесса управления API, которое они предлагают если это реализация шаблона API Publisher, а также с точки зрения процесса доставки конфигурации API в API-шлюз.

Spring Cloud Gateway проект является частью проекта Spring Cloud, целью которого является предоставить разработчикам облачных решений максимальную инструментальную поддержку и избавить их от необходимости реализовывать одно и тоже. Сам же проект Spring Cloud Gateway предоставляет библиотеку для построения шаблона API Gateway. Цель проекта – предоставить простой и эффективный способ маршрутизации запросов к API. Функционал этого решения из коробки предлагает статический способ программной декларации маршрутов, что делает его сложным используя в качестве общего и настраиваемого решения для разных проектов и приложений. Из очевидных плюсов – открытый исходный код.

Kong Gateway также является проектом с открытым исходным кодом и написан на языке Lua. Предоставляет пользователям полноценную реализацию шаблона «API Gateway» с функциями маршрутизации трафика,

аутентификации, мониторинга, протоколирования, кеширования, контроля трафика. Имеет разные форматы настроек, как статичных через файл конфигурации, так и динамическую посредством запросов HTTP. Является не плохим с точки зрения производительности решением, так как основан на «nginx». Но так как полностью состоит из набора скриптов на Lua, то общая производительность и явные недостатки с размером потребляемой памяти остаются под вопросом. К тому же настройка посредством HTTP имеет свои недостатки, так, например, следить за актуальностью маршрутов в API-шлюзе без прочной обратной связи становится чрезвычайно сложно.

NGINX – представляет собой реализацию прокси-сервиса с богатой конфигурацией, на его основе построен проект Kong, но также разработчик NGINX предлагает из коробки множество возможностей: балансировку, кэширование, протоколирование и т. д. Имеет все те же преимущества и недостатки что и Kong, не смотря на то, что является его значительно урезанной по возможностям версией.

Swagger – это проект с открытым исходным кодом, целью которого является предоставить инструменты и платформу для проектирования, тестирования и документации программным интерфейсом приложений. Фокус внимания этого проекта сосредоточен на веб сервисах, построенных на концепциях REST. Swagger является частичной реализацией шаблона API Publisher. Swagger диктует свой процесс разработки API, который начинается с дизайна, а заканчивается автоматической генерацией кода, на основе существующей спецификации.

Envoy Proxy. Название этого проекта говорит само за себя, это реализация прокси сервера на языке C.

Этот проект предлагает большой список возможностей, но что более главное – широкие возможности по его настройке. Этот проект часто используется в личных разработках разных компаний в качестве реализации шаблона API Gateway.

Явным преимуществом проекта является его динамическая настройка и предоставляемая специальная библиотека для управления конфигурацией прокси-сервера из вне. Такой слой в терминологии Envoy proxy является «Control plane». Протокол управления конфигурацией строится на gRPC. Это упрощает реализацию модели издатель-подписчик в конфигурировании envoy proxy.

1.7 Анализ требований к инструменту управления API

Перед тем как приступать к разработке модели проектируемого программного продукта стоит определиться с тем что именно из себя будет представлять конечный продукт, какими свойствами и характеристиками он будет обладать, в каких средах он будет работать так, как задумано и учесть ограничения, накладываемые средой выполнения на программный продукт.

На основе выводов из предыдущего раздела составим список функциональных требований к программному продукту.

И так, программный продукт должен.

Предоставлять возможность настраиваемой маршрутизации сетевого трафика. Чтобы осуществить работу API-шлюза необходимо иметь проксирующий сетевой трафик модуль или сервис, который можно настроить во времени выполнения.

Работать с сетевым трафиком, использующим протокол HTTP. Запросы на API-шлюз по протоколу HTTP должны корректно проксироваться.

Использовать задекларированный формат конфигурации. Например, воспользоваться известными форматами данных (XML, JSON, YAML)

Конфигурация должна содержать правила, по которым определяется маршрут сетевого трафика. Например, должна быть возможность указать что, придя на условный API-gateway по адресу

http://gateway:80/api/v1/service1/action трафик будет передан на другой адрес http://service1:80/action.

Правила определения направления трафика должны учитывать специфику популярного подхода к организации API REST и работать с путем URL, HTTP заголовками. Здесь имеется ввиду поддержка маршрутизации в случаях с использованием переменных путей. Например, запрос на http://gateway:80/api/v1/service1/action/{идентификатор} будет перенаправлен на http://service1:80/action/{идентификатор}, идентификатор будет также передан.

Должна быть возможность динамического изменения конфигурации во времени выполнения. В то время как приложение запущено должна быть возможность поменять конфигурацию маршрутизации без рестарта шлюза и публикатора API.

Конфигурация маршрутизации должна быть устойчивая к сбоям и не теряться после рестарта приложений.

Инструментом управления API является реализация шаблонов облачного проектирования приложений API Publisher и API Gateway.

Оба этих приложения представляют собой самостоятельную часть или модуль полноценного программного решения в облаке.

Учитывая этот аспект, мы можем точно сказать, что они имеют аналогичный цикл разработки и развертывания, как и облачное программное решение.

Разработка приложений для работы в облачном окружении имеет свои отличительные особенности от приложений для использования на ПК, WEB-приложений или приложений, предназначенных для работы на мобильных платформах (Android, iOS).

Чтобы приложение заработало в облачном окружении необходимо выполнить следующие шаги.

Собрать приложение.

Приложение должно быть скомпилировано, слинковано и упаковано в исполняемый файл или набор файлов и иметь конкретную точку запуска.

Упаковать в docker-образ.

Собранное приложение необходимо упаковать в docker-образ, подобрав операционную систему, набор необходимых для работы приложения библиотек.

Опубликовать docker-образ в специальном месте.

Например, образ может быть опубликован на ресурсе «Docker Hub» или на определенном, предназначенном для этого ресурсе внутри компании (Artifactory).

Развернуть приложение.

Разворачивание может производиться вручную, публикацией конфигураций развертывания на облаке, автоматически скриптами или соответствующим инструментом развертывания, как часть процесса CI/CD (Continuous Integration / Continuous Delivery).

Облачная среда на приложение накладывает как ограничения, так и дает возможность использовать более широкий набор технологии для реализации приложения. Из очевидных преимуществ можно выделить следующие:

Большой выбор инструментов реализации программного продукта.

Контейнеризация позволяет нам выбрать любой язык программирования, который поддерживается средой контейнера, а это практически любой язык программирования, который поставляется с средствами разработки на этом языке под популярные операционные системы Windows и семейств UNIX+.

Хранение конфигурации.

Облачная среда предоставляет возможность хранить конфигурации в надежном и отвечающем требованиям высокой доступности хранилище ключ-значение. Такие PaaS среды как Kubernetess и Openshift предоставляют

возможность указать свой собственный ресурс для хранения посредством объявления Custom Resource Definition.

Однако ограничение заключается в том, что мы должны подготовить наши приложения для работы в облачной среде. Выделим основные пункты требований.

Работа с сетью. Приложение обязано уметь работать с сетью, получать и отправлять данные по сети используя популярные протоколы передачи данных и организации взаимодействия между приложениями в сети – HTTP, gRPC.

Контейнер. Для работы в облаке приложение должно быть запаковано в специальный контейнер, с настроенным окружением, в котором есть всё необходимое для полноценной безошибочной работы приложения;

Конфигурация развертывания. Для запуска и поддержания приложения в рабочем состоянии PaaS слою виртуализации необходима конфигурация, описывающая как разворачивать контейнер, как проверять его работоспособность, какие точки коммуникации приложение имеет.

Глава 2 Методы разработки и развертывания инструментов управления прикладным программным интерфейсом микросервисов

2.1 Методы управления прикладным программным интерфейсом

Управление прикладным программным интерфейсом – это способ организации архитектуры микросервисов с целью получить четкий контракт для каждого сервиса и скрыть от пользователя этого контракта реализацию, в виде конкретного микросервиса. Также это является точкой расширения для других, не менее важных функции внутри микросервисного приложения: интроспекция авторизационных токенов, ограничение трафика, балансировка нагрузки.

Так как подход к управлению API и сама технология облачных сред и микросервисная архитектура достаточно «молодые» направления в разработке прикладного ПО, то и методов к такому подходу существует не так много. Однако есть два метода, которые достаточно просты и имеют опытное применение в разных программных решениях. Первый и самый простой метод заключается в ручной настройке API-шлюза (см. Рисунок 7).

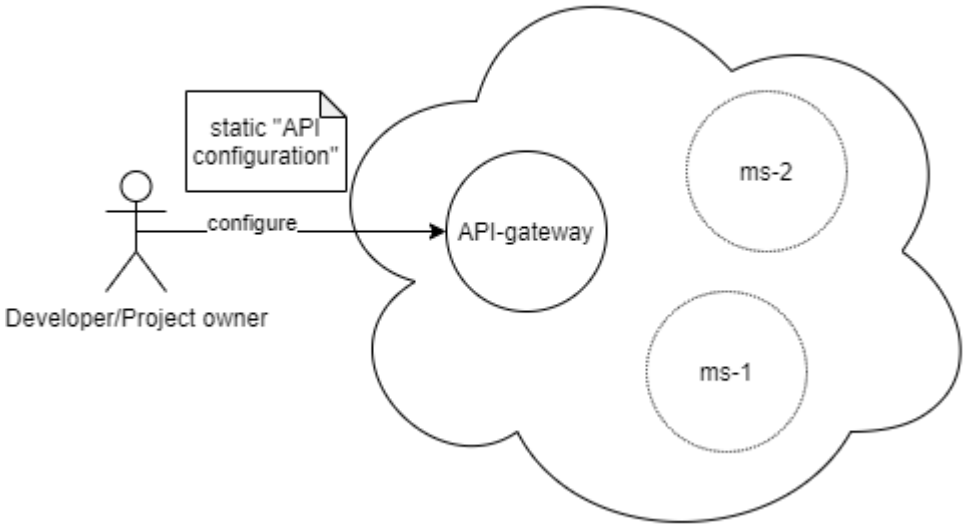

Рисунок 7 - Метод ручного управления API

В качестве API-шлюза выбирается любой программный продукт с возможностями маршрутизации сетевого трафика, а также возможностью гибкой настройки этих маршрутов. В качестве API-шлюза может выступать даже простой прокси-сервер NGINX, настройка которого осуществляется посредством специального файла конфигурации. Однако для осуществления такого метода управления API необходимы глубокие знания в настройке API-шлюза. Конкретный подход в настройке API шлюза, конечно, зависит от выбранного программного продукта, реализующего функции API-шлюза, но достаточно часто такой подход выражается в файле конфигурации с конкретным контрактом. Контракт настройки может абсолютно отличаться от одного программного продукта к другому и в большинстве случаев сложен в освоении. Поэтому очевидным недостатком такого подхода является, во-первых, сложность в настройке API-шлюза, во-вторых, консервативность конфигурации к динамическим изменениям маршрутов, что например исключает возможность динамического перенаправления сетевого трафика без перевода всего облачного решения в режим обслуживания. Также такой метод применим в относительно небольших по функциональным возможностям программных решениях в облаке, поскольку с ростом возможностей растет и количество микросервисов и активно расширяется API. Вручную управлять и следить за актуальностью API, а также вручную изменять конфигурацию API-шлюза становится трудоемкой задачей и неэффективной тратой человеческих ресурсов. Плюсом такого подхода является гибкая настройка API-шлюза, ограниченная возможностями программного продукта его реализующего.

Рассмотрим следующий метод управления API, широко применяемый в компании NetCracker. Этот метод изображен на рис. 9. Суть метода в наличии двух дополнительных составляющих в архитектуре управления API, а именно сервиса публикации API и специальной библиотеки. Библиотека используется в качестве зависимости в микросервисах и при старте микросервиса собирает

все метаданные из исходного кода самого микросервиса (для примера аннотации Java), формируя далее запрос на изменение конфигурации в API-шлюзе. Этот запрос отправляется в API-публикатор, где подготавливается, трансформируется и отправляется уже непосредственно в сам API-шлюз. В применении такого метода есть однозначные преимущества:

– API-шлюз настраивается в автоматическом режиме, что исключает простые и глупые ошибки;
– не нужно следить за актуальностью конфигурации API в API-шлюзе, так как при каждом старте приложения в облаке актуальная конфигурация тут же будет отправлена в сервис публикации API и сформирована актуальная конфигурация;
– во избежание использования библиотеки описание API микросервиса можно отправить в публикатор вручную.

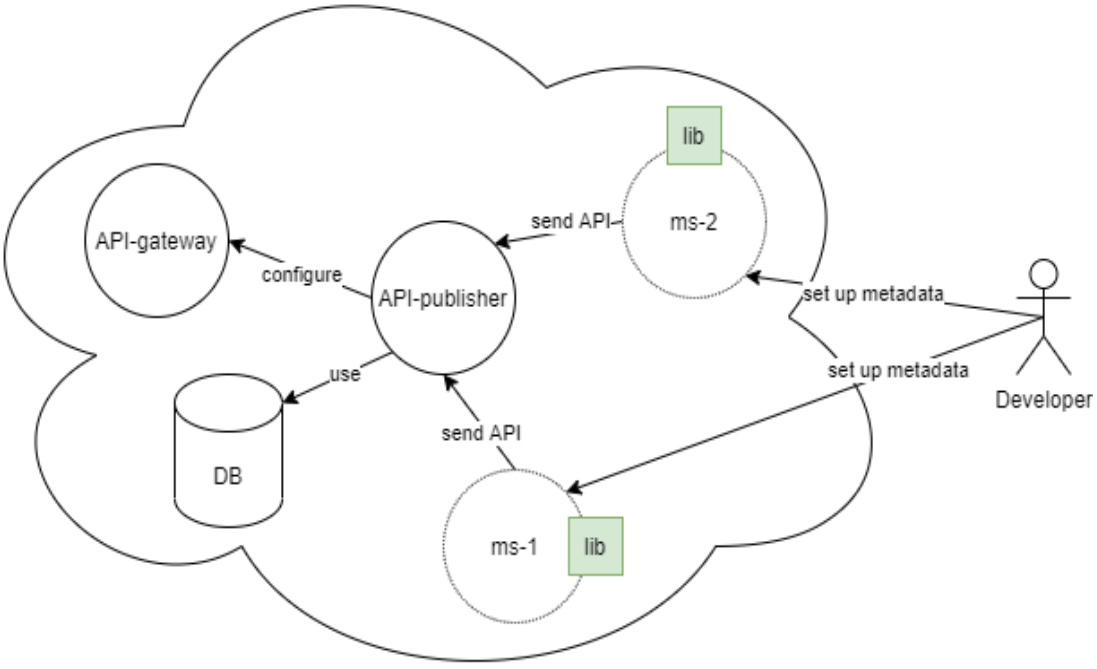

Рисунок 8 – Метод авто-конфигурации с метаданными и хранилищем

Такой метод не лишен и недостатков. Библиотеку для микросервисов нужно реализовывать на множестве языков, потому что микросервисная архитектура не накладывает ограничения на язык, разработчики могут использовать любой, и чтобы воспользоваться преимуществами этого метода библиотека должна быть реализована на всех. Взглянув поглубже, в этом недостатке можно найти ещё и технические ограничения. Не все языки программирования обладают хорошими средствами рефлексии кода. И часто использование рефлексии ухудшает производительность. Следующим существенным недостатком является то, что конфигурация, формируемая и хранимая в публикаторе API (для дальнейшего распространения в API-шлюз) имеет жизненный цикл только на добавление и обновление списка API. Другими словами, с точки зрения публикатора невозможно оценить, какие API потеряли актуальность и должны быть удалены из конфигурации. Этот недостаток накладывает обязательства на владельца продукта следить за актуальностью конфигурации и удалять вручную уже не используемы API из публикатора. Также этот метод вместе с добавление публикатора добавляет зависимость на БД или внешнее хранилище данных, что в случае его отказа выводит из строя всю цепочку управления API и лишает возможностей настройки (а в случае перезапуска API-шлюза лишает и возможности маршрутизации трафика).

Проанализировав методы выше и отметив их преимущества и недостатки, можно с уверенностью сказать, что цель удобства использования реализаций этих методов, расширяемость и доступность в этих методах не достигнута. В первом методе мы имеем сложную настройку API-шлюза вручную, что требует специальных знаний о конфигурации конкретного приложения, реализующего возможности проксирования запросов. Также ручной метод исключает возможности динамического изменения состояния API-шлюза, что для продуктового решения становится не пригодно. Второй метод покрывает недостатки первого, но привносит новые. Очевидно, что

дополнительное звено в архитектуре в виде хранилища данных обязывает пользователя этого метода дополнительно следить за надежностью хранилища данных, а также имеет проблемы в самой реализации, так как появляется проблема изменяемости конфигурации. Проблема изменяемости конфигурации заключается в том, что пользователю не очевидно в каком, на текущий момент, состоянии находится конфигурация, если позволено динамически изменять разные её части разным пользователям. Эта проблема решается наличием статической, наглядной конфигурации. Чтобы избежать недостатков методов выше первое что необходимо сделать, это сформировать упрощенный контракт конфигурации по отношению к конфигурации API-шлюза, который в виде файла конфигурации будет хранится в надежном хранилище данных, за доступность которого отвечает слой PaaS.

Такой метод разработан в рамках этой научно-исследовательской работы на основе знаний о инфраструктуре PaaS слоев и их технологических возможностей и изображен схематично на рис. 9.

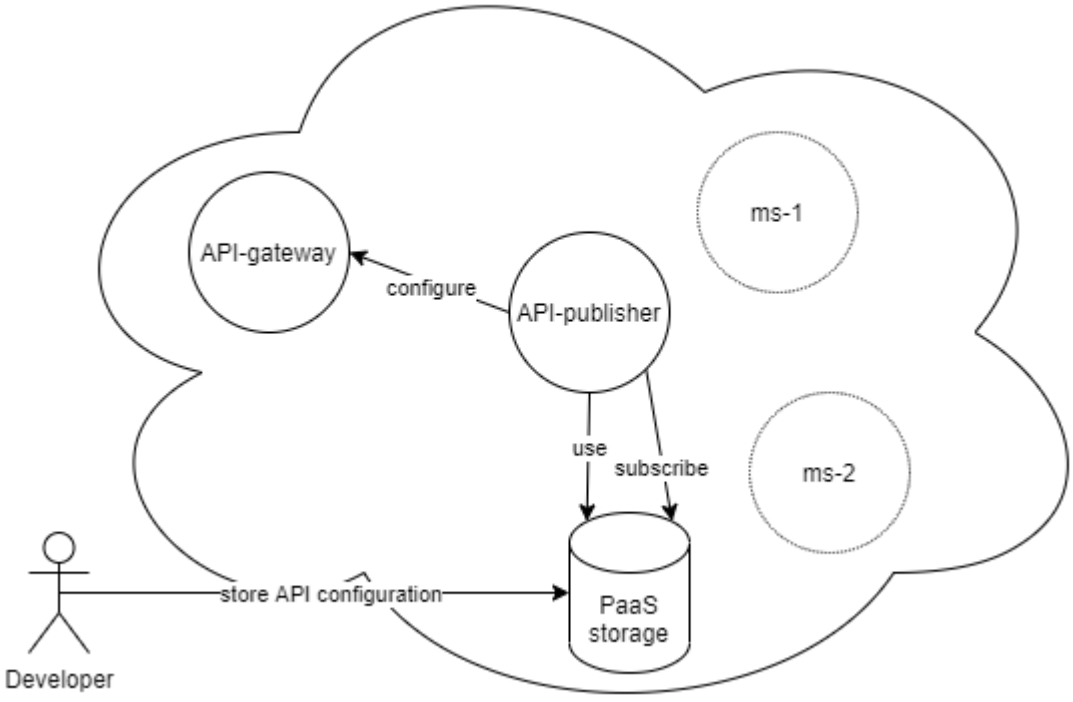

Рисунок 9 – Метод управления API с преимуществами использования PaaS

Метод расширяет и дополняет возможности метода выше, разработанного в компании NetCracker, однако избавляется от очевидных недостатков и добавляет новые «точки расширения».

Суть этого метода в том, чтобы в качестве хранилища конфигурации использовать встроенный в PaaS слой механизм. Он является надежным, т. к. сам PaaS слой на нем и построен. В качестве реализации этого хранилища используется etcd. Это хранилище организовывает данные по принципу ключ-значение и предоставляет целостность данных и их доступность по требованию. Также PaaS слой предоставляет возможность отслеживать изменения данных конкретного типа. В этом методе публикатор подписывается на отслеживание конкретных типов данных, именуемых конфигурацией API, в PaaS слое и при каждом изменении производит полную пересборку всей конфигурации API-шлюза.

Этот метод устраняет недостатки в виде зависимости от внешнего хранилища данных, поскольку зависит уже непосредственно от хранилища, встроенного в PaaS слой. Однако хранилище PaaS слоя неотделимо от него и поставляется сразу вместе с ним, что означает что везде, где есть этот PaaS слой — значит есть и это хранилище. В общем и целом, сложность публикатора в таком методе значительно упрощается, он становится простым «конвертером» простого формата конфигурации API-шлюза в более сложный. Необходимость в библиотеке отпадает, но конфигурацию необходимо создавать вручную и сохранить её в PaaS слое.

2.2 Технологии разработки микросервисов

Как и под конкретную платформу, архитектуру процессора, операционную систему есть своя специфика разработки, так и для облачных сред имеются свои нюансы. Несмотря на то, что контейнеризация дает широкий выбор языков программирования для реализации микросервисов, мы

остановим свой выбор на двух одних и самых популярных языков, адаптированных для работы в облачной среде. Это языки Java и Golang.

Java на данный момент один из самых популярных языков для разработки корпоративных приложений да и в целом в области прикладной разработки имеет лидирующие позиции за счёт большого сообщества, богатого выбора библиотек и готовых решений. В совокупности за счёт скорости разработки и достаточно хорошие показатели производительности этот язык сейчас и выбирается как основной инструмент для реализации программных продуктов для бизнеса. Для быстрого старта в облаке микросервиса, написанного на java существует библиотека Spring Boot, она позволяет с минимальным количеством кода (1 класс) запустить полноценный HTTP сервис, обрабатывающий запросы и это всё запаковано в один большой пакет с байт-кодом и готовом набором библиотек. Всё что нужно, чтобы запустить это приложение – java машина, установленная в среде запуска.

Однако язык Java и библиотека «Spring Boot», в частности, не лишена недостатков. Использование этой комбинации требует большое время старта и большой объем памяти. Хотя и на среднестатистическом компьютере время старта Spring Boot приложения занимает 1 секунду – это все равно достаточно много для приложения, которое ничего не делает. С добавлением так называемых бинов (Bean) в приложение увеличивает время старта и на практике это время может исчислять как 30 секундами, так и минутами. Что касается памяти, то потребление памяти у приложения Spring Boot начинается от 64 Мб и растет по мере роста функционала микросервиса и может достигать размеров 1Gb и даже более.

Golang себя зарекомендовал как язык программирования для создания высокопроизводительных приложений. Он сочетает в себе простоту языка и предоставляет очень простые, но эффективные инструменты для реализации многопоточных приложений. Что примечательно многие части PaaS слоя Kubernetess и средства контейнерной виртуализации Docker были написаны на

этом языке. Приложения на языке golang компилируются в один исполняемый файл, однако для каждой платформы существует свой компилятор. Время старта простого микросервиса с схожим функционалом с Spring Boot приложением на Java исчисляется миллисекундами (5-10 ms), а размер используемой памяти 2-5 Мб. Однако язык golang не обладает большой базой качественных и проверенных временем библиотек, в отличии от языка Java. И часто возникает потребность реализовывать необходимый шаблонный функционал самостоятельно. Рассмотренные аспекты снижают время разработки на этом языке, поэтому «крупные» микросервисы с разнообразной бизнес-логикой и функционалом писать на таком языке не целесообразно из-за трудозатрат на разработку и не мало важно на сопровождение такого микросервиса. Более детальное сравнение технологий изображено на таблицу 1.

Таблица 1 – Сравнение характеристик технологий разработки для облака

Характеристика	Java + SpringBoot	Golang + mux
Время старта	1 секунда	5-10 милисекунд
Потребление памяти	от 64 Мб	От 2-5 Мб
Порог вхождения	средний	низкий
Поддержка gRPC	есть	есть
Библиотека работы с Envoyproxy	есть	есть
Управление памятью	сборка мусора	сборка мусора
Многопоточная модель	тяжеловесная (потоки, блокировки)	облегченая (gorutine, каналы, блокировки)
Скорость разработки	быстрая за счет большого количества готовых библиотек	условно средняя (с ростом сложности приложения замедляется), готовых решений очень мало

Оба этих языка предоставляют возможности по созданию HTTP-сервера без особых трудностей. В golang это предоставляется стандартной библиотекой языка, в java большая часть реализации возложена на сторонние библиотеки. Оба языка поддерживают работу с протоколом gRPC, а также прекрасно поддерживают работу и даже сборку приложения в контейнере.

2.2.1 Методы развертывания микросервисов в облаке

С разработкой программного продукта необходимо разработать и схему процесса его автоматизированного развертывания в облаке, потому что конфигурация развертывания должна быть опубликована в PaaS, чтобы процесс жизненного цикла приложения начался.

Развертывание микросервисов в облаке и методы того, как это сделать заслуживают отдельной книги, однако в рамках научно-исследовательской работы будет проведен краткий обзор.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Рисунок 10 – Пример Deployment конфигурации для Kubernetess

Одной из особенностей облачных окружений является файл конфигурации развертывания, который непосредственно публикуется в среде PaaS, он так и называется Deployment.

Для примера возьмем PaaS среду Kubernetess и рассмотрим пример конфигурации.

Очень важным пунктом конфигурации является свойство “image”, в нем указано имя образа контейнера, который будет браться из настроенного на хост машине репозитория docker-образов. По-умолчанию это Docker Hub.

Для большей наглядности изобразим весь путь контейнера от момента его создания (сборки) до момента попадания на хост машину с Kubernetess.

Свой жизненный путь докер образ берет с исходного кода, а если быть точнее с Dockerfile, где описано из чего создается докер образ и что в нем должно содержаться. Именно в этом файле описывается то, что образ будет содержать в себе, какие файлы, какие инструменты и программы.

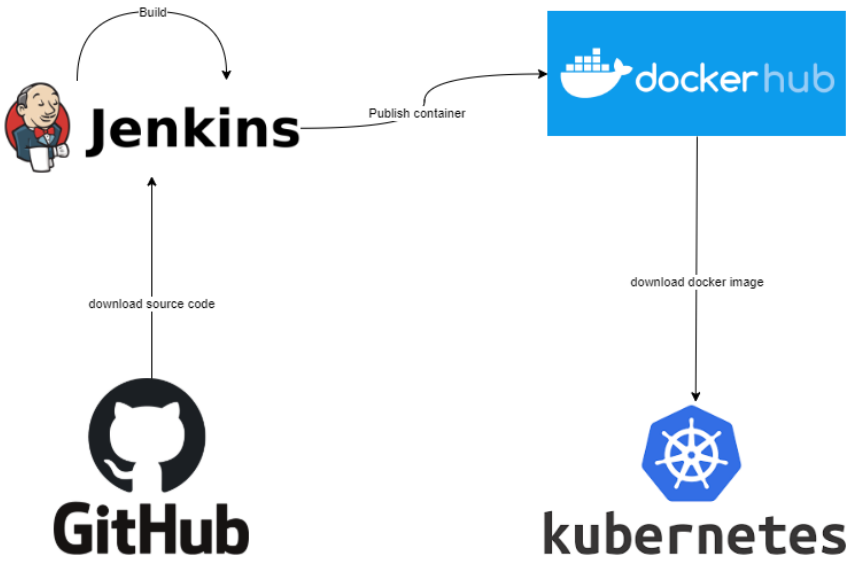

Рисунок 11 – Процесс сборки и публикации Docker контейнера

После того как докер образ собирается вместе с скомпилированным микросервисом он публикуется в репозитории докер образов, после чего ему дается имя и проставляется тэг, что формирует уникальное имя образа. После публикации этот образ может быть загружен и запущен в Kubernetess. Алгоритм сборки изображен на рис. 12.

Однако сам микросервис в облаке появиться не может без конфигурации, а значит она должна быть кем-то опубликована в PaaS.

Есть два способа того, как это сделать автоматически.

Первый. Написать скрипт. Процесс развертывания можно легко автоматизировать при помощи обычных скриптовых программ. Так работает большинство деплоеров, построенных на основе Jenkins;

Второй. Написать программу-оператор, которая на основе одного файла конфигурации развернет несколько микросервисов как модуль.

Не смотря на наличие разных способов автоматизирования развертывания логика остается одинаковой и включает в себя несколько простых шагов. Алгоритм логики развертывания изображен на рис. 13.

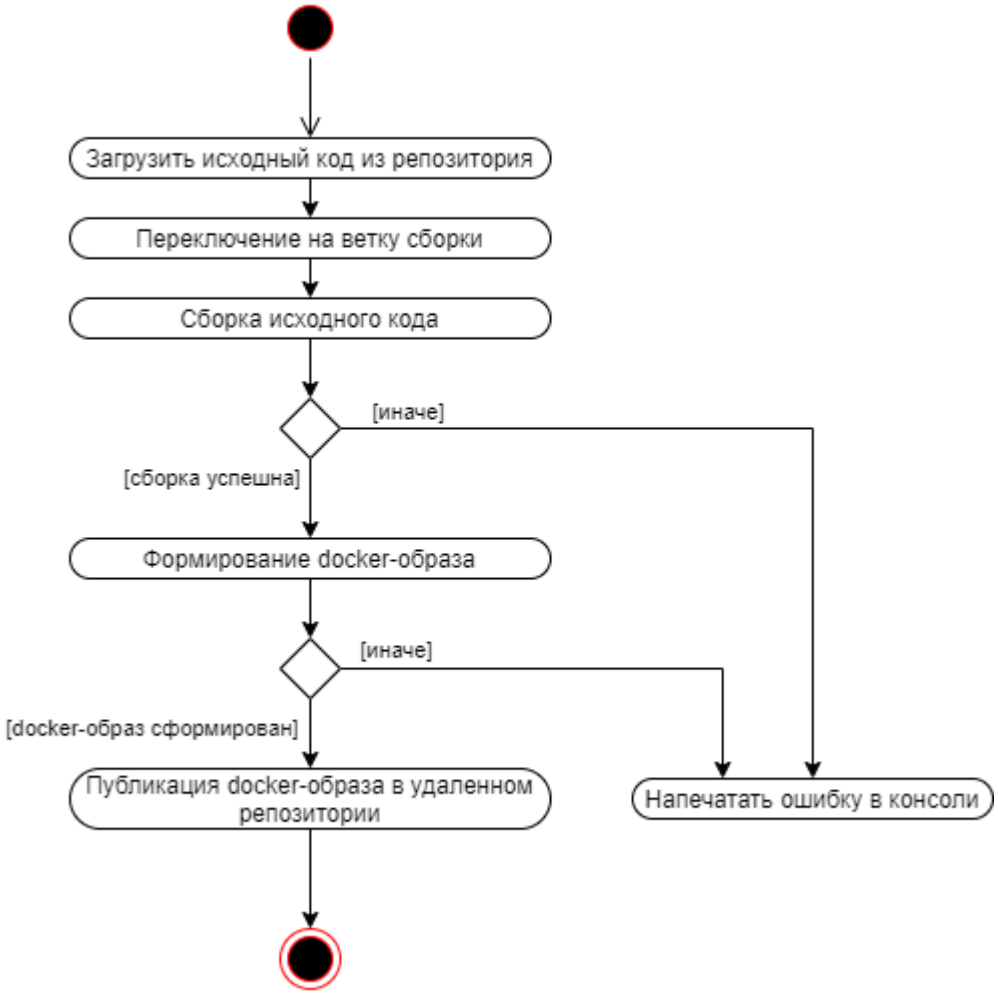

Рисунок 12 – Алгоритм сборки исходного кода микросервиса в docker-образ

Схема развертывания при помощи деплоера (скрипта) удобна, когда деплоер разрабатывается под развертывание конкретного набора микросервисов и под конкретный проект.

Это связано с тем, что набор микросервисов определяется заранее и, если мы хотим предоставить наш программный продукт как набор микросервисов, мы должны и предоставить деплоер пользователю, чтобы он мог им воспользоваться.

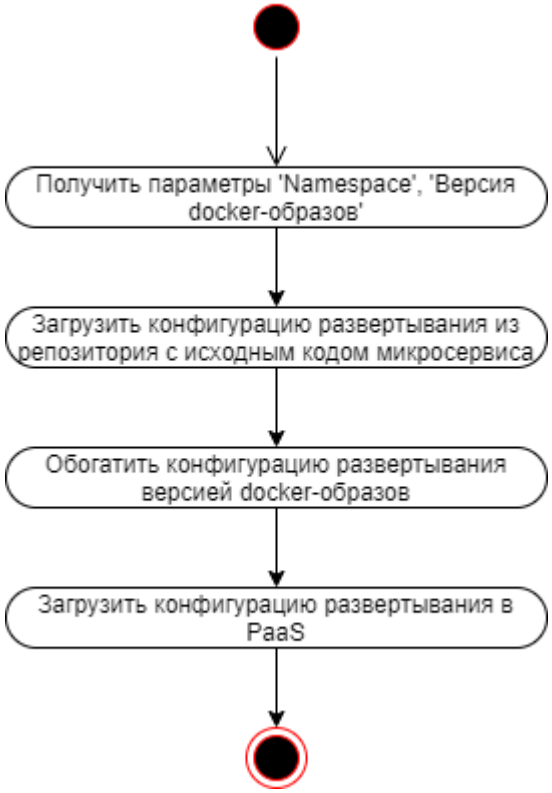

Рисунок 13 – Алгоритм развертывания микросервиса

Для случаев предоставления программного продукта на основе набора микросервисов лучше использовать программу-оператор, которая в ручном режиме будет установлена один раз, и она будет заниматься развертыванием нашего решения.

В таком выборе возможна и разная конфигурация развертывания, настройку которой можно предоставить пользователю.

Это можно назвать одним способов предоставления стратегии развертывания программного продукта.

В более простых случаях это выглядело как предоставление пользователю скрипта, который установит программный продукт и настроит его.

2.2.2 Концептуальная архитектура инструмента управления прикладным программным интерфейсом

В качестве API-шлюза выберем готовый продукт с гибкой возможностью настройки Envoyproxy.

Плюсами является то, что разработкой пользуется достаточно большое количество крупных компаний по разработке ПО, среди них и Google и IBM.

Также среди преимуществ то, что он написан на языке C++ и является высокопроизводительным и с очень низким потреблением по памяти приложением.

Однако разработка приложения с ролью API Publisher, которое будет поставлять конфигурацию маршрутизации в API-шлюз, кладется на наши плечи. Envoyproxy для динамической конфигурации требует внешний сервис, который по gRPC протоколу предоставляет конфигурацию для маршрутизации. В качестве источника конфигурации будет выступать API Publisher.

Рассмотрим концептуальную схему работы нашего программного продукта. И так на рисунке ниже изображено облачная среда под управлением PaaS Kubernetess (см. Рисунок 14).

Все фигуры, кроме человечков обозначают ресурсы облака.

Круглым обозначены сервисы. Сервисы в Kubernetess предназначены для сокрытия набора Pod под одним DNS именем.

Например, исходя из схемы у нас может быть несколько экземпляров API-gateway запущенных в облаке, однако обращение к ним возможно через DNS имя api-gateway.

Благодаря service записи, опубликованной в облаке kubernetes поймет, что пользователю, обратившемуся во внутренней сети по доменному имени нужен любой из запущенных экземпляров api-gateway.

Для более простого понимания можно сказать, что service это DNS запись, скрывающая несколько IP-адресов запущенных экземпляров приложения.

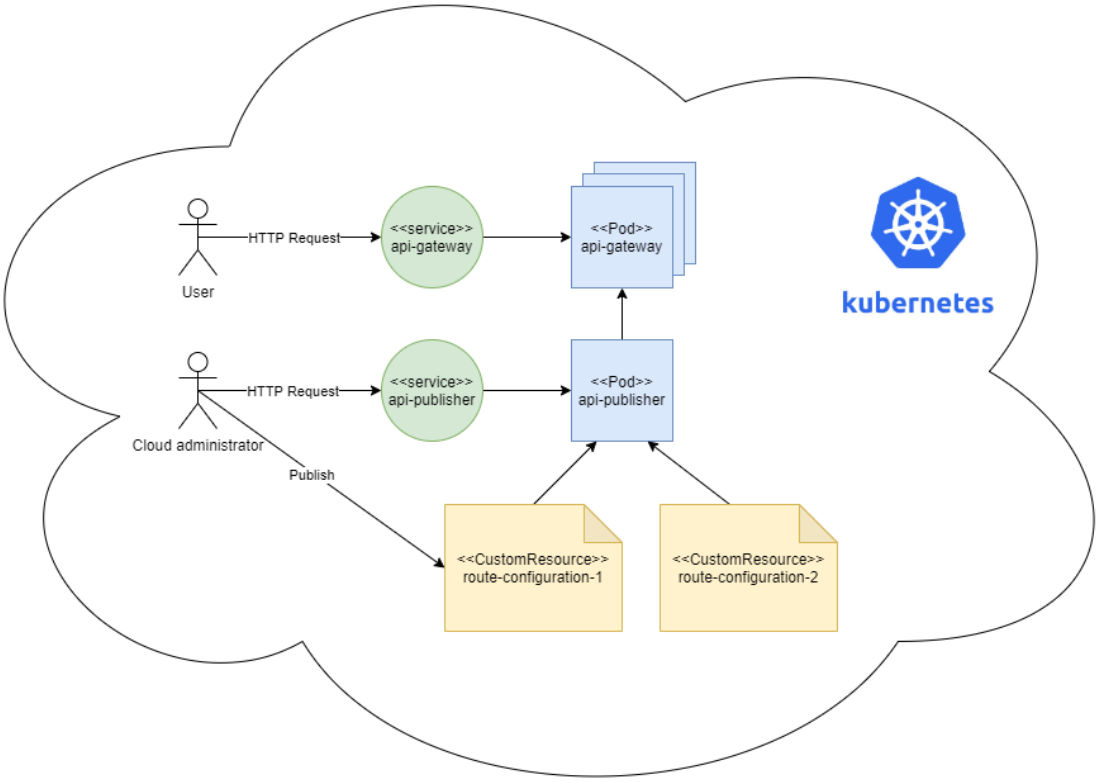

Рисунок 14 - Концептуальная архитектура

Квадратом обозначены Pod’ы, которые в своем более простом представлении являются олицетворением запущенных Docker контейнеров. Жизнь Pod определяется ресурсом Deployment, которая ради упрощения на схеме не отображена.

Фигурой документа (прямоугольник с как-бы загнутым углом) изображены самописные ресурсы, опубликованные в облаке, которые создаются и описываются разработчиком. Для того, чтобы такой самописный ресурс опубликовать, необходимо создать описание самописного ресурса (Custom Resource Definition), который не был указан на схеме ради простоты и наглядности.

Под пользователем (User) подразумевается либо человек, либо другой микросервис, который использует опубликованное API для. Это использование выражено в отправлении HTTP-запросов в gateway, который в дальнейшем, при наличии соответствующего правила маршрутизации, перенаправит запрос в нужный микросервис.

Под администратором облака, подразумевается человек, который занимается сервисным обслуживанием микросервиса публикации API, а непосредственно публикацией этого API.

Глава 3 Модель инструмента управление прикладным программным интерфейсом на основе микросевисной архитектуры

3.1 Разработка модели инструмента управления прикладным программным интерфейсом

Архитектура разрабатываемого программного продукта будет состоять из трёх модулей: API-шлюз, API-публикатор, модуль развертывания. API-шлюз будет обрабатывать запросы и перенаправлять их в соответствии с конфигурацией, которую получит от API-публикатора. А модуль развертывания обеспечит легкую установка всего программного продукта в облако.

На начальных этапах проектирования приложения была создана диаграмма вариантов использования. Диаграмма вариантов использования описывает функциональное назначение моделируемой системы. Каждый вариант использования последовательность действий, которые должны быть выполнены системой.

Варианты использования представляют собой овал с текстом сценария. Участие актера в системе моделируется линией между актером и сценарием использования. Для изображения границы системы используют рамку вокруг самого варианта использования.

Диаграммы вариантов использования идеально подходят для:

– изображения целей взаимодействия системы с пользователем;
– определение и организация функциональных требований к системе;
– указание контекста и требований системы;
– моделирование основного потока событий в сценарии использования.

Диаграмма вариантов использования продукта управления API представлена на рисунке 15. Система содержит 4 действующих лица.

В качестве пользователя может выступать, как и внешний пользователь облачного программного продукта, так и внутренний микросервис. Использование это заключается в вызове конкретного API, которое настроено в API шлюзе. Задача администратора облака опубликовать конфигурацию для API-шлюза в облаке. Формат конфигурации будет описан далее. Далее архитектор облачного программного решения может вручную написать файл конфигурации API и предоставить его администратору облака на публикацию. Разработчику предоставляется библиотека, используя которую он может аннотировать свой исходный код метаданными, по которым можно будет сгенерировать конфигурацию API, готовую для публикации в облаке.

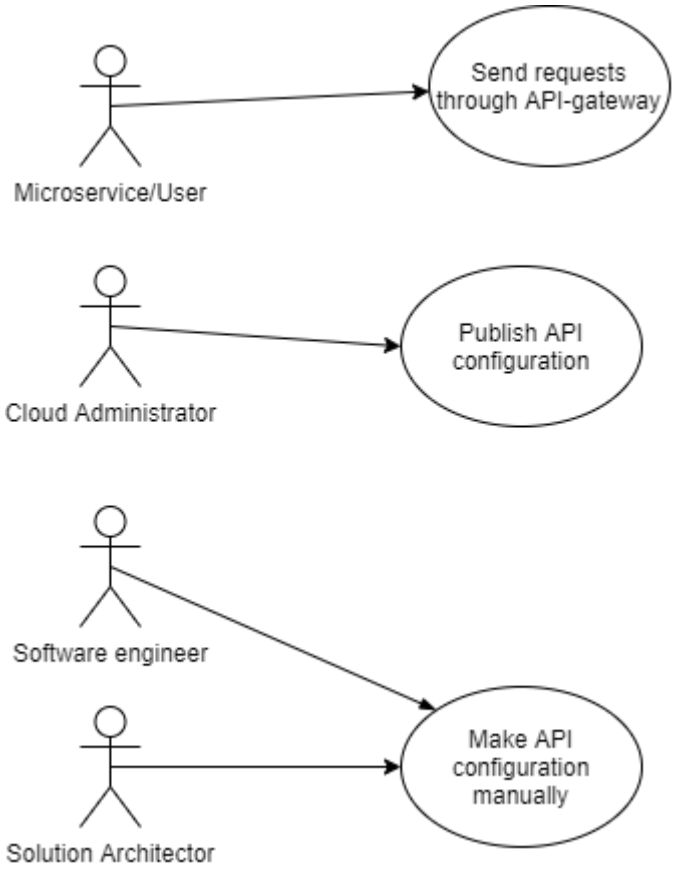

Рисунок 15 - Диаграмма вариантов использования

Далее была спроектирована диаграмма сущностных классов для того, чтобы выделить основные сущности для работы системы.

«Класс-сущность» - объекты сущностных классов представляют собой блоки длительно хранимой информации, используемые для организации баз

данных и знаний, файловых систем хранения, данных различной логической структуры в основном в этих классах развит атрибутный раздел, однако имеется небольшое число операций контроля ограничений целостности, как стандартных, так и специфичных для данной предметной области.

Диаграммы классов предлагают ряд преимуществ, что позволяет использовать для следующих задач:

- изображение модели данных для информационных систем, независимо от того, насколько они просты или сложны;
- лучшего понимания программной архитектуры приложения;
- визуально выражения любых специфических потребностей системы;
- создания подробных диаграммы, для выделения классов, которые необходимо реализовать в описанной структуре;
- предоставить независимое от реализации описание типов, используемых в системе, которые впоследствии передаются между ее компонентами.

Диаграмма классов в первую очередь предназначена для разработчиков, чтобы обеспечить концептуальную модель и архитектуру разрабатываемой системы. Как правило, диаграмма классов состоит из более чем одного класса или всех созданных классов для системы. Для конфигурации API в программном продукте выделены следующие сущности: «APIConfig», «APIConfigSpec», «Route», «Destination», «Rule», «Match», «MatchHeader».

Наличие в данной диаграмме сущностей «APIConfig» и «APIConfigSpec» на первый взгляд является избыточным, однако спецификация платформы Kubernetess по расширению обязывает нас следовать именно такому способу разделения. Сущность «APIConfig» в данном случае олицетворяет сущность с набором метаданных конфигурации, включая имя, тип, а также наличие ключа группа + версия конфигурации для осуществления встроенных возможностей по валидации конфигурации.

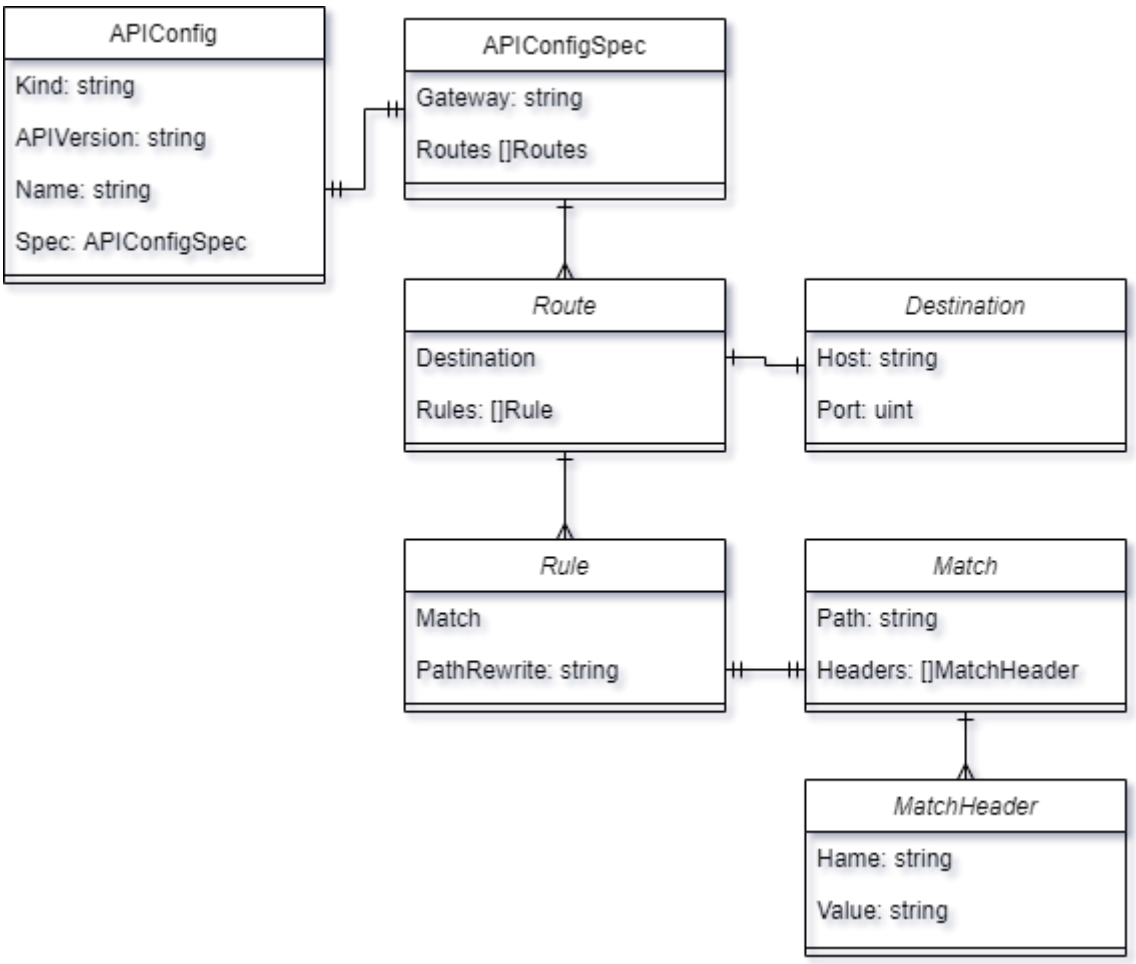

Рисунок 16 - Диаграмма сущностей API конфигурации

Сущность «ApiConfigSpec» является корневой сущностью, в которой располагается список правил маршрутизации, а также указано имя шлюза, к которому будет применена данная конфигурация. На основе конфигурации API опубликованной в облаке в формате YAML будет сформирована сущность «ApiConfigSpec» включая все остальные сущности, расположенные по иерархии ниже.

Сущность «Route» агрегирует набор правил по определению HTTP запросов, связывая их с конкретной точкой назначения. Другими словами, определят маршрут для HTTP запросов.

Сущность «Rule» описывает правило, по которому будут фильтроваться внешние запросы для дальнейшей маршрутизации. Сущность «Match» в данном случае определяет критерий, по которому будет выбираться HTTP

запрос, а поле «PathRewrite» определяет модификацию пути HTTP запроса. На конечную точку «Destination» запрос придет уже с другим Path если поле «PathRewrite» имеет конкретное значение.

Сущность «MatchHeader» содержит описание заголовка в HTTP запроса, на который должна производится дальнейшая маршрутизация сетевого трафика. Сущность включает имя заголовка и его значение. На рис. 17 и 18 изображены упрощенные диаграммы, описывающие внутреннюю структуру конфигурации EnvoyProxy.

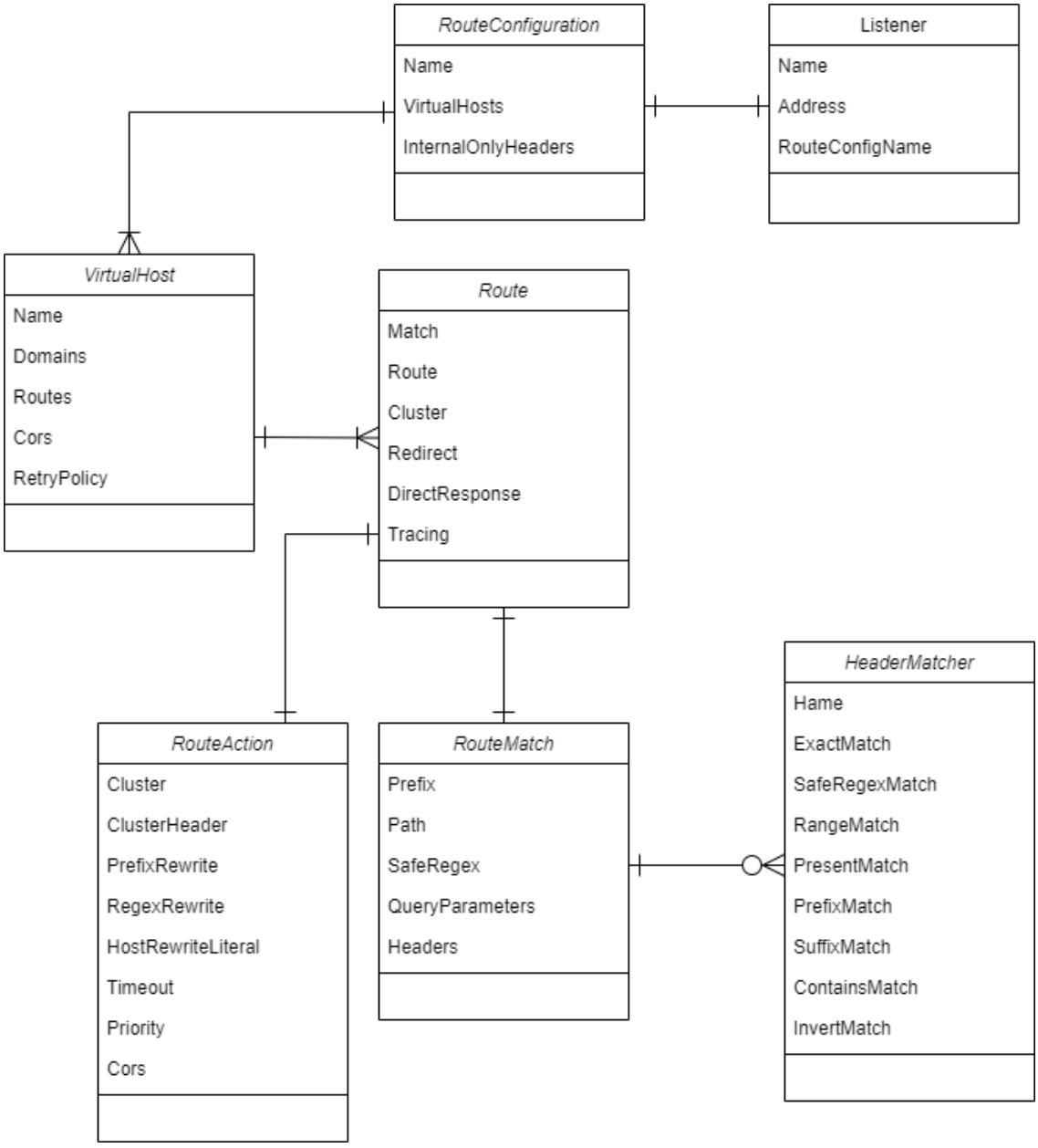

Рисунок 17 – Диаграмма сущностей EnvoyProxy - конфигурация маршрутов

Эта структура сложна и обширна, но в нашем случае не нужно использовать все те обширные возможности, которые поставляет компонент. Недостающая часть значении, которые будет невозможно достать из структуры на рис. 16 будет заполнятся подобранными значениями по умолчанию.

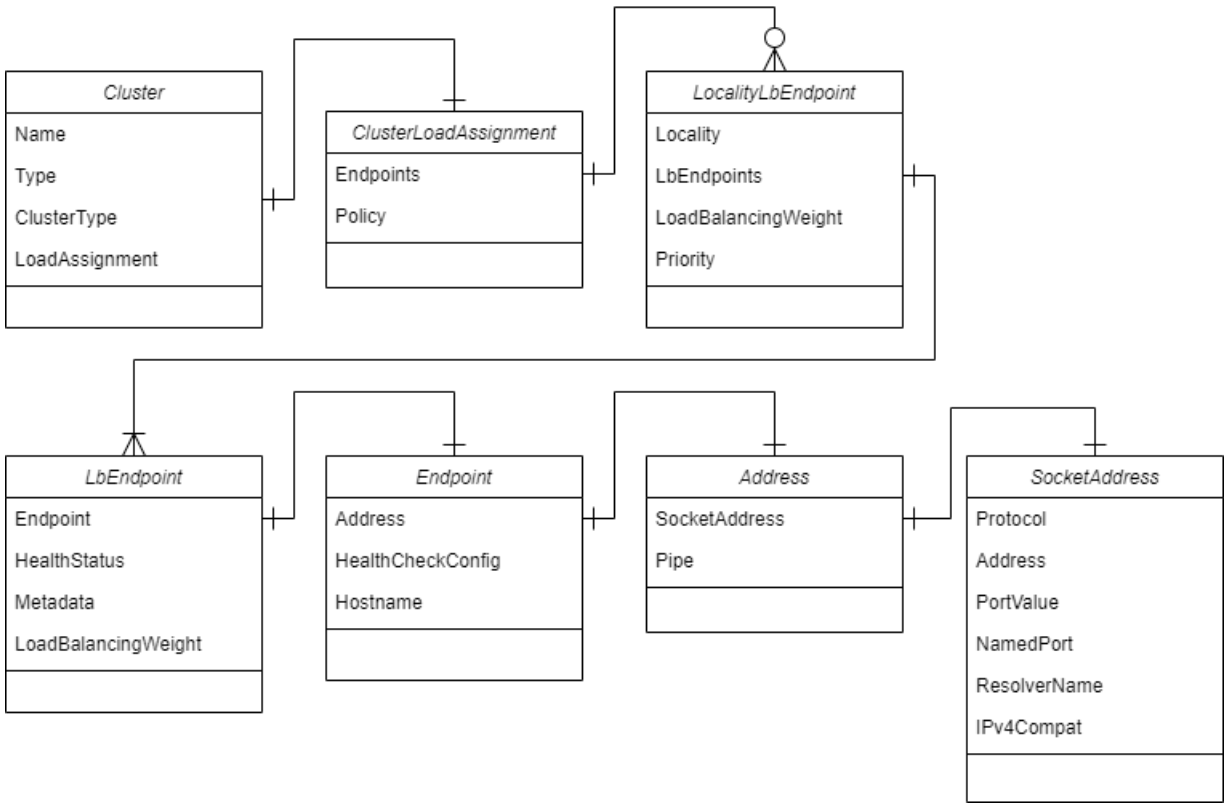

Рисунок 18 – Диаграмма сущностей конфигурации EnvoyProxy - конфигурация кластеров

Структура инструмента будет разбита на компоненты с четко разделенными обязанностями.

На рис. 19 изображена диаграмма компонентов и их способы взаимодействия между собой.

Компонент «Operator» отвечает за разворачивание API-publisher и API-gateway в облачной среде. Если рассматривать «Operator» более детально, то это компонент, задача которого упростить настройку двух других компонентов.

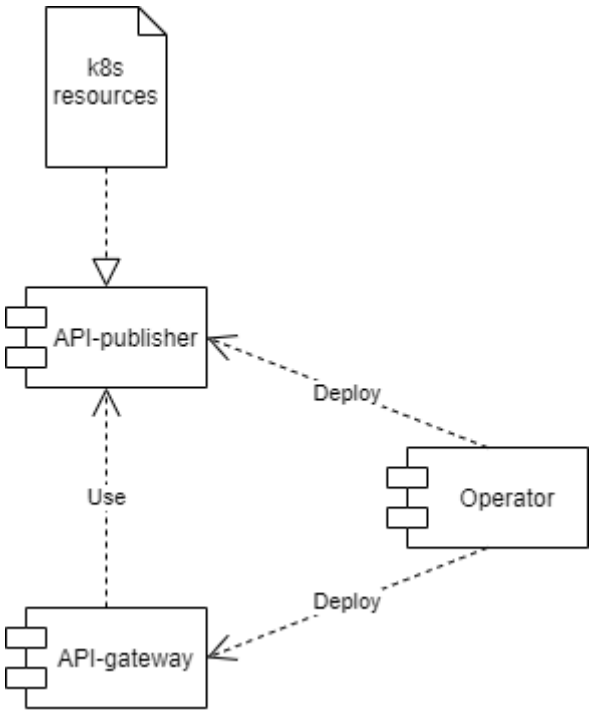

Рисунок 19 - Диаграмма компонетов

Можно было бы сказать, что он управляет жизненным циклом публикатора и шлюза, однако это не совсем верно. Да, оператор способен удалять, обновлять и создавать публикаторы и шлюзы, однако это не жизненный цикл контейнера. Было бы правильно сформулировать эту возможность как управление жизненным циклом конфигурации развертывания компонентов API-publisher и API-gateway.

Рассмотрим детальнее непосредственно алгоритм управления прикладным программным интерфейсом.

Его можно разделить на две части. В одной будет изображено формирование конфигурации для API-gateway (см. Рисунок 20) в другой, процесс получения конфигурации компонентом API-gateway из API-publisher (см. Рисунок 21).

Как можно видеть из рисунка 20 алгоритм основывается на распространенном шаблоне проектирования Observer. Компонент API-publisher подключается к слою Kubernetes через Kubernetes API с целью отслеживания происходящих событий с ресурсом APIConfig.

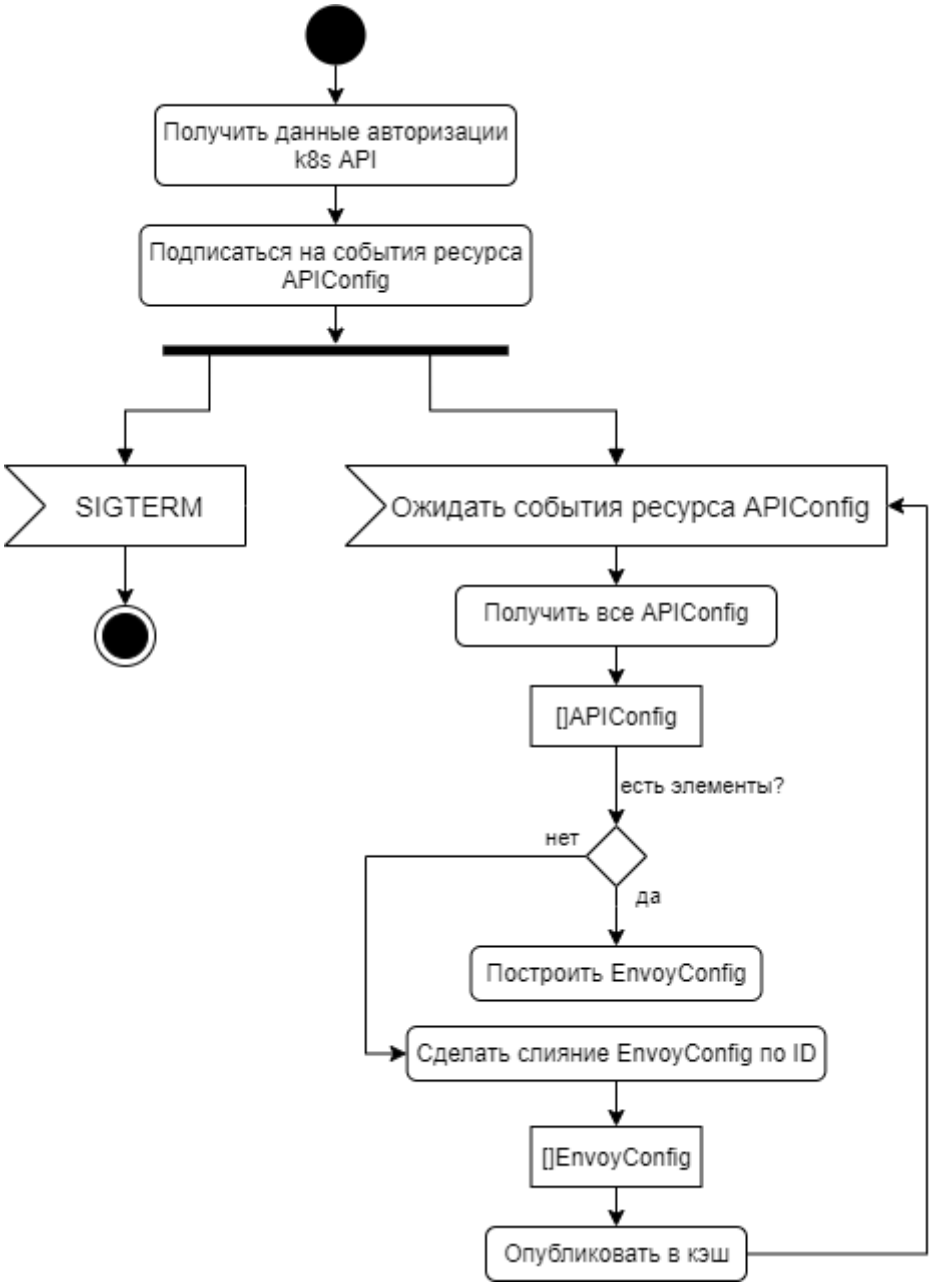

Рисунок 20 – Алгоритм формирования API-gateway конфигурации

После успешной подписки поток выполнения разделяется на две части, одна ожидает завершения работы приложения, а другая ожидает событий (Events) от Kubernetes API. Когда событие от Kubernetes API поступает в API-publisher происходит получение всех ресурсов типа APIConfig в том же пространстве имен (namespace) где расположен API-publisher. Далее, для

каждого найденного ресурса формируется полный набор сущностей, необходимый для работы API-gateway. Далее сформированные конфигурации объединяются в уникальный набор (сущности с одним именем объединяются), после чего опубликовываются в кэш памяти API-publisher. Стоит отметить, что кэш конфигураций для API-gateway предоставляется библиотекой go-control-plane из коробки.

Далее на рисунке 21 изображен алгоритм получения конфигурации компонентом API-gateway из API-publisher.

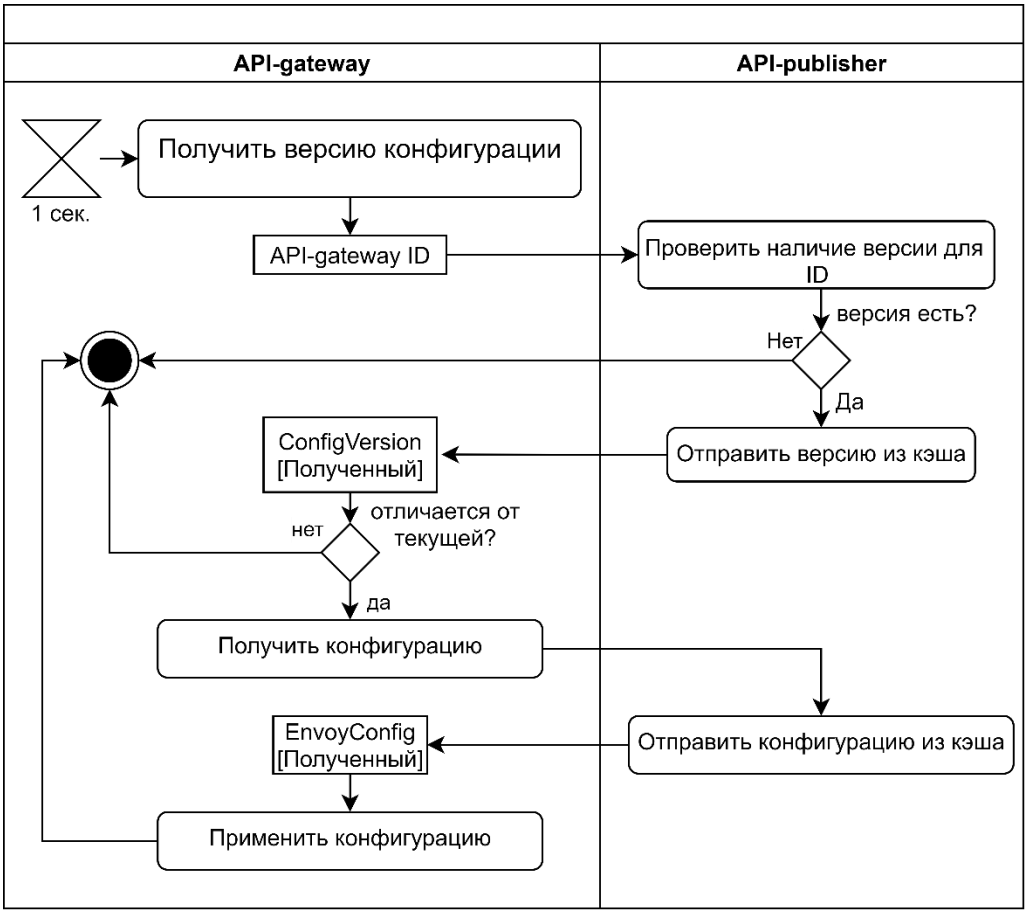

Рисунок 21 – Алгоритм получения конфигурации из API-publisher

В данном изображении алгоритма опущены детали сериализации / десериализации и особенности взаимодействия компонентов между собой по протоколу gRPC. API-gateway настроен так, что каждый промежуток времени (1 сек.) он отправляет запрос в API-publisher на проверку версии

конфигурации, опубликованной в нем. В запросе отправляется идентификатор API-gateway. После получении версии, если такая есть, API-gateway сравнивает её с уже известной ему, и в том случае если версии различаются, начинается процесс получения конфигурации.

3.2 Разработка инструмента управлением прикладным программным интерфейсом микросервисов

3.2.1 Разработка компонента API-gateway

Компонент “API-gateway” выполняет функции маршрутизации HTTP трафика. В качестве основы этого компонента был выбран программный продукт «Envoyproxy». Разработчик «Envoyproxy» опубликовывает программный продукт в разных формах, включая готовые к запуску бинарные файлы и формат докер-контейнера. В реализации компонента «API-gateway» мы возьмем за основу «envoyproxy/envoy-alpine:v1.17.0». На текущий момент версия 1.17.0 является последней версией «Envoyproxy». Приписка «alpine» указывает на то, что этот образ основан на минималистично возможной версии ОС Linux – Alpine.

Докер-образ «Envoyproxy» предоставляется без конфигурации и подразумевает что пользователь её напишет и поместит внутрь этого образа. Наша задача указать «Envoyproxy» откуда забирать конфигурацию после старта.

Структура конфигурации «Envoyproxy» имеет свою специфику. Для начала настроем страницу администрирования, чтобы было просто посмотреть текущее состояние приложения в контейнере, а также увидеть полный слепок текущей его конфигурации. На рис. 22 изображена часть конфигурации, отвечающая за настройку страницы администрирования. С такой конфигурацией «Envoyproxy» будет прослушивать на локальном хосту порт 9903 в режиме совместимости со стандартом IPv4.

Для того чтобы «Envoyproxy» загружал конфигурацию из компонента «API-publisher» по протоколу gRPC нужно воспользоваться секцией «dynamic_resources».

```
admin:
  access_log_path: /dev/null
  address:
    socket_address: { address: 0.0.0.0, port_value: 9903, ipv4_compat: true }
```

Рисунок 22 - Конфигурация страницы администрирования «Envoyproxy»

На рис. 23 изображена конфигурация, в которой описано, что брать сущности Cluster и Listener нужно из кластера с именем «xds_cluster». Также указано что источник конфигурации «xds_cluster» работает с ресурсами версии «V3», использует протокол «GRPC».

```
dynamic_resources:
  cds_config:
    resource_api_version: V3
    api_config_source:
      api_type: GRPC
      transport_api_version: V3
      grpc_services:
        - envoy_grpc:
            cluster_name: xds_cluster
  lds_config:
    resource_api_version: V3
    api_config_source:
      api_type: GRPC
      transport_api_version: V3
      grpc_services:
        - envoy_grpc:
            cluster_name: xds_cluster
```

Рисунок 23 - Конфигурация динамических ресурсов «Envoyproxy»

Следующим шагом объявим в конфигурации кластер, который мы упомянули в секции динамических ресурсов выше. Эта настройка объявляется в статической секции конфигурации, как заранее известная и неизменяемая. На рис. 24 изображена полная конфигурация кластера «xds_cluster».

```
static_resources:
  clusters:
  - name: xds_cluster
    connect_timeout: 1s
    type: STRICT_DNS
    lb_policy: ROUND_ROBIN
    typed_extension_protocol_options:
      envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
        "@type": type.googleapis.com/envoy.extensions.upstreams.http.v3.HttpProto
colOptions
        explicit_http_config:
          http2_protocol_options:
            connection_keepalive:
              interval: 30s
              timeout: 5s
    load_assignment:
      cluster_name: xds_cluster
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: api-publisher
                port_value: 15010
```

Рисунок 24 - Конфигурация источника конфигурации «Envoyproxy»

Она включает множество настроек уровня протокола транспорта, однако для нас важно чтобы в качестве типа кластера был указан «STRICT_DNS», что позволяет указывать в качестве адреса конечной точки не IP адрес, а DNS имя, что очень важно, учитывая, что в облаке у сервиса IP адрес может меняться от перезапуска и назначается динамически. В качестве адреса укажем имя сервиса «api-publisher», а привязку конкретного запущенного контейнера привяжем к этому имени соответствующей конфигурацией уже уровня PaaS. Также далее в качестве номера порта укажем «15010» и зафиксируем это

значение, чтобы потом при реализации компонента «api-publisher» прослушивать именно этот порт.

Далее нам необходимо создать свой докер-образ, в котором мы скопируем конфигурацию и укажем по какому пути в контейнере она расположена, а также привяжем запущенный экземпляр «Envoyproxy» к кластеру. Привязка к кластеру нужна для того, чтобы при увеличении количества запущенных экземпляров «Envoyproxy», каждый экземпляр забирал из «api-publisher» одну и ту же конфигурацию.

Для запуска «Envoyproxy» напишем простой shell-скрипт, который добавляет несколько параметров для пометки с каким именем кластера (--service-cluster) запустить экземпляр, с каким именем узла (--service-node) и в каком каталоге расположен статический файл конфигурации, написанный нами ранее. «Envoyproxy» будет передавать в запросе указанные параметры узла и кластера при обращении к источнику конфигурации (api-publisher).

```
#!/bin/sh

envoy --service-cluster "${POD_HOSTNAME}" --service-node "${POD_HOSTNAME}" -
c '/envoy/envoy.yaml'
```

Рисунок 25 – Shell скрипт запуска «Envoyproxy»

Файл конфигурации и скрипт запуска – это всё что необходимо, чтобы создать свой докер-образ.

На рис. 26 изображена готовая конфигурация для запуска сервиса «api-gatewa» в облачной среде, прокомментируем её. На первой строке указано из какого докер-образа собрать текущий.

Далее мы внутри образа устанавливаем утилиты gettext и coreutils для удобства использования контейнера.

Следующая инструкция добавляет внутрь образа переменную окружения «ENVOY_UID» со значением ноль, что запущенный «Envoyproxy» распознает как флаг для работы от имени корневого пользователя.

```
FROM envoyproxy/envoy-alpine:v1.17.0

RUN apk --no-cache add gettext coreutils

ENV ENVOY_UID=0
USER 10001

COPY config.yaml /envoy/envoy.yaml
COPY run-gateway.sh /envoy/run-gateway.sh

EXPOSE 8080 9903

USER root
RUN chmod -R 777 /envoy
USER 10001

CMD ["/envoy/run-gateway.sh"]
```

Рисунок 26 – Dockerfile для создания образа «api-gateway»

Далее идет пометка, что скрипт сборки докер-образа работает от имени пользователя «1001» это необходимо для того, чтобы контейнер из этого докер образа мог работать в среде «Kubernetes», где политиками безопасности запрещено запускать докер-контейнеры от имени root пользователя. Следующие команды копируют файл конфигурации и скрипт в каталог /envoy в образе. Далее делается пометка, что контейнер с открытыми портами 8080 и 9903. Данная команда в среде «Kubernetess» не имеет значения, так как такие настройки задаются на уровне конфигурации развертывания, однако вне PaaS среды без этих настроек docker-container не будет работать так, как ожидается. Следующей парой осуществляется выдача прав на каталог /envoy в образе в формате «полный доступ для всех». На последней строке мы указываем каким образом контейнер будет запускаться, какой командой. И в качестве этой команды указываем написанный нами скрипт.

3.2.2 Разработка компонента API-publisher

Задача компонента API-publisher следить за тем была ли опубликована в Kubernetes конфигурация, описывающая маршруты сети и если такова есть опубликовать её в компоненте API-gateway.

Для успешной реализации этого компонента необходимо решить три проблемы:

- описать схему конфигурации используя за основу диаграмму сущностей из раздела 0;
- реализовать функцию отслеживания изменении конфигурации на облаке;
- реализовать реакцию на событие изменения конфигурации на облаке с последующей её конвертацией и публикацией в компоненте api-gateway.

Чтобы Kubernetes мог сохранять и обрабатывать конфигурации определенного нами формата нам необходимо сформировать описание нашего формата.

Kubernetes для самописных ресурсов требует сформировать ресурс типа «CustomResourceDefinition». Перед тем как его создавать стоит сформировать примерный формат ресурса, который мы будем использовать для описания нашей конфигурации, беря за основу модель, изображенную на рисунке 16.

На рис. 27 изображен полноценный пример такой конфигурации.

Он полностью соответствует описанной нами ранее модели, включая такие поля как «apiVersion», «kind», «metadata», «spec», которые добавлены исключительно из-за специфики Kubernetes, который требует наличие этих полей.

На основе конфигурации выше составим его описание.

На рис. 28 изображено описание ресурса конфигурации API, а сама схема, из-за громоздкости вынесена в приложение А.

Kubernetes требует от нас в описании ресурса указывать группу (group), тип ресурса (kind), а также список версий (versions) со схемами, позволяющие проверять загружаемые в Kubernetes ресурсы такого типа, группы и версии на достоверность.

Например, изображенная на рис. 27 конфигурация при публикации в Kubernetes будет проверена на достоверность при помощи схемы, описанной в приложении А.

```yaml
apiVersion: apimanagement.cloud/v1alpha1
kind: APIConfig
metadata:
  name: example-api-config
spec:
  gateway: gateway
  routes:
  - destination:
      address:
        host: service
        port: 8080
    rules:
    - match:
        path: /api/v1/service
        headers:
        - name: X-Header
          value: x-value
      pathRewrite: /
```

Рисунок 27 – Пример формата конфигурации, хранимой в Kubernetes.

Имея описание формата конфигурации для Kubernetes и саму конфигурацию, мы получаем возможность сохранять в Kubernetes конфигурации нашего формата.

Также теперь есть возможность отслеживать эти конфигурации, причем отслеживать любые изменения от удаления и заканчивая добавлением.

Однако, чтобы осуществить решение следующей проблемы, а именно отслеживанием, нужно наладить для нашего приложения коммуникацию с Kubernetes.

Для этого существует библиотека k8s.io/client-go, исходный код которой, а также документация расположены на ресурсе https://github.com/kubernetes/client-go.

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: apiconfigs.apimanagement.cloud
spec:
  group: apimanagement.cloud
  names:
    kind: APIConfig
    listKind: APIConfigList
    plural: apiconfigs
    shortNames:
    - apicfg
    singular: apiconfig
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema: …
    served: true
    storage: true
```

Рисунок 28 - Часть ресурса, описывающего конфигурацию API

Описание использование этой библиотеки в данной работу будет опущено в виду громоздкости.

Однако стоит отметить шаги, которые были предприняты, чтобы «научить» приложение отслеживать изменения конфигурации с помощью этой библиотеки.

Первым шагом было описание используемых структур в формате языка Golang (см. Рисунок 29).

```
11  // +resource:path=apimanagement
12  type APIConfig struct {
13      metav1.TypeMeta   `json:",inline"`
14      metav1.ObjectMeta `json:"metadata"`
15      Spec              APIConfigSpec `json:"spec"`
16  }
17
18  // +k8s:deepcopy-gen=true
19  // +k8s:deepcopy-gen:interfaces=k8s.io/apimachiner
20  // +resource:path=apimanagement
21  type APIConfigList struct {
22      metav1.TypeMeta `json:",inline"`
23      metav1.ListMeta `json:"metadata"`
24      Items           []APIConfig `json:"items"`
25  }
26
27  // APIConfigSpec is configuration of routes for AP
28  // +k8s:deepcopy-gen=true
29  type APIConfigSpec struct {
30      Gateway string  `json:"gateway"`
31      Routes  []Route `json:"routes"`
32  }
```

Рисунок 29 – Часть кода, описывающий структуры конфигурации

Далее, на основе этого было сгенерировано несколько клиентов, позволяющих работать с Kubernetes API и непосредственно отслеживать конфигурации.

Работа с клиентом достаточно проста, мы выбираем пространство имен, в котором будем осуществлять наблюдение за конфигурацией и получаем канал, по которому будем получать события.

Для реализации функции публикации конфигурации была использована библиотека github.com/envoyproxy/go-control-plane, позволившая использовать уже готовые структуры для создания конфигурации.

Для того чтобы Envoyproxy мог забрать публикуемую конфигурацию необходимо запустить сервер, обслуживающий gRPC запросы, а также зарегистрировать сервисы для обработки запросов на каждую из сущностей.

Алгоритм запуска приложения api-publisher достаточно прост и умещается на небольшой диаграмме активностей на рисунке 30.

Рисунок 30 – Алгоритм старта компонента api-publisher

Из алгоритма становится очевидно, что для работы компонента необходим HTTP-сервер и gRPC. Если с gRPC сервером всё прозрачно – он нужен для обслуживания запросов на получение конфигурации от Envoyproxy, то с наличием HTTP-сервера возникают закономерные вопросы.

HTTP-сервер необходим для того, чтобы Kubernetes мог отслеживать состояние компонента.

Для этих целей мы не просто запустили сервер, обслуживающий любые HTTP запросы на порту 8080, а также дополнительно обрабатываем запросы, поступающие по пути /health.

Это для того, чтобы вернуть актуально состояние запущенного компонента и в случае необходимости дать Kubernetes понять, что с нашим приложением что-то не так, когда это действительно так.

Алгоритм, обрабатывающий события от Kubernetes изображен на рисунке 31.

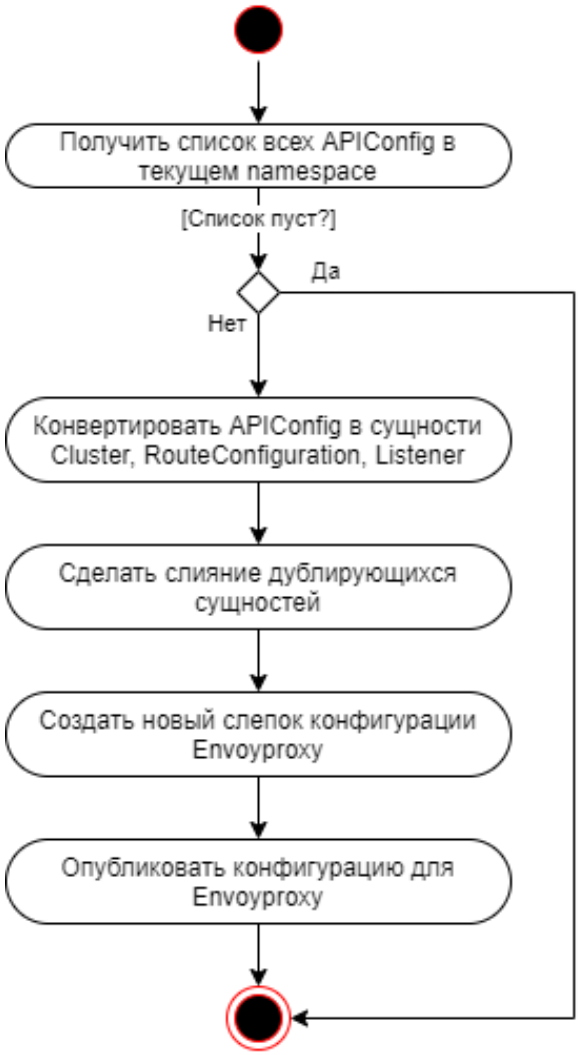

Рисунок 31 – Алгоритм обработчика события изменения ресурса

Другая часть логики компонента api-publisher, на которой стоит обратить внимание – обработка события изменения ресурса.

После того, как Kubernetes оповестит наше приложение у нас есть несколько вариантов как поступить.

Первое – обработать тип события (добавление, изменение, удаление) и на основе типа поступить соответствующим образом, преобразовав конфигурацию для Envoyproxy.

Однако так как слепок данных конфигурации для Envoyproxy нам изменять нельзя.

Это связано с ограничением библиотеки go-control-plane, а также спецификой работы протокола получения динамической конфигурации.

Принято решение игнорировать тип события и поступить проще – доставать на событие все конфигурации, существующие в определенном пространстве имен Kubernetes и конвертировать их в готовую конфигурацию Envoyproxy.

Затем опубликовывать, чтобы Envoyproxy мог его беспрепятственно забрать.

3.3.3 Разработка компонента operator

За основу компонента operator мы возьмем реализацию shell-operator, однако допишем своим функционалом.

Shell-operator предназначен для того чтобы на основе базового механизма наблюдения за изменениями конфигураций предоставляемого Kubernetes делать свои реализации шаблона проектирования «operator».

Основой расширения shell-operator являются так называемые крюки (hook).

Каждый «крюк» — это скрипт, который при вызове с параметром – config возвращает конфигурацию, на основе которой shell-operator понимает, когда основную часть скрипта стоит запускать.

На текущем этапе проектирования достаточно чтобы operator при старте конфигурации развертывания для остальных компонентов.

По причине массивности этих конфигураций полная их версия в данной работе приводится не будет, однако весь исходный код, а также файлы конфигурации и скрипты развертывания и запуска опубликованы по адресу https://github.com/dmol5E/api-management-app.

На рис. 32 изображен текст скрипта создания докер-образа компонента operator.

```
FROM flant/shell-operator:v1.0.0-rc.1
ADD hooks /hooks
ADD deploy /deploy
RUN chmod -R 777 /hooks
RUN chmod -R 777 /var/run/

USER 10001
```

Рисунок 32 – Dockerfile компонента operator

Как было сказано ранее компонент основан на shell-operator и по инструкции из документации созданы директории /hooks и команда в строчке 2 копирует скрипты в этот каталог и далее раздаются права, чтобы были права запускать эти скрипты.

Глава 4 Апробация модели инструмента управления прикладным программным интерфейсом

4.1 Процесс внедрения инструмента управления прикладным программным интерфейсом

Процесс внедрения инструмента управления прикладным программным интерфейсом представляет собой предварительную настройку окружения, для развертывания компонента «operator» и непосредственно самого развертывания компонента «operator». Далее сам компонент «operator» управляет дальнейшим процессом развертывания остальных составных частей инструмента управления прикладным программным интерфейсом, а именно компонентов «api-gateway» и «api-publisher». Развертывание инструмента показано на рисунке 33.

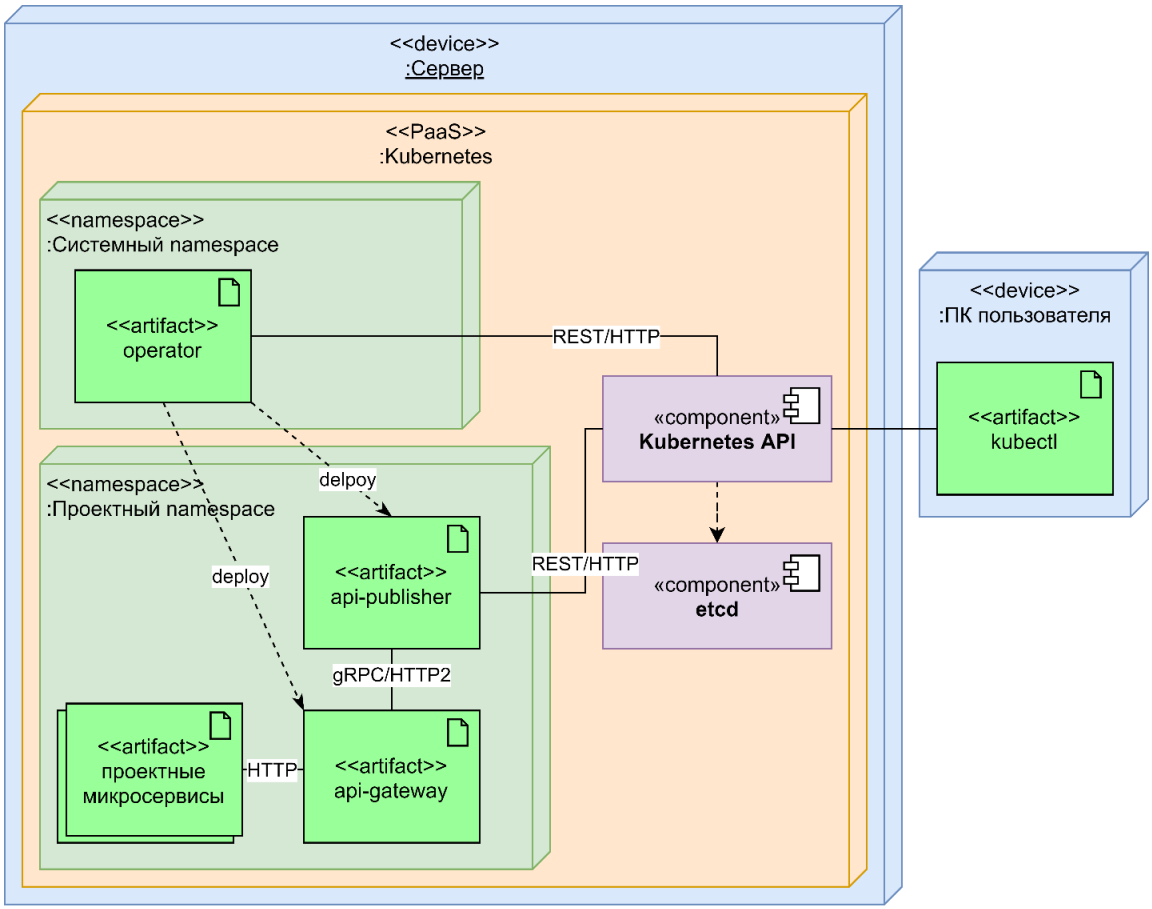

Рисунок 33 – Диаграмма развертывания

Прежде чем описывать диаграмму развертывания сказать, что диаграмма развертывания – это один из типов диаграмм UML стандарта, созданный с целью отобразить отношение между выполняемыми компонентами программного продукта и физической вычислительной системы, отобразить отношение компонентов в окружении. Диаграмма развертывания показывает архитектуру системы выполнения, включая такие узлы как аппаратные или программные среды выполнения, а также промежуточное программное обеспечение, как компонент, соединяющий их.

Диаграммы развертывания используются для визуализации программного обеспечения и непосредственно физического оборудования. Используя этот тип диаграмм из стандарта UML, можно понять как программный продукт будет физически развернут на оборудовании. Также диаграммы развертыванию подходят для моделирования топологии оборудования системы, если их сравнивать с другими типами диаграмм UML, которые описывают в основном логические компоненты системы.

Разработанный инструмент имеет узлы сервер и клиент. Клиент, это человек, который используя клиентскую утилиту kubectl осуществляет публикацию ресурсов непосредственно в Kubernetess. На сервере развернут программный продукт Kubernetes со своими компонентами Kubernetes API и etcd. Здесь etcd это надежное хранилище данных типа «ключ-значение», а Kubernetes API это прикладной программный интерфейс, основанный на протоколе HTTP и предоставляющий возможности расширения платформы самописный ресурсами и их обработчиками. Также внутри Kubernetes существует разделение на пространства имен, позволяющее организовывать развёртывание разных проектов внутри одного Kubernetes и избежать конфликтов имен. Артефакт operator развернут в отдельном namespace, чтобы иметь возможность разворачивать остальные компоненты в любом другом. По факту каждый артефакт олицетворяет Pod – ресурс Kubernetes, расширяющий докер-контейнеры и представляет собой запущенный экземпляр приложения.

Каждый запущенный сервис представлен на вкладке Pods. Скриншот вкладки представлен на рисунке 34.

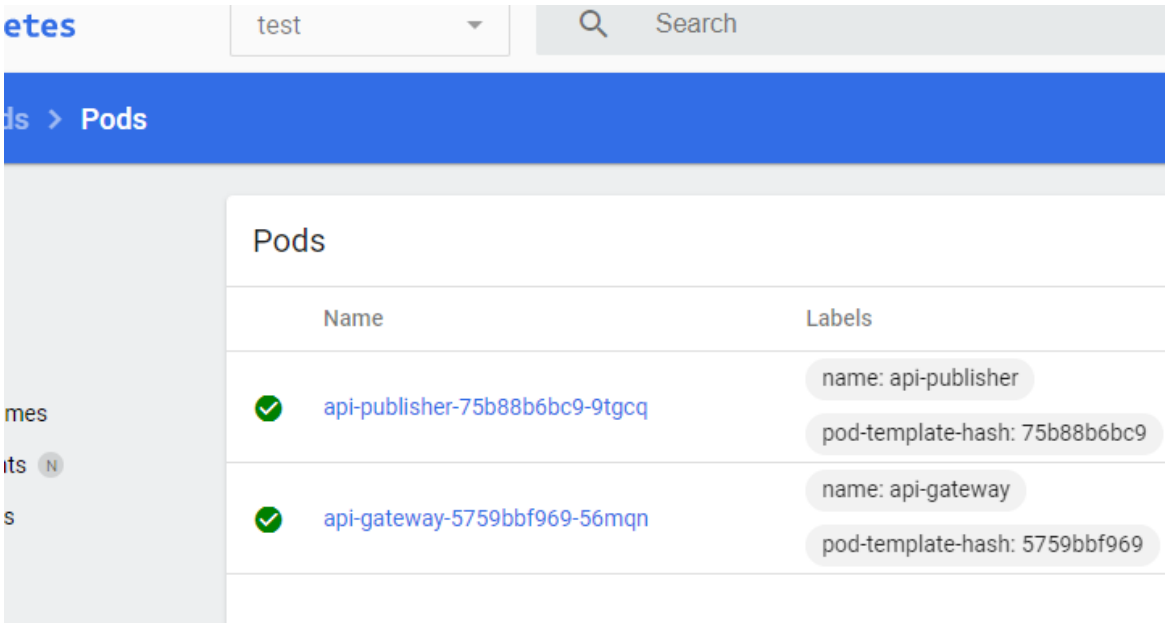

Рисунок 34 – Список сервисов системы

После стадии успешного развертывания программного продукта необходимо провести полноценное функциональное тестирования, чтобы убедиться в его работоспособности. Функциональное тестирование используется для доказательства того, что функции программного обеспечения имеют ожидаемый результат, требуемый конечному пользователю или бизнесу. Функциональное тестирование, как правило, включает оценку каждой функции ПО и сравнение с требованиями бизнеса. ПО тестируется путем предоставления ему некоторых входных данных, чтобы оценить, как оно соответствует, связано или изменяется по сравнению с его базовыми требованиями. Помимо того, функциональное тестирование помогает раскрыть проблемы использования ПО и оценить его удобство применения. Методы функционального тестирования включают интеграционное тестирование, тестирование белого ящика, тестирование черного ящика, модульное тестирование (оно же unit тестирование),

приемочное тестирование и т. д. Инструмент тестировался при помощи модульного, интеграционного и системного тестирования. Последовательность тестирование отображена на рисунке 35.

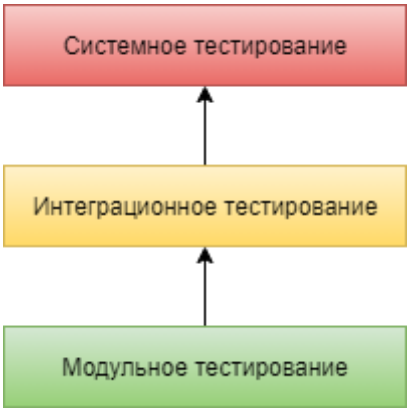

Рисунок 35 – Этапы тестирования

Модульное тестирование – это уровень тестирования ПО, на котором производится тестирование отдельных модулей или компонентов программного обеспечения. Основная цель состоит в том, чтобы проверить, что каждая минимальная единица программного обеспечения работает так, как задумано. Под минимальной единицей ПО понимается самая маленькая тестируемая часть любого программного обеспечения. Так как мы используем язык программирования golang, то в нашем случае минимальной частью является функция. В рамках модульного тестирования тестами были покрыто большая часть написанных функций. Благодаря этим тестам мы сможем исключить ошибки компиляции, внесенные после исправлений или после разработки нового функционала.

На следующем этапе мы разработали интеграционные тесты. На этом уровне тестируются уже взаимодействие разных наборов компонентов программного обеспечения между собой в вычислительном окружении, приближенном к реальным условиям. Целью интеграционного тестирования является обнаружение ошибок во взаимодействии набора компонентов между собой. В нашем случае точкой входа для тестирования являлась публикация

ресурсов чере Kubernetes API, а также проверка правильности маршрута, путем отправки HTTP запроса через api-gateway. Этот этап тестирования позволил выявить ошибки при формировании маршрутов в api-gateway и исправить их путем их сортировки по критерию длинны. Самые «широкие» маршруты, например «/» (корневой) располагаются ниже, в то время как более конкретные, например «/api/v1/service/endpoint1/resourceid/operation» располагаются выше. В качестве основы интеграционных тестов был выбран набор инструментов и библиотек Arquillian.

Arquillian — это платформа для интеграционного тестирования. Главным её преимуществом является легкость и быстрота написания тестов, как если бы это были модульные тесты.

Следующим этапом было проведено системное тестирование, в процессе которого уже было проверено межсервисное взаимодействие. Оно включало в себя:

– тестирование инструмента с развернутым рядом проектом, публикация ресурсов и конфигураций, с целью проверить как компоненты взаимодействуют друг с другом и со средой выполнения;
– проверка, что настроенный API-gateway пропускает сетевой трафик в соответствии с ожидаемым.

В результате системного тестирования были выявлены ошибки при обработке конфигурации. Наличие нескольких конфигураций в одном пространстве имен производило к обработке только одной, в то время как ожидалось их слияние. Также был учтено возможное наличие двух одинаковых конфигураций, но с разными именами, их наличие теперь приводит к корректному слиянию и устранению дубликатов.

Также было решено сделать нагрузочное тестирование. Такое тестирование проверяет стабильность, масштабируемость, скорость, а также надежность программного обеспечения. Нагрузочные тесты могут

ориентироваться на разные показатели. В нашем случае показателями является время обновления конфигурации от момента её публикации, до актуального состояния компонента api-gateway, а также количество конфигураций, обрабатываемых за приемлемое время. В результате тестирования было выявлено большое потребление памяти компонентом api-publisher, несмотря на то что количество конфигураций по тестовому сценарию увеличивалось и уменьшалось, память в этом компоненте только росла. На рисунке 36 изображен график зависимости потребления памяти компонентом api-publisher по отношению к количеству конфигураций фиксированного размера ~10Kb. Причиной этого часто являются утечки памяти.

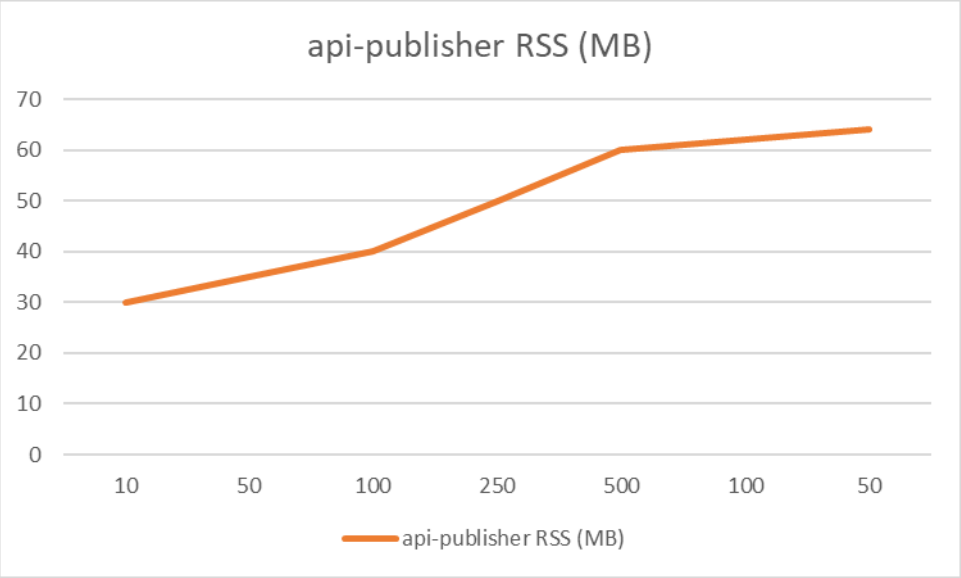

Рисунок 36 – Диаграмма потребления памяти компонентом api-publisher на нагрузочном тестировании

Однако после тщательного анализа и инструментов профилирования мы пришли к выводу, что проблема в специфике работы сборщика мусора в приложениях написанных на golang. Решением проблемы было выставление переменной окружения GOGC для контейнера с сервисом api-publisher в более

низкое значение, что заставляло сборщика мусора работать в более агрессивно.

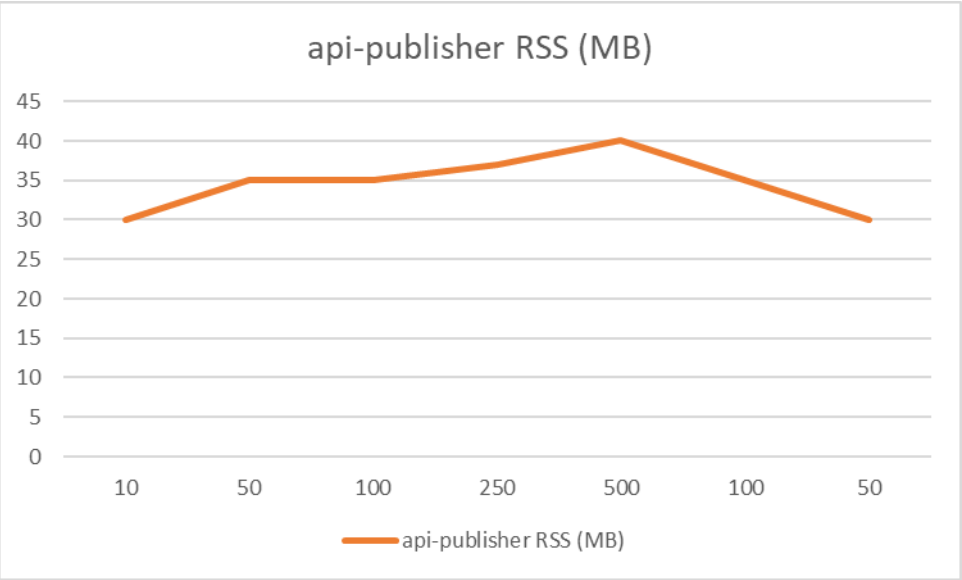

Рисунок 37 – Потребление памяти компонентом api-publisher после оптимизации

После тонкой настройки сборщика мусора в приложении api-publisher график потребления памяти изменился и стал выглядеть более объективно и логично.

4.2 Процесс интеграции инструмента прикладного программного интерфейса с готовым проектным решением

В рамках процесса интеграции подразумевается, что инструмент управления прикладного программного интерфейса уже развернут в PaaS инфраструктуре рядом с интегрируемым приложением. Далее в зависимости от такого какое API у интегрируемых микросервисов нужно составить конфигурацию.

Также, как часть интеграции придется в микросервисах использовать коммуникацию к другим микро-сервисам через шлюз, а это подразумевает доработку микросервисов.

Однако если инструмент будет использоваться для внешних запросов, то максиму что нужно будет добавить, так это Ingress ресурс, чтобы «открыть» шлюз для запросов вне облака.

```yaml
apiVersion: apimanagement.cloud/v1alpha1
kind: APIConfig
metadata:
  name: example-api-config
spec:
  gateway: gateway
  routes:
  - destination:
      address:
        host: service
        port: 8080
    rules:
    - match:
        path: /api/v1/service
        headers:
        - name: X-Header
          value: x-value
      pathRewrite: /
```

Рисунок 38 – Пример конфигурации API

Подсчитаем потенциальные трудозатраты по интеграции инструмента с существующим проектным решением на микросервисной архитектуре.

Таблица 2 – Трудозатраты на интеграцию

	Разработчик	Старший-разработчик
Развертывание	20 минут	10 минут
Написание конфигурации	~4 часа	~2 час
Публикация конфигурации	1 минута	1 минута

На основе данных таблицы 2 можно сделать вывод, что максимальное время, необходимое на интеграцию равно 5 часам, что является очень коротким периодом времени между решением интегрироваться и выходом в использование.

4.3 Расчет надежности разработанного инструмента

Критерием качества разрабатываемого прикладного программного обеспечения является степень её надежности.

Существует множество примеров, когда разработанная система падала, что приводило к огромным финансовым убыткам компании.

Чтобы избежать такого исхода необходимо оценить надежность разработанной системы.

Существующие модели оценки надежности ПО объединены в три группы:

– статические модели (не учитывают время появления сбоя);
– дискретные динамические модели (ошибки устраняются);
– непрерывные динамические модели (во время тестирования ошибки не исправляют).

Выберем дискретные динамические модели, так как они позволяют в процессе тестирования устранить ошибки.

В качестве такой модели выберем модель Шумана.

Она строится на том, что оценка надежности ПО проводится на основании проведенного тестирования, что удобно, так как тестирование мы уже провели.

Тестирование производится поэтапно и по завершению этапа все выявленные проблемы исправляются и начинается следующий этап тестирования.

Было проведено 4 этапа тестирования. Первый этап продолжался 7 часов, затем второй – 5 часов, третий – 8 часов и последний – 3 часа. На этих этапах было выявлено 3, 2, 2, 0 ошибок.

Полученные результаты приведены в таблице 3.

Таблица 3 – Результаты тестирования

Этап (k)	Продолжительность (t), ч	Количество сбоев (m)
1	7	3
2	5	2
3	8	2
4	3	0

Общее время тестирования будет равно T = 23 часа.

Общее число обнаруженных и исправленных ошибок n = 7.

Количество ошибок, исправленных к началу (i + 1) этапа равно $n_0 = 0$; $n_1 = 0$; $n_2 = 3$; $n_3 = 5$; $n_4 = 7$.

Функция надежности по модели Шумана описывается следующей формулой:

$$R_i(t) = e^{-\lambda_i t}, \quad (4.1)$$

где λ – интенсивность появления ошибок, которая вычисляется по формуле

$$\lambda = (N - n) * C, \quad (4.2)$$

где N – первоначальное количество ошибок;

C – коэффициент пропорциональности, который равен

$$C = \frac{\sum_{i=1}^{k} \frac{m_i}{N - n_{i-1}}}{\sum_{i=1}^{k} t_i}, \quad (4.3)$$

Чтобы найти первоначальное количество ошибок N обратимся к следующей формуле

$$\sum_{i=1}^{k} m_i \frac{\sum_{i=1}^{k} t_i}{\sum_{i=1}^{k} \frac{m_i}{N - n_{i-1}}} = \sum_{i=1}^{k} (N - n_{i-1}) t_i, \quad (4.4)$$

Из данного уравнения путем подбора найдем N = 8.

Теперь зная значение N найдем коэффициент C по формуле 4.3, и он будет равен 0,05.

Рассчитав коэффициент C получаем что интенсивность появления ошибок равна

$$\lambda = (8 - (3 + 2 + 2)) * 0.05 = 0.05$$

Выведем функцию надежности

$$R(t) = e^{-0.05t}$$

По полученным результатам ПО является надежным и может эксплуатироваться.

Так как разработанный инструмент использует платформу Kubernetes, которая из коробки предоставляет механизмы скалирования, автохилинга и т.д., что позволяет еще больше повысить программного продукта в целом.

Заключение

В ходе проведения исследования и написание выпускной квалификационной работы был проведен анализ существующих подходов к управление прикладным программным интерфейсом микросервисов, а также инструментов позволяющих эти подходы сделать эффективными и простыми.

Для достижения результата были выполнены поставленные задачи:

– проанализированы подходы к управлению прикладным программным интерфейсом в облачной среде;
– проанализированы существующие инструменты управления прикладным программным интерфейсом микросервисов;
– разработана модель инструмента с эффективным управлением прикладным программным интерфейсом;
– подтверждена эффективность предлагаемой модели на практике.

Описанный в работе инструмент управления прикладным программным интерфейсом микросервисов позволит при помощи простых конфигураций организовать публикацию прикладного программного интерфейса, а также обеспечит надежную маршрутизацию трафика внутри облачной среды.

Разработанный инструмент построен с учетом внедрения в микросервисную архитектуру с использованием современных платформ облачных технологий таких как Kubernetes, который содержит удобные встроенные инструменты балансировки нагрузки, масштабирования, а также надежного хранилища данных etcd, что позволяет достичь большой степени надежности инструмента в целом.

Доработка инструмента силами проекта позволит легко расширить возможности по управлению API, а также добавить новый функционал, такой как авторизация, консистентная балансировка, защищенные соединения TLS.

Список используемой литературы и используемых источников

1. Bill Doerrfeld Andreas Krohn, Kristopher Sandoval, Bruno Pedro The API Lifecycle: An Agile Process for Managing the Life of an API [Книга]. - [б.м.] : Nordic APIs AB (July 8, 2015), 2015.

2. Brenda Jin Saurabh Sahni, Amir Shevat Designing Web APIs: Building APIs That Developers Love [Book]. - [s.l.] : O'Reilly Media, 2018. - 1st. - 978-1492026921.

3. Brendan Burns Joe Beda, Kelsey Hightower Kubernetess Up & Running [Book] / ed. Cofer Kim. - [s.l.] : O’Reilly, 2019. - 2-e : p. 278. - 978-1-492-04653-0.

4. Bucchiarone Antonio Microservices: Science and Engineering Hardcover [Book]. - [s.l.] : Springer, 2020. - 1st. - 978-3030316457.

5. Carnell John Spring Microservices in Action [Book]. - [s.l.] : Manning Publications, 2017. - 978-1617293986.

6. Fowler Susan J. Production-Ready Microservices [Book]. - [s.l.] : O'Reilly Media, 2017. - 1st. - 978-1491965979.

7. Ian Miell Aidan Hobson Sayers Docker in Practice [Book]. - [s.l.] : Manning Publications, 2019. - 2-e : p. 384. - 978-1617294808.

8. Irakli Nadareishvili Ronnie Mitra, Matt McLarty, Mike Amundsen Microservice Architecture: Aligning Principles, Practices, and Culture [Book]. - [s.l.] : O`Reilly, 2016. - 1-e : p. 146. - 978-1491956250.

9. Jose Ramon Huerga Alex Kovalevych, Robert Buchanan, Daniel Lee, Chelsy Mooy, Xavier Bruhiere Kong: Becoming a King of API Gateways [Книга]. - [б.м.] : Bleeding Edge Press, 2018.

10. Kasun Indrasiri Prabath Siriwardena Microservices for the Enterprise [Book] / ed. Berendson Laura. - San Jose : Apress Media LLC, 2018. - 1 : p. 434. - 978-1-4842-3857-8.

11. Kavis Michael J. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS) (Wiley CIO) [Book]. - [s.l.] : Jhon Wiley & Sons, Inc, 2014. - 1-e : p. 229. - 978-1118617618.

12. Kleppmann Martin Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems Paperback [Book]. - [s.l.] : O'Reilly UK Ltd, 2017. - p. 2nd. - 978-1449373320.

13. Morgan Bruce Paulo A. Pereria Microservices in Action [Book] / ed. Stephens Michael. - NY : Manning Publications Co., 2019. - 1-e. - 9781617294457.

14. Nayyar Dr. Anand Handbook of Cloud Computing: Basic to Advance research on the concepts and design of Cloud Computing [Book]. - [s.l.] : BPB Publications, 2019. - 1-e : p. 415. - 9388176669.

15. Newman Sam Building Microservices: Designing Fine-Grained Systems [Книга]. - [б.м.] : O'Reilly Media, 2015. - 1st. - 978-1491950357.

16. Newman Sam Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith [Книга]. - [б.м.] : O'Reilly UK Ltd., 2019. - 978-1492047841.

17. Pacheco Vinicius Feitosa Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices [Book]. - [s.l.] : Packt Publishing, 2018. - 1st : p. 366. - 978-1788474030.

18. Rahul Sharma Akshay Mathur Traefik API Gateway for Microservices. With Java and Python Microservices Deployed in Kubernetes [Книга]. - [б.м.] : Springer Nature Customer Service Center LLC, 2020.

19. Richardson Chris Microservices Patterns [Book] / ed. Michaels Marina. - NY : Manning Publications Co., 2019. - 1-e : p. 522. - 9781617294549.

20. Rodger Richard The Tao of Microservices [Book]. - [s.l.] : Manning Publications, 2017. - 1st. - 978-1617293146.

21. Sayfan Gigi Hands-On Microservices with Kubernetes: Build, deploy, and manage scalable microservices on Kubernetes [Book]. - [s.l.] : Packt Publishing, 2019. - 1st. - 978-1789805468.

22. Sharma Sourabh Mastering Microservices with Java 9 - Second Edition: Build domain-driven microservice-based applications with Spring, Spring Cloud, and Angular [Book]. - [s.l.] : Packt Publishing, 2017. - 2nd. - 978-1787281448.

23. Subramanian Harihara Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs [Book]. - [s.l.] : Packt Publishing, 2019. - 1st. - 978-1788992664.

24. Vernon Vaughn Implementing Domain-Driven Design [Book]. - [s.l.] : Addison Wesley, 2013. - 978-0321834577.

25. Wolff Eberhard Microservices: A Practical Guide [Book]. - [s.l.] : CreateSpace Independent Publishing Platform, 2018. - 978-1717075901.

26. Wolff Eberhard Microservices: Flexible Software Architecture [Book]. - [s.l.] : Addison-Wesley Professional, 2016. - 1st. - 978-0134602417.

27. Джонс М. Тим Виртуализация приложений: История появления и перспективы дальнейшего развития [В Интернете] // IBM. - 2012 г.. - https://www.ibm.com/developerworks/ru/library/l-virtual-machine-architectures/.

28. О. Савельев А. Решения Microsoft для виртуализации ИТ-инфраструктуры предприятий : учебное пособие [Книга]. - Москва, Саратов : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. - 3-е : стр. 283. - 978-5-4497-0358-3.

29. Шаппелл Дэвид А. ESB - Сервисная Шина Предприятия [Книга]. - [б.м.] : БХВ-Петербург, 2008.

30. Шитько А. М. Проектирование микросервисной архитектуры программного обеспечения [Статья] // Труды БГТУ. - Минск : [б.н.], 2017 г.. - №2.

Приложение А

Схема конфигурации API

```yaml
openAPIV3Schema:
  properties:
    spec:
      properties:
        gateway:
          type: string
        routes:
          items:
            format: object
            properties:
              destination:
                properties:
                  address:
                    properties:
                      host:
                        type: string
                      port:
                        type: integer
                    type: object
                type: object
              rules:
                items:
                  format: object
                  properties:
                    match:
                      properties:
                        headers:
                          items:
                            format: object
                            properties:
                              name:
                                type: string
                              value:
                                type: string
                            type: object
                          type: array
                        path:
                          type: string
                      type: object
                    pathRewrite:
                      type: string
                  type: object
                type: array
            type: object
          type: array
      required:
      - gateway
      type: object
  type: object
```