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 Проектирование виртуальных серверов на основе технологии контейнеризации и каналов передачи данных

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

Научный руководитель канд.пед.наук, доцент, О.М. Гущина
(ученая степень, звание, И.О. Фамилия)

Тольятти, 2021

Оглавление

Введение

В настоящий момент множество компаний, занимающихся разработкой программного обеспечения обращаются к технологии виртуализации для проектирования масштабируемых, отказоустойчивых и полнофункциональных систем, которые также будут более оптимальны для компании с экономической точки зрения и иметь высокую конкурентоспособность в быстро развивающейся бизнес-модели на основе облачных сервисов, как об этом писал Лапидус Л. В [11].

Актуальность и научная значимость настоящего исследования заключается в том, что в наше время, поскольку в связи с эпидемиологической ситуацией возникает высокая потребность вести бизнес в интернете, что приводит к большому притоку пользователей, как на новые системы, так и на старые. В данной работе проводятся исследования различных технологий виртуализации и архитектуры по разработке программного обеспечения, соблюдающая критерии высокого показатели отказоустойчивости системы, производительности и гибкости развёртывания.

Главная проблема состоит в поиске необходимой технологии, способной решить ряд вопросов, возникших в связи с эпидемиологической ситуацией в мире у компаний, занимающихся ведением бизнеса в интернете. Дело в том, что по уже названным причинам, огромное количество пользователей начало производить покупки, или действия в цифровом сегменте, повлёкшие к перегрузке систем. Для решения данной проблемы различные компании начали масштабировать аппаратное обеспечение, что привело к огромным затратам средств, соответственно у большого количества компаний появилось противоречие в плане проектирования своих систем, что привело к вопросу о более эффективном применении технологий, чтобы снизить затраты.

Гипотезой исследования является то, что технология контейнерной виртуализации вместе с применением микросервисной архитектуры позволяет разрабатывать крупные веб-приложения, соответствующие критериям высокой производительности системы, отказоустойчивости и гибкости.

Объектом исследования является технология контейнерной виртуализации как базовая технология для разработки крупных облачных решений, используемая в микросервисной архитектуре.

Предмет исследования в данной работе является облачная платформа компании Netcracker, которая сочетает в себе применение микросервисной архитектуры и технологии контейнерной виртуализации. Данная платформа демонстрирует высокие показатели производительности и отказоустойчивости системы.

Целью исследования в данной работе являются технологии контейнерной виртуализации и возможности микросервисной архитектуры для разработки облачной платформы, в которой будут решены проблемы высокой производительности, гибкости развёртывания и отказоустойчивости.

Для достижения поставленной цели необходимо решить следующие задачи:

- Выполнить статистические задачи оценки существующих и перспективных требований к объекту исследования.
- Привести методологические и теоретические задачи раскрытия предмета исследования.
- Выполнить экспериментальные задачи, подтверждающие достоверность гипотезы, а также предоставить практические примеры выявления эффективности решения.

Для решения поставленных задач диссертации необходимо произвести анализ существующих технологий с указанием в сравнении с аналогами, для описания выбора конкретного инструмента для разработки. После этого

необходимо произвести проектирование схем, моделирующих работу программного обеспечения, после чего описать процесс разработки.

Научная новизна исследования состоит в том, что облачная платформа компании Netcracker способна предоставить конечному заказчику не только решение по продаже цифровых продуктов, но гибкое, видоизменяемое веб-приложение, которое может быть в короткий срок подстроено под нужды определённого сектора бизнеса, и в короткий срок выведено в эксплуатацию.

Правильное использование возможностей виртуализации, позволяет создавать и поддерживать крупные и сложные системы, наделённые высокой гибкостью, эффективностью, удобством в сопровождении и большой продуктивностью. В основном под виртуализацией подразумевается запуск нескольких операционных систем на одном компьютере, но при этом вместе с совместным использованием всех аппаратных ресурсов.

Практика разработки облачной платформы на основе технологии контейнерной виртуализации с использованием микросервисной архитектуры должна принести практическую значимость коммерческих организаций, в чьих требованиях присутствует разработка отказоустойчивой и гибкой масштабируемой системы.

На защиту выносятся:

- Исследование технологий виртуализации и способы их применения.
- Диаграммы, относящиеся к разработке облачной платформы, основанной на технологии контейнерной виртуализации и микросервисной архитектуре.
- Часть приложения облачной платформы «Netcracker» разработанное на технологии контейнерной виртуализации.

Как говорил Макс Плаф [28], виртуализация играет очень важную роль в технологии облачных вычислений. Обычно в облачных вычислениях, пользователи обмениваются данными, присутствующими в приложениях, развёрнутых на облаках. «Облако», или облачная вычислительная система, является онлайн-хранилищем, состоящая из большого количества серверов,

объединённых в одну общую сеть. Доступ к этому хранилищу осуществляется через специальные сайты или приложения. Но фактически с помощью технологии виртуализации, происходит разделение данной инфраструктуры для достижения гибкости и устойчивости системы.

Тенденции развития облачных технологий и технологий виртуализации активизировали работу по предоставлению программных решений на основе облачных вычислений у компаний, специализирующихся в области предоставления электронных услуг. Одной из таких компаний является компания Netcracker.

Netcracker Technology - дочерняя компания корпорации NEC, специализирующаяся на создании, внедрении и сопровождении систем эксплуатационной поддержки (OSS), систем поддержки бизнеса (BSS), а также SDN/NFV-решений для операторов связи, крупных предприятий и государственных учреждений.

Десятилетия решения компании Netcracker BSS и OSS, которые не являются масштабируемыми и слишком сложными для дальнейшей работы, препятствуют конкурентоспособности. В связи с проблемой существенного риска в масштабных преобразованиях системы, поставщики услуг теперь часто стремятся начать с не большой системы и масштабировать её под свои нужды. Направление облачных технологий должно стать необходимым шагом для обеспечения бизнеса и операций следующего поколения за счет повышения гибкости, безопасности, надежности и масштабируемости. Данное решение носит название облачная платформа Netcracker.

Глава 1 Описание методов разработки с использованием технологий виртуализации

В данной главе, будет проведён общий обзор технологий виртуализации. Так как именно эта базовая технология позволяет разработать облачную платформу Netcracker с наименьшими финансовыми затратами и быстрее получить результат. Данная тема особенно актуальна в наше время, поскольку в связи с эпидемиологической ситуацией возникает высокая потребность вести бизнес в интернете, что приводит к большому притоку пользователей, как на новые системы, так и на старые. Выбор технологий будет определён по следующим критериям:

- Высокая скорость работы приложения,
- Отказоустойчивость,
- Простое масштабирование системы,
- Экономичность технологий.

Для реализации облачной платформы компании Netcracker необходимо обосновать основные причины выбора определённого стека технологий, а также пояснить причины выбора технологии виртуализации конкретно для данного проекта. В конце данной главы будут проанализированы аналоги существующих систем, работающих в виртуальной среде выбранной технологии виртуализации.

1.1 Описание архитектуры проекта

Благодаря развитию облачных технологий, развертывание систем микросервисов является более продуктивным, гибким и менее затратным. Тем не менее, Циммерманн отмечает [31], что микросервисы - это деликатная тема, которую с особым интересом исследуют академические структуры, IT компании и промышленность. Впервые термин микросервисы обсуждали на конференции «Архитектура программного обеспечения» (Software Architects)

в Мае 2011 года, чтобы согласовать то, что участники считали общим архитектурным стилем. Год спустя, на той же конференции, термин «микросервисы» официально был подтвержден инженерной группой. Фактически, микросервисная архитектура была разработана как ответ на проблемы в монолитных приложениях, которые имели проблемы с масштабируемостью и гибкостью приложений [23,25].

Монолит – это архитектура построения приложения, модули которого не могут работать независимо друг от друга. Решение основанный на микросервисах, должен рассматриваться как единственный способный выполнять независимые друг от друга инструкции. [10]

Понимая, что, используя технологию микросервисов, необходимо изолировать различные участки приложения и поддержать модульную независимость, инженерное сообщество столкнулось с проблемой выбора подходящей технологии для достижения данной цели [10,17]. И решением данной проблемы стала технология виртуализации.

Технология виртуализации - это основной элемент облачных вычислений. Традиционно в облачных вычислениях используются виртуальные машины для распределения доступных ресурсов и обеспечения изолированной среды среди пользователей. Несколько виртуальных машин с собственной операционной системой и службами могут быть развернуты и запущены одновременно на одной физической машине в облачной инфраструктуре. В последнее время быстро внедряется более легкая технология виртуализации, основанная на контейнерах. Ключевое различие между виртуальными машинами и контейнерами заключается в том, что контейнеры используют одну и ту же базовую операционную систему.

Вскоре, микросервисная архитектура, с использованием контейнерной виртуализации была принята в качестве основных инструментов разработки крупных приложений, и сама концепция получила широкую поддержку интернет сообщества, что привело к потребности использования данной технологии в качестве разработки новых облачных решений, в том числе в е-

commerce сегменте. Потребность в данных решениях возникла у заказчиков вместе с тенденцией развития облачных вычислений, поэтому, компания Netcracker приняла решение разработать облачную платформу, способную предоставить заказчикам гибкую, масштабируемую и отказоустойчивую систему, используя микросервисную архитектуру и сопутствующие к ней технологии.

Платформенное решение может быть разработано до того, как будет продано конкретному заказчику. Его можно преобразовывать под нужды заказчика, вне зависимости от характера приложения и его бизнеса.

Разработка платформы подразумевает абстрактное понимание системы и её возможностей для дальнейшего преобразования в конкретную систему с небольшими дополнительными возможностями специфичные для заказчика.

Для облачной платформы важны следующие критерии:

– Производительность – один из важнейших факторов определяющий скорость работы приложения. Данный фактор основополагающий для систем, ориентированных на работу с бизнесом, так как ценится любая единица времени;

– Гибкость – критерий, который определяет, насколько приложение поддаётся изменению и масштабированию;

– Экономичность – критерий, который влияет на стоимость разработки продукта и на его поддержку.

Для заказчиков Netcracker данный продукт будет представлять облачное решение, которое имеет гибкое развитие технологии виртуализации на уровне ОС. Облачная платформа Netcracker основана на микросервисах, и может быть развернута в частных, общедоступных или гибридных облачных средах. Платформа объединяет такие возможности, как универсальный пользовательский интерфейс, широкие возможности мониторинга и отчетности и гибкие инструменты администрирования.

Данная облачная платформа может быть развернута постепенно, используя подход DevOps для ускорения выхода на рынок [6]. При этом

снижаются эксплуатационные расходы благодаря многопрофильности и гибкой масштабируемости платформы. Это создает прочную основу для построения ИТ-экосистемы.

1.2 Микросервисная архитектура

Как писал Цимерман, микросервисная архитектура - вариант сервис-ориентированной архитектуры программного обеспечения, ориентированный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов, получивший распространение в середине 2010-х годов в связи с развитием практик гибкой разработки и DevOps [6].

Если в традиционных вариантах сервис - ориентированной архитектуры модули могут быть сами по себе достаточно сложными программными системами, а взаимодействие между ними зачастую полагается на стандартизованные тяжеловесные протоколы (такие, как SOAP, XML-RPC), в микросервисной архитектуре системы выстраиваются из компонентов, выполняющих относительно элементарные функции, и взаимодействующие с использованием экономичных сетевых коммуникационных протоколов (в стиле REST с использованием, например, JSON, Protocol Buffers, Thrift). [17]

За счёт повышения гранулярности модулей архитектура нацелена на уменьшение степени зацепления и увеличение связности, что позволяет проще добавлять и изменять функции в системе в любое время.

Данная инновация может быть предложена компаниям, как альтернатива классических монолитных приложений, так как микросервисы идеально подходят для разработки кроссплатформенных приложений, которые для взаимодействия используют протоколы сетевого взаимодействия. Такой подход показан на рисунке 1.

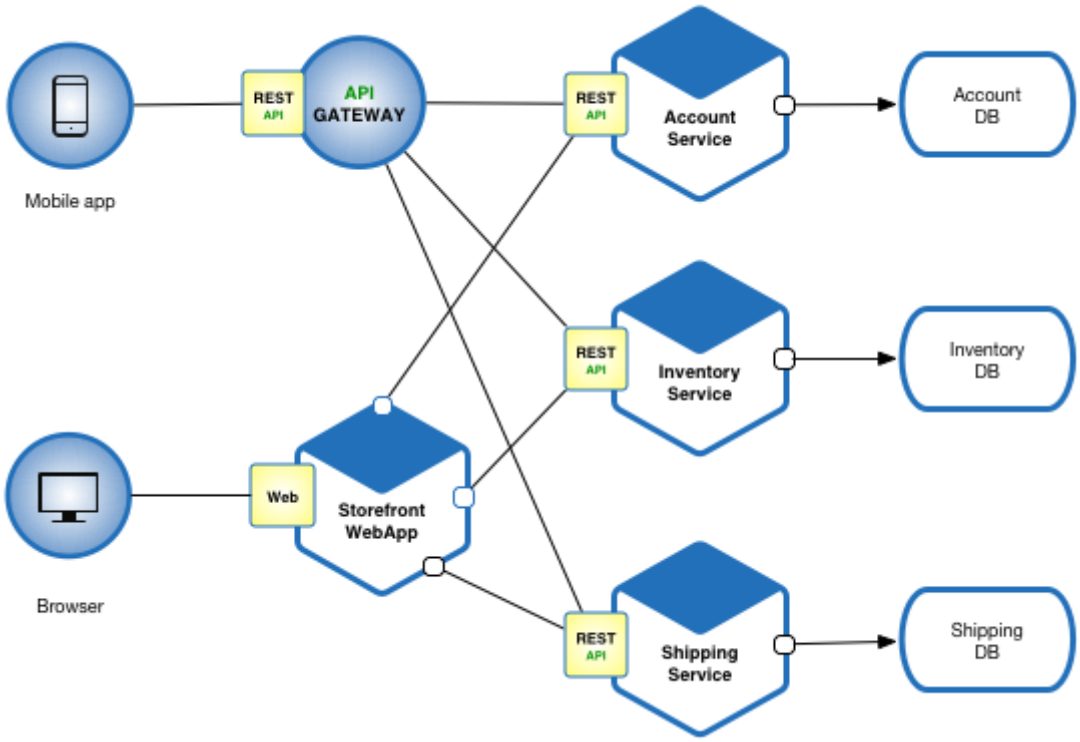

Рисунок 1 - Схема взаимодействия приложения, основанного на микросервисной архитектуре

Данная схема демонстрирует работу пользователей с приложением компании Netcracker. Приложение основано на микросервисной архитектуре и работает на разных платформах. Основная идея данной схемы, гибко настраивать систему под среду пользователя, и быть одинаковым с функциональной точки зрения для пользователя и разработчика.

Микросервисная архитектура была применена в следующих компаниях:

– Netflix имеет широко распространенную архитектуру, которая превратилась из монолитной в SOA. Он принимает более миллиарда звонков в день с более чем 800 различных типов устройств на API потокового видео. Затем каждый вызов API предлагает около пяти дополнительных вызовов бэкэнд-сервису;

– Amazon также перешел на микросервисы. Они получают бесчисленные вызовы из различных приложений, включая приложения,

которые управляют API веб-служб, а также самого веб-сайта, что было бы просто невозможно для их старой двухуровневой архитектуры;

– Аукционный сайт eBay является еще одним примером, который пережил тот же переход. Их основное приложение состоит из нескольких автономных приложений, каждое из которых выполняет бизнес-логику для различных функциональных областей.

Что касается сетевого взаимодействия, есть три компании, которые, внедряют инновации с микросервисами:

– Avi Networks предоставляет сервисы приложений, в том числе программный балансировщик нагрузки, брандмауэр, функции самообслуживания инфокоммуникационных технологий и многое другое;

– Cisco Systems признает потенциал прикладных услуг. Помимо перепродажи платформы Avi Vantage, сетевой гигант инвестировал средства в последний этап финансирования венчурного капитала Avi. Vantage интегрируется с прикладной инфраструктурой Cisco (ACI) и вместе обеспечивает решения для сетей и автоматизации Cisco для центров обработки данных Cisco. Как об этом написано в источнике [17], архитектура микросервисов позволяет автоматизировать, масштабировать и управлять платформой Vantage из централизованного местоположения. Это очень убедительно с точки зрения экономии операционных расходов, учитывая повышение эффективности и снижение зависимости от взаимодействия с человеком;

– MuleSoft помогает компаниям соединять приложения, данные и устройства с помощью общих API. Интересно в MuleSoft, что его платформа Anypoint позволяет перестраивать унаследованную проприетарную инфраструктуру в среду микросервисов, которая может быть развернута как локально, так и в общедоступном или частном облаке. Эта способность и гибкость могут стать преобразующими.

Корпоративные сети претерпевают быстрые изменения. Как об этом пишет Таненбаум [5], программно-определяемые сетевые топологии

используются во многих организациях из-за их способности упростить операции и снизить затраты. Микросервисы обеспечивают эту революцию, предоставляя услуги масштабирования, высокой доступности и оркестровки в Интернете, которые не могут сравниться с монолитным дизайном программного обеспечения. Конечно, в использовании микросервисов есть некоторые недостатки - трудности отладки дефектов и дополнительные издержки при управлении координацией между микросервисами, и это лишь некоторые из них.

1.2.1 Основные компоненты микросервиса

Способность разбивать монолитные приложения на более мелкие, независимо управляемые и обновляемые компоненты кажется идеальным подходом для организаций, занимающихся информационными технологиями, перегруженных требованиями к более быстрому продвижению.

Для правильного построения микросервисной архитектуры необходимо понимать основные компоненты микросервиса, которые понадобятся микросервису, прежде чем он сможет занять свое место в архитектуре распределенных приложений:

– модули можно легко заменить в любое время: акцент на простоту, независимость развёртывания и обновления каждого из микросервисов;

– модули организованы вокруг функций: микросервис по возможности выполняет только одну достаточно элементарную функцию;

– модули могут быть реализованы с использованием различных языков программирования, фреймворков, связующего программного обеспечения, выполняться в различных средах контейнеризации, виртуализации, под управлением различных операционных систем на различных аппаратных платформах: приоритет отдаётся в пользу

наибольшей эффективности для каждой конкретной функции, нежели стандартизации средств разработки и исполнения;

– архитектура симметричная, а не иерархическая: зависимости между микросервисами одноранговые [].

Микросервисы - это логичный ответ на недостатки громоздких монолитных приложений во время частого изменения функциональности и постоянного оттока операций. Архитектура микросервисов обеспечивает большую гибкость и производительность приложений, но имеет сложную инфраструктуру. [9,25]

1.2.2 Сравнение сервис ориентированной и микросервисной архитектуры

Сервис-ориентированная архитектура (SOA) - модульный подход к разработке программного обеспечения, основанный на использовании распределённых, слабо связанных заменяемых компонентов, оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам. [25]

Программные комплексы, разработанные в соответствии с сервис-ориентированной архитектурой, обычно реализуются как набор веб-служб, взаимодействующих по протоколу SOAP, но существуют и другие реализации (например, на базе jini, CORBA, на основе REST).

Интерфейсы компонентов в сервис-ориентированной архитектуре инкапсулируют детали реализации (операционную систему, платформу, язык программирования) от остальных компонентов, таким образом обеспечивая комбинирование и многократное использование компонентов для построения сложных распределённых программных комплексов, обеспечивая независимость от используемых платформ и инструментов разработки, способствуя масштабируемости и управляемости создаваемых систем.

В SOA есть две основные роли: поставщик услуг и потребитель услуг. Программный агент может играть обе роли. Потребительский уровень - это точка, в которой потребители (пользователи, другие сервисы или третьи

стороны) взаимодействуют с SOA, а уровень провайдеров состоит из всех сервисов, определенных в SOA. На рисунке 2 показан пример архитектуры SOA.

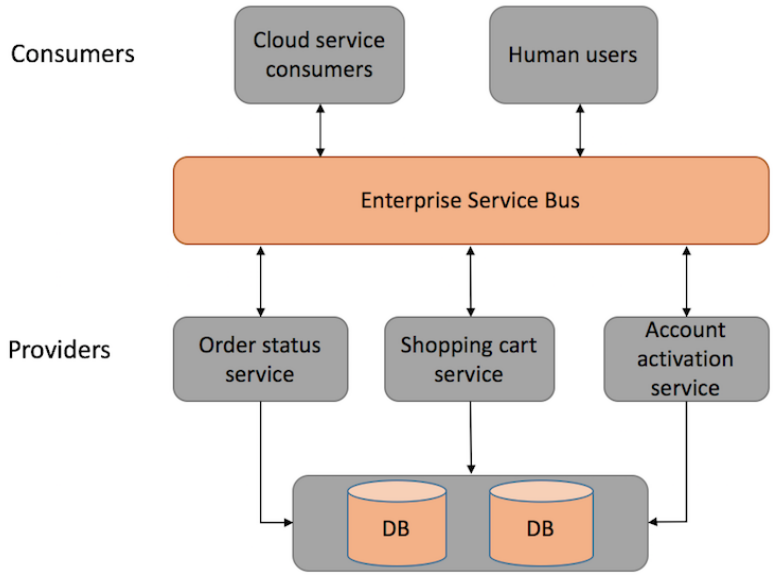

Рисунок 2 - Архитектура SOA

Enterprise Service Bus (ESB) - это стиль архитектуры интеграции, который позволяет осуществлять связь через общую коммуникационную шину, состоящую из множества соединений, точка-точка между поставщиками и потребителями. В дополнение к вышесказанному, хранилище данных совместно используется всеми сервисами в SOA.

Обе архитектуры имеют схожие плюсы и минусы и некоторые различия. В обеих архитектурах каждая служба (в отличие от монолитной архитектуры) несет определенную ответственность. Рассмотрим основные отличия этих двух архитектур, и приведём их в таблице 1.

Помимо отличий в обеих архитектурах при разработке таких систем придётся иметь дело со сложностью архитектуры и распределенной системы. В данных системах необходимо реализовать механизм межсервисной связи между микросервисами (если очередь сообщений используется в архитектурах микросервисов) или в ESB и сервисах.

Таблица 1 - Отличия сервис ориентированной архитектуры и микросервисной архитектуры

SOA	Микросервисная архитектура
Разработка сервисов может быть организована в рамках нескольких групп, однако каждая группа должна знать об общем механизме коммуникации в SOA.	В микросервисах сервисы могут работать и развертываться независимо от других сервисов, в отличие от SOA. Таким образом, легче часто развертывать новые версии служб или независимо масштабировать службу.
В SOA ESB может стать единой точкой отказа, которая влияет на все приложение. Поскольку каждый сервис связывается через ESB, если один из сервисов замедляется, это может привести к засорению ESB запросами на этот сервис.	С другой стороны, микросервисы намного лучше в отказоустойчивости. Например, если в одном микросервисе есть утечка памяти, то будет, затронут только этот микросервис. Другие микросервисы будут продолжать обрабатывать запросы.
В SOA сервисы совместно используют хранилище данных (как показано на рисунке 2). Совместное использование данных имеет свои плюсы и минусы. Например, данные могут повторно использоваться всеми службами, в то время как они обеспечивают зависимость и тесную связь между службами.	В микросервисах каждый сервис может иметь независимое хранилище данных.
SOA может быть либо монолитным, либо состоять из нескольких микросервисов.	Микросервис должен быть значительно меньше, чем обычно SOA, и, в основном, это небольшой (или) независимо развертываемый сервис.

Модульное тестирование сложнее, так как необходимо определить механизм связи в тестах. Из-за множества различных типов услуг, развертывание и эксплуатационная сложность являются проблемой для обеих архитектур. Вопросы тестирования ПО также касается и администрирования, так как, в конечном счете, запуск этих самых тестов должно происходить автоматически на стороне интеграции.

Исходя из плюсов и минусов двух архитектур, предпочтение отдаётся микросервисной архитектуре, поскольку данное решение более гибкое и отказоустойчивое, чем SSO. Всему виной ESB, которая вместе с увеличением размера системы, усложниться логика данной шины, что сильно замедлит разработку и ухудшит взаимодействия между сервисами.

Для разработки системы на базе микросервисной архитектуры необходимо владеть системой, способной запускаться в изолированной среде, способной связываться с другими подобными приложениями по сети. Для подобной системы подходят некоторые типы технологии виртуализации, описание которых приведены в следующей части проекта.

1.3 Технология виртуализации

Сама технология виртуализации существует довольно давно, и имеет несколько отличающихся между собой подходов, подразделяющихся на программные и аппаратные. Говоря общими словами, технология виртуализации эмулирует, или имитирует программным путём работы реальных физических устройств. Другими словами, виртуализация – это установка программному обеспечению выглядеть и вести себя как аппаратное, с большим преимуществом в стоимости, гибкости, общих возможностях, масштабируемости, производительности и в широком спектре приложений.

Важность технологии была признана компанией IBM вместе с развитием «мейнфремов», поскольку System/360 Model 67 виртуализировала все интерфейсы оборудования через программу Virtual Machine Monitor (VMM). В начале эпохи развития вычислительной техники операционные системы (ОС) называли «супервизором» (supervisor), но после появления возможности запуска гостевой ОС на другой ОС, был введён термин «гипервизор» (hypervisor). [3]

Технология VMM работает на основном оборудовании, позволяя запускать несколько виртуальных машин (Virtual machine), с возможностью использования разных операционных систем и в изоляции друг от друга. За период развития технологии были разработаны не мало видов технологий виртуализации, применимые в различных системах для разных типов задач, отличающихся уровнем абстракции в достижении одинаковых результатов. У

каждого вида технологий есть свои недостатки и преимущества. Самым сложным видом технологии, является эмуляция оборудования, при котором VM аппаратных средств создается на хост-системе, чтобы эмулировать интересующее оборудование.

В настоящий момент все виды технологии виртуализации имеют общую цель: оптимизировать работу системы, найти подход более эффективного использования ресурсов оборудования и достижения высокого уровня безопасности.

Основная причина, по которой множество компаний, занимающихся разработкой программного обеспечения, прибегают к виртуализации в том или ином виде, связана с большим объёмом возможностей, таких как:

- Сокращение затрат на приобретение и поддержку оборудования,
- Сокращение серверного парка,
- Сокращение штата IT-сотрудников,
- Простота в обслуживании,
- Клонирование и резервирование.

1.4 Современные технологии виртуализации

Существует множество видов виртуализации. Проведём анализ для описания основных видов современной технологии виртуализации, укажем их недостатки и преимущества. Также рассмотрим новейшие системы виртуализации и подходы к проектированию и созданию виртуальных сред.

1.4.1 Эмуляция оборудования

Эмуляция в вычислительной технике - комплекс программных, аппаратных средств или их сочетание, предназначенное для копирования (или эмулирования) функций одной вычислительной системы (гостя) на другой, отличной от первой, вычислительной системе (хосте) таким образом, чтобы эмулированное поведение как можно ближе соответствовало

поведению оригинальной системы (гостя). [22] Целью является максимально точное воспроизведение поведения в отличие от разных форм компьютерного моделирования, в которых имитируется поведение некоторой абстрактной модели. [2]

Эмулироваться могут такие устройства как: маршрутизатор, коммутатор, мобильный телефон и так далее. Схема работы данного вида виртуализации представлено на рисунке 3.

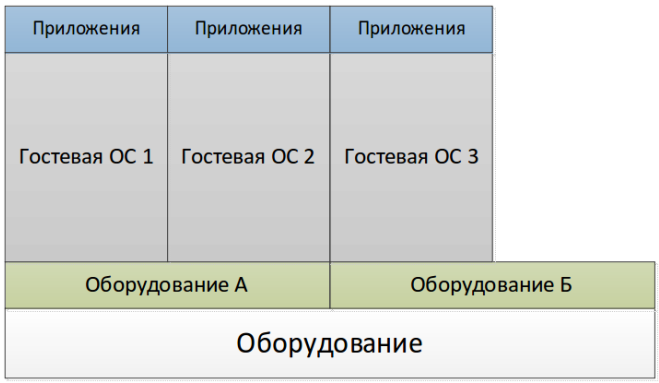

Рисунок 3 - Схема эмуляции оборудования

Главной проблемой использования эмулирования является чрезмерное потребление ресурсов основного оборудования, так как для поддержания работы эмулятора необходимо много аппаратных ресурсов основного оборудования, на котором производится эмуляция. Тем не менее, эмулирование аппаратного обеспечения имеет существенные преимущества. Например, используя эмуляцию аппаратных средств, можно управлять неизмененной операционной системой, предназначенной для Power PC на системе с ARM процессором, также можно управлять многочисленными виртуальными машинами, каждая из которых будет моделировать другой процессор. [27]

Эмуляция оборудования подойдёт для создания и тестирования программного обеспечения для разных систем, например, для системы

Android, или IOS. Также можно эмулировать работу простой сети с помощью таких эмуляторов как GNS3, или Huawei eNSP.

Тем не менее, данный вид виртуализации не подходит для приложений, которые должны иметь высокий показатель производительности, поскольку довольно много ресурсов аппаратного обеспечения будут задействованы на поддержание работы эмулятора, что приведёт к низкой скорости работы при больших нагрузках на систему. Также данный вид виртуализации сложно поддерживать и автоматизировать в рамках продуктового решения.

1.4.2 Аппаратная виртуализация

Аппаратная виртуализация - виртуализация с поддержкой специальной процессорной архитектуры. В отличие от программной виртуализации, с помощью данной техники возможно использование изолированных гостевых систем, управляемых гипервизором напрямую. [3]

Гостевая система не зависит от архитектуры хостовой платформы и реализации платформы виртуализации. Аппаратная виртуализация обеспечивает производительность, сравнимую с производительностью невиртуализованной машины, что дает виртуализации возможность практического использования и влечет её широкое распространение. Наиболее распространены технологии виртуализации Intel-VT и AMD-V.

Другими словами, на одной операционной системе можно создать несколько изолированных друг от друга виртуальных систем, под управлением собственной ОС. Гипервизор – это специальный программный менеджер виртуальных машин, который осуществляет связь между аппаратными средствами главной системы и гостевой операционной системой.

Основными преимуществами данного вида виртуализации являются:

- На главной корневой операционной системе, или сервере виртуальных машин, можно создать множество виртуальных машин, с установленными на них разными операционными системами. Нет зависимости от единого ядра ОС;

– Гостевые операционные системы выглядят и ведут себя как реальное оборудование с установленной операционной системой. На данных системах можно запускать программное обеспечение, которое будет работать также, если бы гостевая ОС являлась основной системой на аппаратном обеспечении;

– Гостевые операционные системы полностью изолированы друг от друга.

Потребность в использовании аппаратной виртуализации заставила производителей аппаратного обеспечения улучшить архитектуру процессоров, добавив поддержку предоставления прямого доступа к ресурсам процессора из гостевых систем. В свою очередь процессор с поддержкой виртуализации теперь мог работать в двух режимах:

– root operation - включено специальное программное обеспечение гипервизор, являющееся прослойкой между реальным оборудованием и гостевой ОС. Данная прослойка также носит название монитор виртуальных машин (Virtual Machine Monitor, VMM);

– non-root operation – отключена поддержка работы гипервизора.

Схема взаимодействия приложения, гостевой операционной системы и реального оборудования через гипервизор показана на рисунке 4.

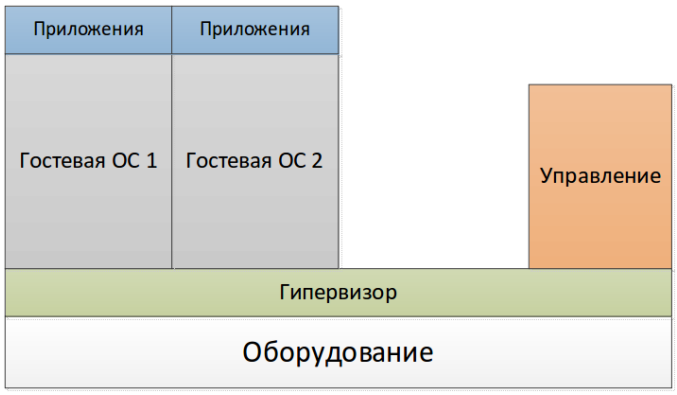

Рисунок 4 - Схема работы аппаратной виртуализации

Несмотря на все вышеперечисленные преимущества, аппаратная виртуализация имеет и некоторые недостатки:

- При обновлении гостевой операционной системы должно пройти не мало времени, прежде чем приложение на нём снова можно будет использовать. Для решения данной проблемы можно использовать перенаправление сетевых запросов на другую гостевую ОС, пока обновление на главной ОС не будет завершено;
- При большой нагрузке на гостевые ОС, частые запросы не успевают быстро обработаться гипервизором, для передачи на реальное оборудование, что несколько уменьшает производительность;
- В зависимости от выбранного производителя процессоров технология виртуализации может отличаться. Следовательно, на разных системах настройка и поддержка приложений, работающих на данной системе, может быть затруднена;
- Ресурсы основного оборудования будут израсходованы на работу гипервизора.

Подведя итог, можно сказать, что данный тип виртуализации имеет несколько весомых преимуществ, но, тем не менее, продукт работающей на данной технологии во время пиковой нагрузки на сеть будет менее производителен из-за обработки запросов гипервизором. Также ресурсы основного оборудования будут израсходованы на работу гипервизора, что приводит к покупке более дорого оборудования для поддержания системы.

1.4.3 Паравиртуализация

Паравиртуализация - техника виртуализации, при которой гостевые операционные системы подготавливаются для исполнения в виртуализированной среде, для чего их ядро незначительно модифицируется. Операционная система взаимодействует с программой гипервизора, который предоставляет ей гостевой API, вместо использования напрямую таких ресурсов, как таблица страниц памяти, код, касающийся виртуализации, локализуется непосредственно в операционную систему. Паравиртуализация

таким образом требует, чтобы гостевая операционная система была изменена для гипервизора, и это является недостатком метода, так как подобное изменение возможно лишь в случае, если гостевые ОС имеют открытые исходные коды, которые можно модифицировать согласно лицензии [18]. Но зато паравиртуализация предлагает производительность почти как у реальной не виртуализированной системы. Как и при полной виртуализации, одновременно могут поддерживаться многочисленные различные операционные системы. Метод паравиртуализации позволяет добиться более высокой производительности, чем метод динамической трансляции. [17]

Преимуществом данного вида виртуализации является:

- Более высокая производительность по отношению к аппаратной виртаулизации. Скорость работы приложения почти такая же как на реальном оборудовании;
- Требуется меньше времени на создание гостевой операционной системы, используя модифицированный образ ОС.

Недостатками же данного вида виртуализации являются следующее:

- Всё ещё используется гипервизор для поддержания связи между оборудованием и гостевыми ОС, что при высоких нагрузках также замедляет работу приложений;
- Так как данный подход подразумевает модифицирование ОС, то выбор между такими системами ограничен, так как не все системы имеют открытый исходный код;
- Как писал Бёрнс Брендан [21], модификацию ОС нужно производить аккуратно, так как это может привести к нежелательным последствиям. Например, вместо ускорения системы, можно привести к обратному эффекту, когда система начнёт работать медленнее.

Схема работы системы при работе паравиртуализации представлена на рисунке 5.

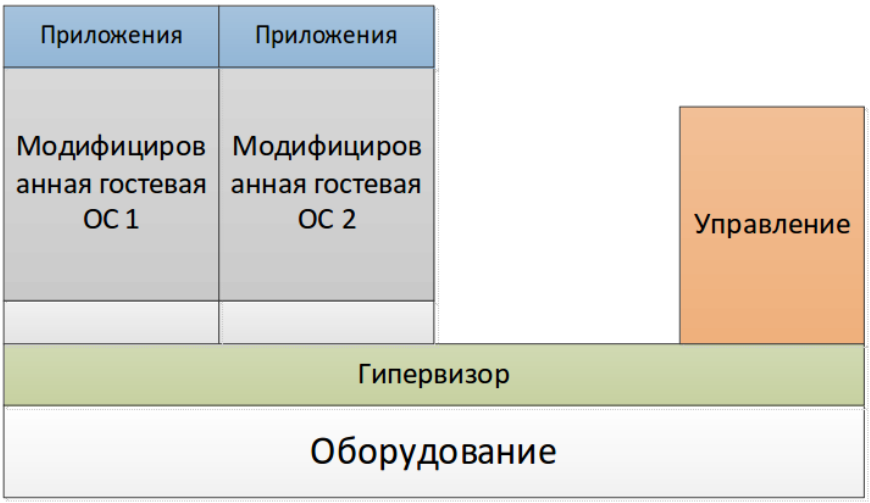

Рисунок 5 - Схема работы системы при работе паравиртуализации

Цель изменения интерфейса заключается в сокращении доли времени выполнения гостя, отведённого на выполнение операций, которые являются существенно более трудными для запуска в виртуальной среде по сравнению с не-виртуальной средой. Паравиртуализация предоставляет специально установленные обработчики прерываний, чтобы позволить гостю (гостям) и хосту принимать и опознавать эти задачи, которые иначе были бы выполнены в виртуальном домене (где производительность меньше). Таким образом, успешная паравиртуализированная платформа может позволить монитору виртуальных машин (VMM) быть проще (путём перевода выполнения критически важных задач, с виртуального домена к хосту домена) и/или уменьшить общие потери производительности машинного выполнения внутри виртуального гостя.

Широкое применение данный подход может получить при использовании в особых случаях для конкретных систем, или для проведения экспериментов над системами для создания более производительных и защищённых систем. Наиболее известные гипервизоры, использующие технологию паравиртуализации, наравне с аппаратной виртуализацией использует Xen и его ответвления (Citrix XenServer, XCP).

1.4.4 Виртуализация на уровне операционной системы

Контейнеризация (виртуализация на уровне операционной системы, контейнерная виртуализация, зонная виртуализация) - метод виртуализации, при котором ядро операционной системы поддерживает несколько изолированных экземпляров пространства пользователя вместо одного. Эти экземпляры (обычно называемые контейнерами или зонами) с точки зрения пользователя полностью идентичны отдельному экземпляру операционной системы. Для систем на базе Unix эта технология похожа на улучшенную реализацию механизма chroot. Ядро обеспечивает полную изолированность контейнеров, поэтому программы из разных контейнеров не могут воздействовать друг на друга.

В отличие от аппаратной виртуализации, при которой эмулируется аппаратное окружение и может быть запущен широкий спектр гостевых операционных систем, в контейнере может быть запущен экземпляр операционной системы только с тем же ядром, что и у хостовой операционной системы (все контейнеры узла используют общее ядро). При этом при контейнеризации отсутствуют дополнительные ресурсные накладные расходы на эмуляцию виртуального оборудования и запуск полноценного экземпляра операционной системы, характерные при аппаратной виртуализации.

Данный метод виртуализации поддерживает несколько изолированных друг от друга сущностей пространства пользователя, вместо одного. Данные сущности, именуемые контейнерами, с точки зрения конечного пользователя работают также, как и реальный сервер. Полная изолированность контейнеров друг от друга достигаются ядром операционной системы, следовательно, приложения, работающие на выделенных серверах, не могут повлиять на работу друг друга.

Так как за поддержание работы контейнеров отвечает операционная система, выделенные виртуальные сервера не ограничены одним CPU, и вправе использовать всю мощь оборудования. Данное улучшение сильно

увеличивает показатель производительности системы в сравнении с другими видами виртуализации. Также, значительно экономится память, так как контейнеры разделяют между собой общие динамические библиотеки.

Схема работы контейнерной виртуализации представлена на рисунке 6.

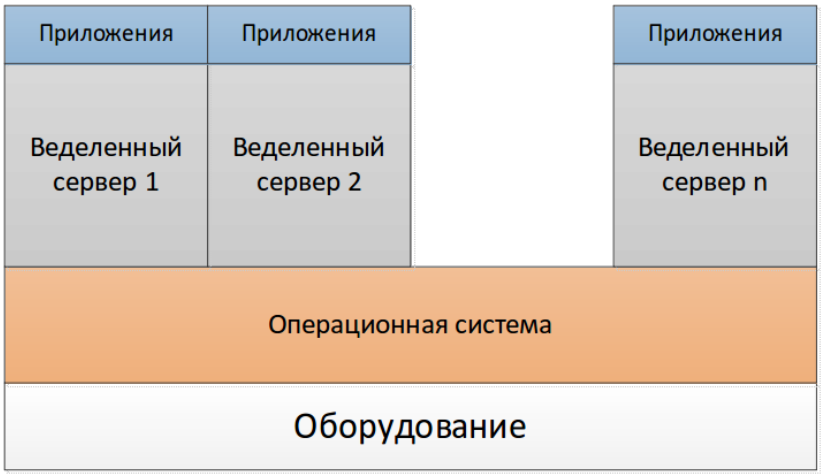

Рисунок 6 - Схема работы виртуализации на уровне операционной системы

Виртуализация на уровне операционной системы это способ, который был найден в рамках исследования сотрудниками компании Google по вопросу улучшения производительности виртуальной среды, а именно в вопросе подхода к гибкому масштабированию ресурсов в своём центре обработки данных. Данный способ позволил каждому пользователю технологии получить необходимый уровень услуги в любой момент времени, независимо от текущей нагрузки.

Эксперимент показал, что другие виды виртуализации, такие как аппаратная виртуализация и её подвиды имеют недостаточно высокий уровень производительности за счёт избыточной траты ресурсов на поддержание работы гипервизора и драйверов для аппаратной виртуализации. Также важным недостатком таких систем является низкая

скорость динамического обновления системы под изменившиеся условия для массового предоставления продукта.

Наиболее популярные решения, использовавшие данный вид виртуализации:

- Chroot,
- Docker,
- LXC.

Основными преимуществами виртуализации на уровне операционной системы являются:

- Единое исполнение на одном уровне с физическим сервером;
- Отсутствие виртуального оборудования и использование реального оборудования, что позволяет достичь высокой производительности;
- Каждый из контейнеров имеет возможность масштабироваться от уровня владения ресурсами до уровня владения целого физического сервера;
- Возможность создавать до сотни различных виртуальных выделенных серверов на одном физическом сервере;
- Так как контейнеры используют единую операционную систему, то их поддержка и обновление весьма простой процесс. Также приложения могут быть развёрнуты в другом окружении.

Другие типичные сценарии использования данного вида виртуализации включают в себя разделение нескольких приложений на отдельные контейнеры для повышения безопасности, независимости от оборудования и дополнительных функций управления ресурсами. Однако, как об этом писали Виникус Ф.М. [3], повышенная безопасность, обеспечиваемая использованием механизма chroot, далеко не такая эффективная, как может показаться. На рисунке 7 показано как контейнеры могут использоваться при применении виртуализации на уровне ОС и при применении аппаратной виртуализации.

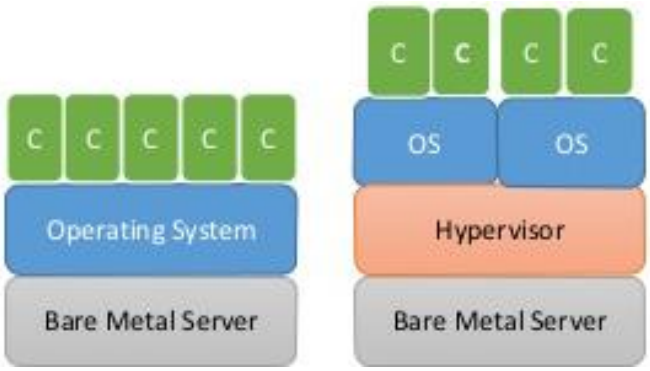

Рисунок 7 - Использование контейнеров, применяя виртуализацию на уровне ОС и аппаратную виртуализацию

Используя таблицу 2, можно провести сравнение между двумя видами виртуализации – аппаратной и виртуализации на уровне ОС.

Таблица 2 - Сравнение между аппаратной и виртуализацией на уровне ОС

Аппаратная виртуализация	Виртуализация на уровне ОС
Аппаратная виртуализация использует гипервизор, который в свою очередь эмулирует работу аппаратного, обеспечения, на котором запускаются гостевые операционные системы	Виртуализация на уровне ОС не эмулирует работу системы и за счёт этого контейнеры быстрее и более компактные, чем гипервизорные гостевые среды
Взаимосвязь между гостевой операционной системой и главной системой следует «железной» парадигме: гостевой ОС доступны только те ресурсы, которая ей предоставляет главная ОС. Ресурсы при этом должны быть сначала проанализированы главной операционной системой, после чего те же действия производят и виртуальные машины	Контейнеры обладают возможностью использовать общие ресурсы между несколькими контейнерами. Так, если несколько контейнеров обращаются к одной и той же библиотеке главной операционной системы, её идентификатор присутствует в памяти в единственном экземпляре
При использовании виртуальных машин в гипервизорных средах создаются множественные копии ядра	При работе контейнеров используется единое ядро системы, что приводит к более высокой плотности размещения контейнеров. Следовательно, скорость взаимодействия между контейнерами на одной операционной системе более высокая, чем между двумя виртуальными машинами у гипервизорных сред, а также требуется меньше ресурсов для поддержания работы данной системы

Продолжение таблицы 2

Аппаратная виртуализация	Виртуализация на уровне ОС
Виртуальные машины могут использовать любую операционную систему, отличную от главной операционной системы. Также есть доступ к настройкам к каждой из гостевых операционных систем	Контейнерная виртуализация использует подход, при котором контейнеры представляют собой изолированную копию главной операционной системы с настраиваемым пользовательским окружением и общими базовыми ресурсами системы. Следовательно, использование другой версии операционной системы невозможно, что является недостатком

Другие виды виртуализации сравнивать не имеет смысла, так как эмуляция оборудования имеет слишком низкую скорость работы, а паравиртуализацию имеет смысл использовать только в целях эксперимента.

Стоит отметить, что оба типа виртуализации повсеместно используется для создания качественных производительных систем и задачи, при которых можно использовать аппаратную виртуализацию, паравиртуализацию и виртуализацию на уровне ОС имеют место быть в разных потребительских участках бизнеса.

Благодаря проведённому сравнению аппаратной виртуализации и виртуализации на уровне операционной системы, было выяснено, что описанная архитектура проекта, лучше всего будет реализована с помощью технологии виртуализации на уровне ОС. Тем не менее, необходимо выяснить, действительно ли данная технология подходит по всем критериям требований проекта, а также необходимо решить вопрос с автоматизацией интеграции и развёртывания системы, используя контейнерную виртуализацию.

1.5 Микросервисная архитектура и виртуализация на уровне ОС

На основе виртуализации на уровне ОС можно разработать приложения на микросервисной архитектуре. Контейнеры предназначены именно для систем, основанных на этой архитектуре по ряду критериев:

- Контейнеры инкапсулируют облегченную среду выполнения для приложения, предоставляя согласованную программную среду, которая может сопровождать приложение от рабочего стола разработчика до окончательного развертывания в рабочей среде, а также можно запускать контейнеры на физических или виртуальных машинах;
- Контейнеры, выполняют изоляцию выполнения на уровне операционной системы. В данной технологии виртуализации один экземпляр операционной системы может поддерживать несколько контейнеров, каждый из которых работает в своей отдельной среде выполнения. Запуская несколько компонентов в одной операционной системе, можно сократить накладные расходы, освобождая вычислительную мощность для компонентов приложения;
- Поскольку контейнеры позволяют нескольким средам выполнения существовать в одном экземпляре операционной системы, несколько компонентов приложения могут сосуществовать в одной среде ВМ. Кроме того, в Linux можно использовать группы управления (cgroups), чтобы изолировать полную среду выполнения для определенного набора кодов приложений, гарантируя, что каждая из них имеет частную среду и, таким образом, не может влиять на работу других приложений;
- Контейнеры обеспечивают более тонкую среду выполнения и позволяют изолировать приложения. В целом, контейнеры запускаются в считанные секунды или даже миллисекунды в некоторых случаях. Это намного быстрее, чем виртуальные машины. Вот почему, с точки зрения производительности, контейнеры являются гораздо лучшей основой исполнения для архитектур микросервисов. Их быстрое создание гораздо лучше соответствует неустойчивым характеристикам рабочей нагрузки, связанным с микросервисами. Исходя из перечисленных выше пунктов, более тонкие среды выполнения контейнеров и возможность размещения совместно размещенных компонентов приложений в одном экземпляре

операционной системы помогут добиться более высоких показателей использования сервера.

С точки зрения технологий, приложения, разработанные на микросервисной архитектуре, используют не только виртуализацию на базе ОС, но и другие технологии. С момента первого упоминания о микросервисах появлялись разные инструменты для расширения возможностей использования сервисов и обслуживая большую пользовательскую базу. Описание появления подобных инструментов приведено в таблице А.1. Хронология появления данных инструментов определена в таблице сверху вниз. В данной таблице указаны не все инструменты, которые могут использоваться в микросервисной архитектуре, но это также показывает, насколько в данной архитектуре, преобладает модульность и открытость в разработке новых подходов и решений.

Основанные на контейнерах приложения микросервисов в производственных средах могут лучше реагировать на неустойчивые рабочие нагрузки. По мере того, как компании начинают все больше переходить от ведения бизнеса к цифровым предложениям, сокращение времени запуска контейнеров может помочь повысить удовлетворенность пользователей и улучшить финансовые показатели приносящих доход приложений.

Приложение, работающее на микросервисной архитектуре при использовании контейнеров, где применяется оркестровка контейнеров, показано на рисунке 8. Данная диаграмма приведена с целью ознакомления простой системы контейнерной взаимосвязи, поскольку для облачной платформы компании Netcracker будет необходимо использовать несколько контейнеров в цепочке.

Далее, так как в облачной платформе Netcracker будут использоваться облачные технологии, службы могут иметь динамическое расположение в сети из-за перезапуска, сбоя и масштабирования. Ведение файла конфигурации вручную просто невозможно. Сервисы обычно должны общаться друг с другом. Так, например, в монолитном приложении сервисы

вызывают друг друга посредством вызовов методов или процедур на уровне языка. В традиционном распределенном развертывании системы службы работают в фиксированных, хорошо известных местоположениях (хостах и портах) и поэтому могут легко вызывать друг друга, используя HTTP / REST или какой-либо механизм RPC.

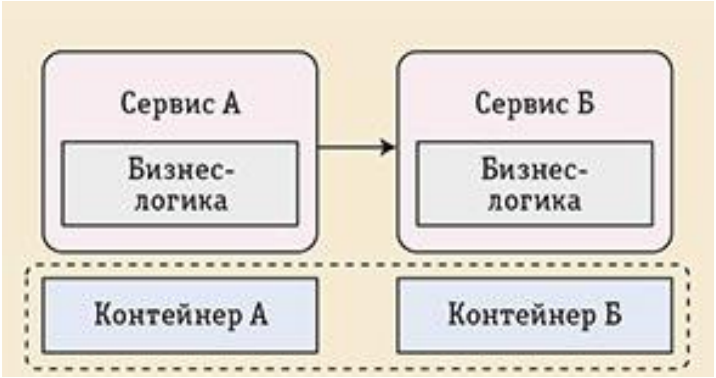

Рисунок 8 - Оркестровка контейнеров

Однако современное приложение, основанное на микросервисах, работает в виртуализированных или контейнерных средах, где количество экземпляров службы и их расположение динамически изменяются [1]. Следовательно, необходимо реализовать механизм, позволяющий клиентам сервиса отправлять запросы динамически меняющемуся набору эфемерных экземпляров службы. Такой пример сервисного взаимодействия в платформе компании Netcracker является основным, поскольку на базе облачной платформы создаются уже более специализированные проекты, изменяя исполняемую среду под свои нужды и добавляя свой функционал в существующую базу. Схема взаимодействия подобной системы показана на рисунке 9.

Далее, рассмотрим сценарий, при котором стоит задача вызывать несколько нисходящих сервисов и представить эту функциональность в качестве другого (составного) сервиса. Когда необходимо реализовать тот же сценарий с использованием микросервисов, больше не будет

централизованного уровня интеграции, а есть только набор (составных и атомарных) микросервисов.

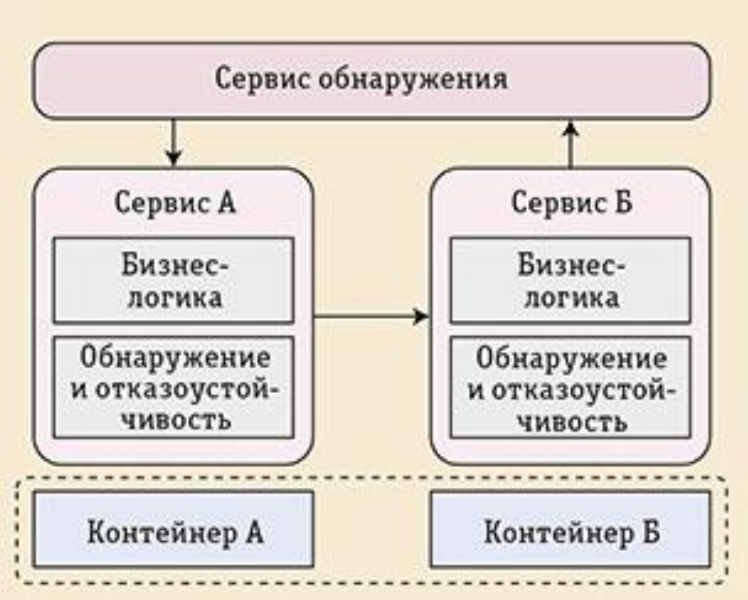

Рисунок 9 - Схема взаимодействия контейнеров, используя сервис обнаружения

Таким образом, необходимо реализовать все эти функции на уровне микросервисов. Поэтому данный микросервис, который взаимодействует с другими сервисами, включает в себя:

- Бизнес-логика, которая реализует бизнес-функции, вычисления и логику составления / интеграции услуг;
- Сетевые функции, которые заботятся о механизмах межсервисной связи (базовый вызов службы по заданному протоколу, применение шаблонов устойчивости и устойчивости, обнаружение службы и т. д.) Эти сетевые функции построены поверх базового сетевого стека уровня ОС.

Микросервисы могут быть развернуты различными способами; они могут быть частью серверной архитектуры, размещенной в контейнерах, разработанной с использованием PaaS, или, теоретически, используемой для создания локально размещенного приложения.

Схема взаимодействия контейнеров представлена на рисунке 10.

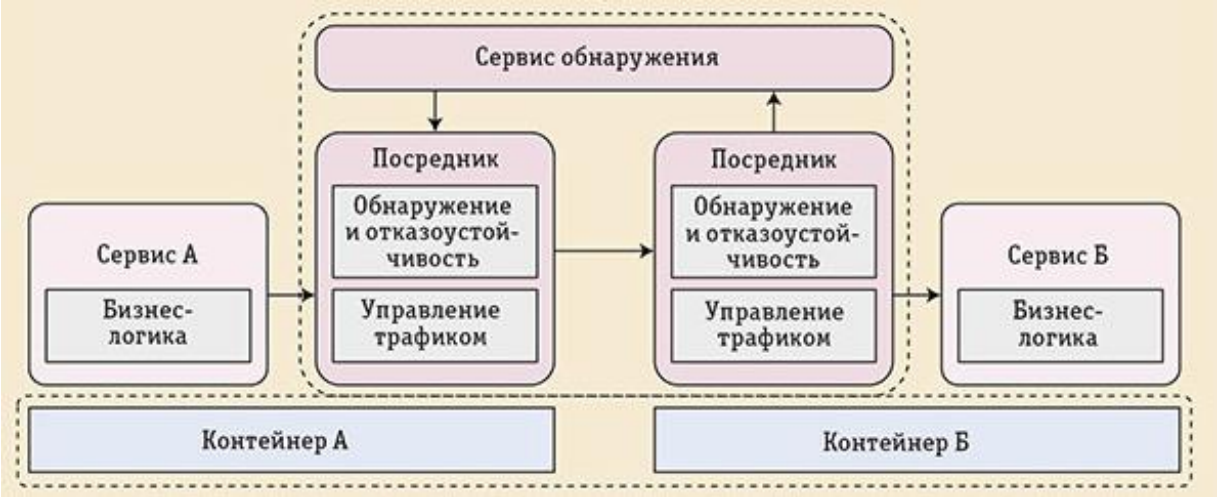

Рисунок 10 - Схема взаимодействия контейнеров при использовании сервисной mesh сети

Выводы

В данной главе был выполнен системный анализ проекта; исследована архитектура микросервисов и технология виртуализации для разработки системы на основе микросервисной архитектуры – ей стала технология виртуализации на уровне ОС. В рамках исследования выяснено, что контейнеризация подходит для разработки систем, основанных на микросервисах. Приведены примеры абстрактных проектов, в которых вкратце раскрывается принцип взаимодействия контейнеров между собой.

В следующей главе будет приведён пример работы системы на базе технологии контейнеризации, а также процесс от начала разработки до начала эксплуатации продукта потенциальным заказчиком.

Глава 2 Проектирование схем взаимодействия элементов платформы с использованием технологии контейнерной виртуализации.

В данной главе будет рассмотрена разработка прототипа продукта, основанного на технологии контейнеризации для ООО «Нэткрекер». Для выполнения данной задачи, будет использован ряд инструментов и технологий, такие как, например:

- Контейнерный движок,
- Обнаружение сервисов,
- Оркестровка контейнеров.

По каждым инструментам и технологиям, которые будут использоваться при разработке облачной платформы, будет пояснено, почему выбор упал именно на эти технологии.

Затем прототип платформы будет двигаться к понятию микросервисной архитектуры, описание которой, также приведено в главе, посвящённой исследованию технологий необходимых для разработки, но более подробный анализ будет приведён в данной главе, с практической точки зрения. Так, будут должны учитываться следующие рекомендации при разработке продукта микросервисной архитектуры:

- Каждый портал, который демонстрирует веб-интерфейс, должен располагаться на отдельном домене;
- Каждый микросервис, который будет разработан, должен концентрироваться только на одном сервисе приложения;
- Каждый микросервис должен быть развернут отдельно, и быть независимым от других микросервисов;
- Каждый сервис может быть преобразован в более мелкие сервисы.

В данной главе также будут продемонстрированы возможности микросервисной архитектуры и программного обеспечения, позволяющие

контролировать контейнерную среду и взаимодействие микросервисов между собой. Для выполнения данной задачи будут настроены процессы оркестровки контейнеров, мониторинга и тестирования среды, а также приведён план развития облачной платформы.

2.1 Облачная платформа компании Netcracker

Облачная платформа компании Netcracker берет на себя ответственность «Динамических дата центров». Цель, создания данной платформы: гибкость ресурсов, которые могут быть предоставлены, заменены, обновлены, или удалены почти в реальном времени. Данные ресурсы, или их также называют сервисами, можно копировать и устанавливать весь архитектурный стек за несколько часов из одного региона в другой, используя CloudFormation, который устанавливает регуляцию построения приложения из разного сочетания блоков сервисов для различных регионов.

На любой платформе облачного хостинга ресурсы должны будут распределены между многими арендаторами, сохраняя безопасность приложения, как об этом пишет Богнер и Циммерман [20], и что является важным требованием заказчика. Это означает, что ресурсы, такие как: ЦП, память, дисковое хранилище, которые размещены на общей физической архитектуре, будут совместно использоваться в многопользовательской модели.

2.1.1 Разделение приложения на микросервисы

Для понимания, какие именно микросервисы необходимы для облачной платформы компании Netcracker, был проведён анализ со стороной заказчика, где главной целью анализа, являлось сбор критериев и согласование возможностей платформы и потребностей заказчика. Были выделены следующие задачи:

- Платформа должна предоставлять возможность покупки продуктов облачных сервисов с возможностью подписок и портал для цифровых сервисов следующего поколения, облачных приложений и виртуализированных сервисов. Данный портал будет основным для взаимодействия конечного пользователя, и должен будет называться Shopping Portal, или интернет магазин;
- Для редактирования, удаления и добавления новых продуктов, а также все взаимодействия с пользователем и кастомизацией интернет-магазина, должен заниматься Admin Portal, или портал для администрирования интернет-магазина.

Данные задачи должны быть реализованы, используя микросервисную архитектуру, в качестве базовой архитектуры. Так как данные порталы, являются частью одной экосистемы, необходимо выделить общие сервисы, которые понадобятся при работе, как интернет-магазина, так и порталом для его администрирования, и как пользователь будет взаимодействовать с данной системой.

Виртуальные сервера в данном случае должны будут обеспечить быструю и безопасную работу с порталами. Добиться этого можно, разделив приложение, на несколько независимых друг от друга сервисов, которые имеют открытые для вызова функции. Таким образом, разделив приложение, на несколько частей, можно редактировать и обновлять приложение гораздо быстрее и без особых сложностей.

Рассмотрим модель взаимодействия пользователя интернет-магазина и динамического обновления списка продуктов после внесения изменений в портал администрирования. Оба этих портала являются уже существующими микросервисами, которые будут представлять отдельный контейнер и свой собственный виртуальный сервер. Данная модель упрощена, для демонстрации взаимодействия порталов между собой и пояснения о выделении общей логики в отдельный контейнер, и графически показана на рисунке 11.

Данная модель сконцентрирована вокруг взаимодействия двух порталов между собой только на динамическом изменении списка продуктов. В случае если на портале администрирования интернет-магазином, произойдёт изменение списка продуктов, при переходе на интернет-магазин, обычный пользователь должен увидеть эти изменения.

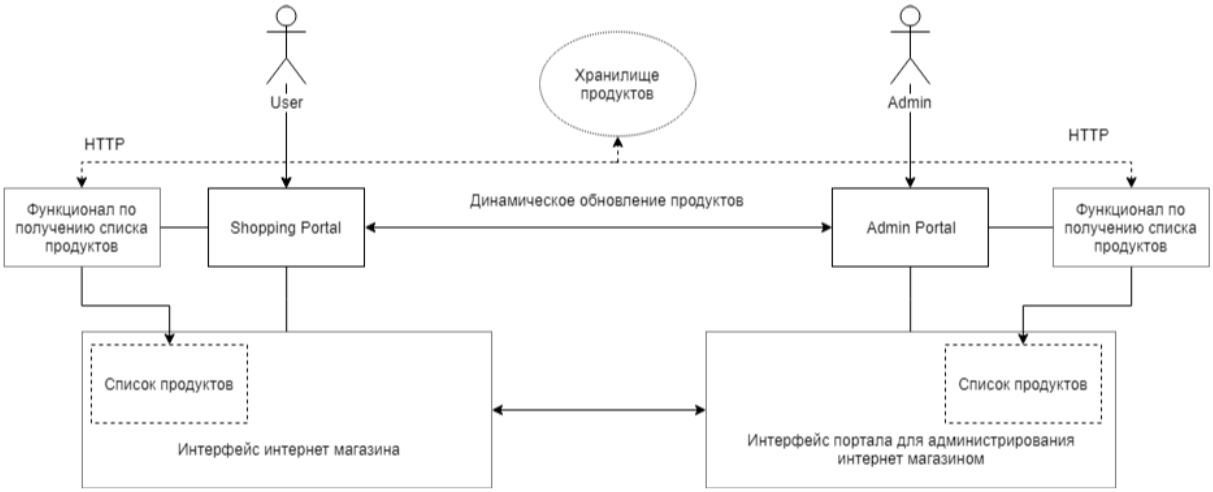

Рисунок 11 - Клиентская модель взаимодействия пользователя с интернет-магазином и динамическое обновление списка продуктов.

Как показано на рисунке 11, и интернет-магазин и портал администрирования должны отобразить список продуктов и для этого потребуется реализовать функцию получения из базы данных продуктов на стороне одного из порталов, что вызывает сложности, поскольку данный функционал должен быть открыт для использования в других порталах. Возможности микросервисов и контейнеризации помогут отделить данный функционал от конкретного портала. Для этого вся логика должна быть вынесена в отдельный микросервис, который в свою очередь является контейнером, со своими собственными переменными окружения и может обладать своей базой данных, или использовать базу данных других микросервисов.

Таким образом, достигается отделение логики по работе с продуктами в общий сервис, демонстрация которого представлена на рисунке 12.

Добавленный микросервис будет называться Product Storage, и под его зону ответственности попадают следующие функциональные задачи:

- Хранение продуктов в базе данных и функционал изменения, добавления, удаления продуктов из списка;
- Предоставление открытых ссылок для взаимодействия других микросервисов с продуктами через протокол http;
- Универсальность функционала, которая подойдёт для разных порталов.

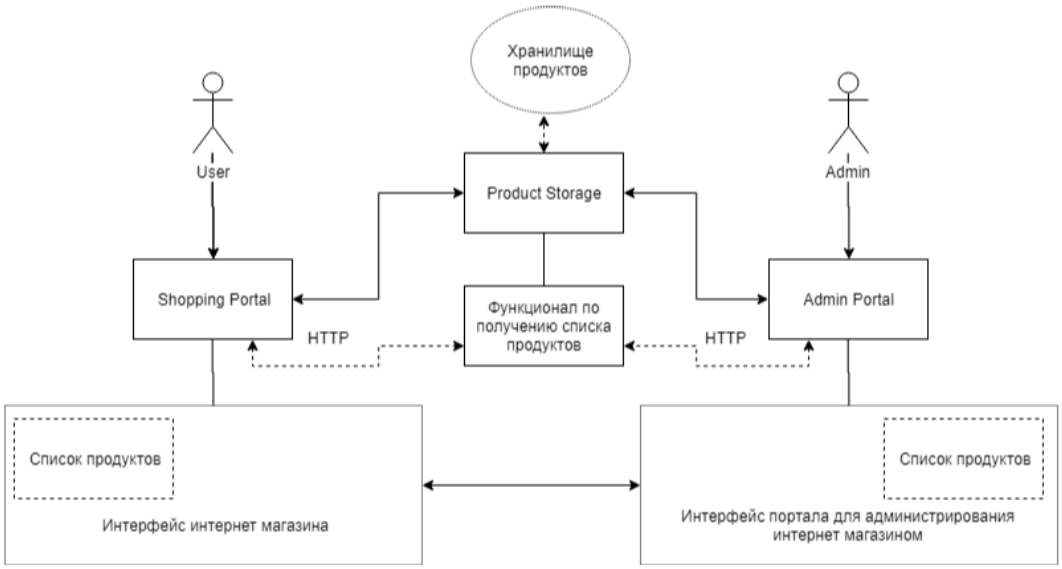

Рисунок 12 - Выделение функционала для получения списка продуктов в отдельный сервис

Важно также учесть, что разработкой и поддержкой будет заниматься отдельная команда, что приводит к тому, что в случае возникновения ошибок при работе с продуктами, или при возникновении задачи добавления нового функционала, задача будет завершена раньше и экспертиза будет более тщательной.

Технология контейнеризации в данном случае играет прямую роль, поскольку для проверки функционала, необходимо обновление версий образа контейнера.

На рисунке 13 отражён реальный опыт работы с контейнерами и представляет начальную схему по разработке облачной платформы компании Netcracker. Пока данная система не выросла в объёме и не стала слишком большой для запуска на локальном сервере, её можно запустить и разрабатывать, используя средне производительные системы. Далее, данная схема будет расти по мере необходимости новых сервисов, и для большего количества контейнеров, понадобится оркестратор.

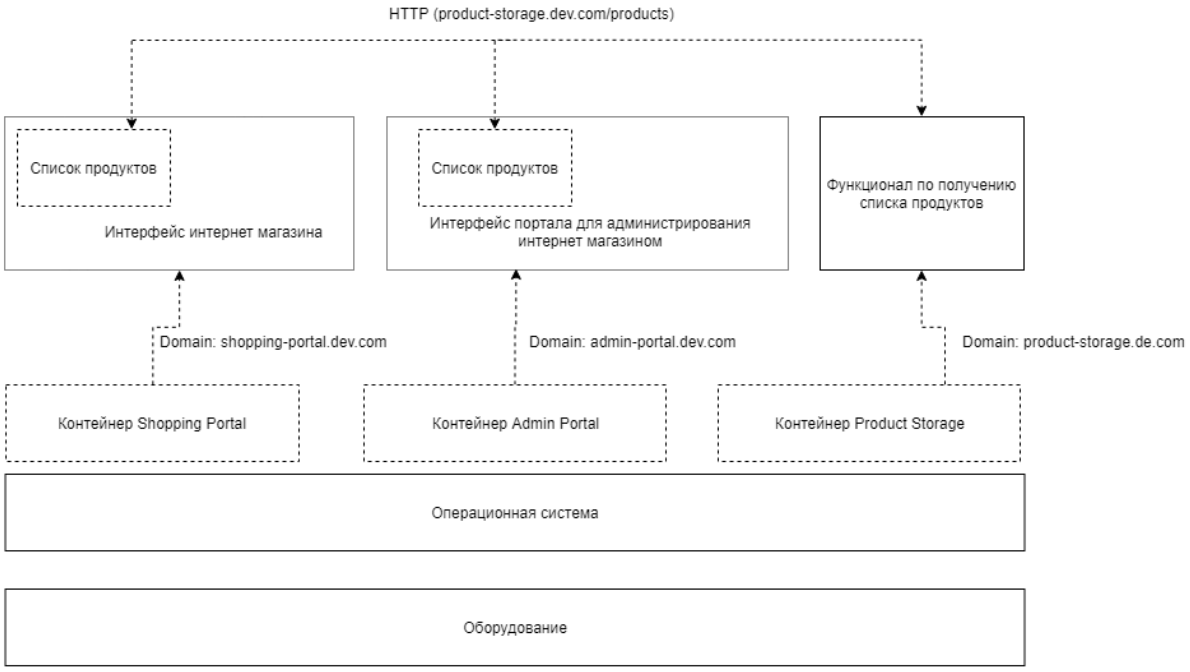

Рисунок 13 - Логическая схема контейнеров и установленных на них приложений

Ключевые моменты данной схемы:

- Для каждого приложения выделен свой контейнер, на котором запущено приложение, которое может иметь несколько сущностей, для

отказоустойчивости, а также каждый контейнер независим друг от друга и управляется контейнерным движком;

- Каждый контейнер имеет свой домен, что соответствует концепции доменного управления для проектирования микросервисов. Ограниченный контекст явно определяет границы модели;
- Взаимодействие между приложениями и сервисами происходит по протоколу HTTP.

В современном мире API обычно разрабатываются с использованием стиля RESTful. Эти API будут иметь серию глаголов, связанных с действиями HTTP, например:

- GET (получить одну сущность или коллекцию),
- POST (добавить элемент в коллекцию),
- PUT (редактировать элемент, который уже существует в коллекции),
- DELETE (удалить элемент в коллекции).

Преимущество такой согласованности в разных приложениях заключается в наличии стандарта при выполнении различных действий. Четыре вышеупомянутых HTTP-глагола коррелируют с общими возможностями CRUD, которые сегодня используют многие приложения. При работе с различными API-интерфейсами в одном приложении это обеспечивает узнаваемый способ понять последствия действий, выполняемых на разных интерфейсах.

2.1.2 Сервисы авторизации и механизм обнаружения служб

Для построения системы облачной платформы Netcracker, необходимо так построить архитектуру проекта, чтобы микросервисы взаимодействовали между собой и были небольшими частями приложения, вместе образующие устойчивую экосистему. Рассмотрим часть сервисов, составляющие большую часть облачной платформы Netcracker и их взаимодействие, начиная с сервиса авторизации.

Для реализации авторизации, используя микросервисную архитектуру, можно применить использование сеанса на стороне сервиса для сохранения состояния пользователя. Поскольку сервис имеет возможность сохранять состояние авторизации, это влияет на горизонтальное расширение сервиса. Рекомендуется использовать токен для записи статуса входа пользователя в архитектуре микросервиса.

Токен используется для указания личности пользователя. Поэтому содержимое токена должно быть зашифровано, чтобы избежать фальсификации со стороны запрашивающей стороны или третьей стороны. JWT (Json Web Token) - это открытый стандарт (RFC 7519), который определяет формат токена, определяет содержимое токена, шифрует его и предоставляет lib для различных языков. [19]

Используя токен для аутентификации пользователя, сервис не сохраняет статус пользователя. Клиент должен отправлять токен на сервер для аутентификации каждый раз, когда клиент запрашивает его. Основной процесс аутентификации пользователя в режиме токена графически показан на рисунке 14.

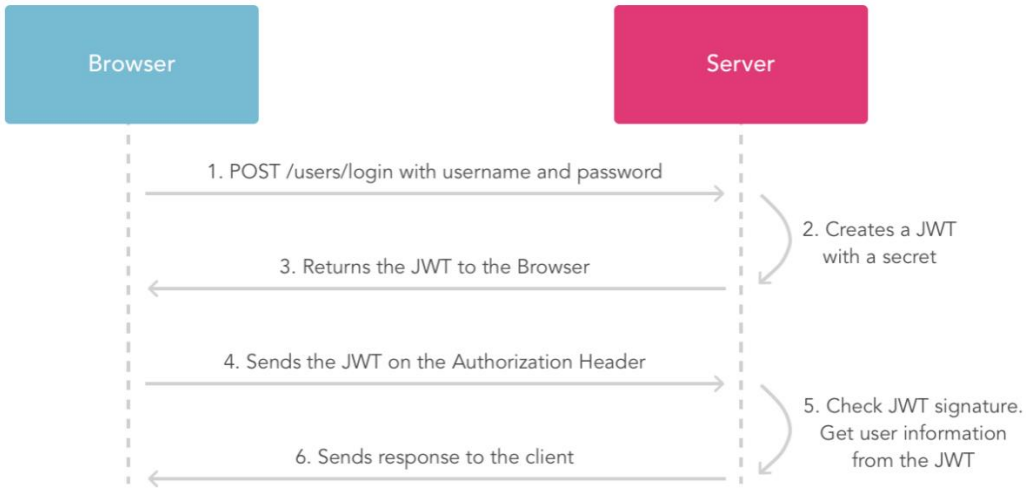

Рисунок 14 - Процесс аутентификации пользователя в режиме токена

Процесс аутентификации пользователя аналогичен базовому процессу аутентификации токена. Разница в том, что шлюз API добавляется как вход внешнего запроса. Этот сценарий означает, что все запросы проходят через шлюз API, эффективно скрывая микросервисы.

Рассмотрим сервис API gateway, который выполняет крайне важную функцию, а именно то, как клиенты приложений на основе микросервисов могут получить доступ к другим сервисам, так как для микросервисной архитектуре применимы следующие принципы:

- производительность сети различна для разных типов клиентов. Например, мобильная сеть обычно намного медленнее и имеет гораздо большую задержку, чем немобильная сеть. И, конечно, любая WAN намного медленнее, чем LAN. Это означает, что собственный мобильный клиент использует сеть с очень разными характеристиками производительности, чем локальная сеть, используемая веб-приложением на стороне сервера. Серверное веб-приложение может отправлять несколько запросов к внутренним сервисам, не влияя на пользовательский интерфейс, в то время как мобильный клиент может сделать только несколько;
- количество экземпляров сервисов и их расположение (хост + порт) изменяется динамически. Например, сервис с адресом product-storage.development.com:8080 может быть перенесён на другой домен product-storage.prod.com:4040;
- разделение на сервисы может меняться со временем и должно быть скрыто от клиентов;
- сервисы могут использовать разнообразный набор протоколов, некоторые из которых могут быть не дружественными к сети.

Решение создания сервиса API gateway трактуется этими критериями и реализуется следующим образом: API-шлюз, являющийся единой точкой входа для всех клиентов, обрабатывает запросы. Некоторые запросы просто проксируются / направляются в соответствующий сервис. Он обрабатывает другие запросы, разветвляясь на несколько сервисов. Вместо того чтобы

предоставлять единый для всех стиль API, шлюз API может предоставлять разные API для каждого клиента. Например, шлюз Netflix API запускает специфичный для клиента код адаптера, который предоставляет каждому клиенту API, который лучше всего соответствует его требованиям.

Как пишет об этом Лиз Райс [12], шлюз API также может обеспечивать безопасность, например, проверять, авторизован ли клиент для выполнения запроса. И возвращаясь к сервису авторизации, по запросу шлюз API преобразует исходный токен пользователя в непрозрачный токен, который может разрешить только сам, как показано на рисунке А.2.

В этом случае выход из системы не является проблемой, поскольку шлюз API может отозвать токен пользователя при выходе из системы, а также добавляет дополнительную защиту токена аутентификации от дешифрования, скрывая его от клиента.

С точки зрения производительности системы в сравнении с монолитной системой, данный подход выигрывает, поскольку сервисы авторизации работают через механизм обнаружения сервисов API Gateway, что с точки зрения архитектуры приложения правильно и безопасно [13]. Для API Gateway с помощью технологии контейнеризации можно создать несколько сущностей данного сервиса, для увеличения пропускной способности системы, так как большинство запросов будут обращаться именно к этому сервису, по адресу gateway.development.com/.

2.1.3 Сервис кастомизации веб-приложения и сервис кеширования статических данных

Как обсуждалось ранее, веб-порталы должны иметь возможность менять свой внешний вид и переключение между разными темами веб порталов, не должно занимать слишком большое количество времени. Как выяснилось, для таких целей целесообразно также использовать преимущества микросервисной архитектуры и вынести логику по кастомизации приложения в отдельный сервис, названный customization-manager. Данный сервис, как и другие микросервисы, будет обладать

открытым API для кастомизации приложения, а обращаться к нему из других сервис можно используя, ранее описанный сервис gateway API.

Механизм кастомизирования приложения определяется логикой самого сервиса customization-manager, который в свою очередь выносится в открытый доступ для других микросервисов. Данная функция принимает на вход текстовый файл, который имеет расширение .css, и заменяет по специальным ключам значения, которые определены в базе самого customization-manager.

На рисунке 16 показано, как файл стилей, запрашивается интернет-магазином у сервиса customization-manager.

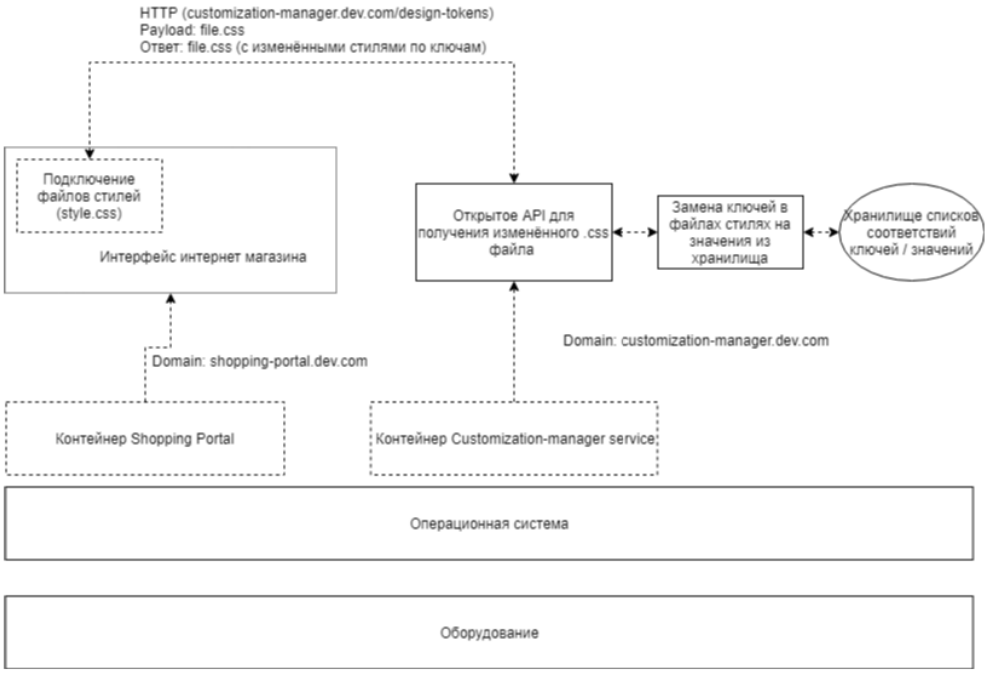

Рисунок 16 - Демонстрация получения стилей веб-портала от сервиса customization-manager

Данный http запрос имеет поле payload, это данные, которые обычно отправляются с помощью запроса POST или PUT, и в данном случае отправляется файл file.css, который в процессе обрабатывается сервисом кастомизации. После чего возвращается обновлённый файл, с заменёнными

ключами, которые необходимо было указать в файле на стороне интернет-магазина. Процесс замены ключей, и то, как выглядит файл до и после обработки сервисом кастомизации показан на рисунке Б.1.

Таким образом, каждый раз, когда интернет-магазину, или порталу администрирования интернет-магазином потребуется загрузить страницу, необходимо будет каждый раз запрашивать стили у customization-manager, что сильно повлияет на нагрузку сети и производительность системы в целом. Для решения данной проблемы можно использовать сервис кеширования статических данных.

Кэш - это память с большей скоростью доступа, предназначенная для ускорения обращения к данным, содержащимся постоянно в памяти с меньшей скоростью доступа (далее «основная память»). Кэширование применяется ЦПУ, жёсткими дисками, браузерами, веб-серверами, службами DNS и WINS.

Кэш состоит из набора записей. Каждая запись ассоциирована с элементом данных или блоком данных (небольшой части данных), которая является копией элемента данных в основной памяти. Каждая запись имеет идентификатор, часто называемый тегом, определяющий соответствие между элементами данных в кэше и их копиями в основной памяти.

Сервис кеширования данных важен, так как в платформе множество случаев использования таких данных в своих нуждах, например:

- Java Script файлы,
- HTML файлы,
- CSS файлы.

Для выполнения функции кеширования данный сервис будет иметь собственную базу данных, где будут храниться все кэшируемые файлы, и отданы по необходимости клиентам по имени микросервиса. Таким образом, сохранится уникальность переданных данных и специфика определённого портала на кеширование определённых файлов. Время хранения в кеш в данный момент будет составлять 10 минут.

В процессе запроса за кешируемыми данными, сервис будет проверять, есть ли данные в базе, и если нет, то автоматически сделает запрос за данными к определённым сервисам, используя всё тот же сервис API Gateway. Демонстрация получения кешируемых данных, на примере получения файлов стилей, показано на рисунке 17.

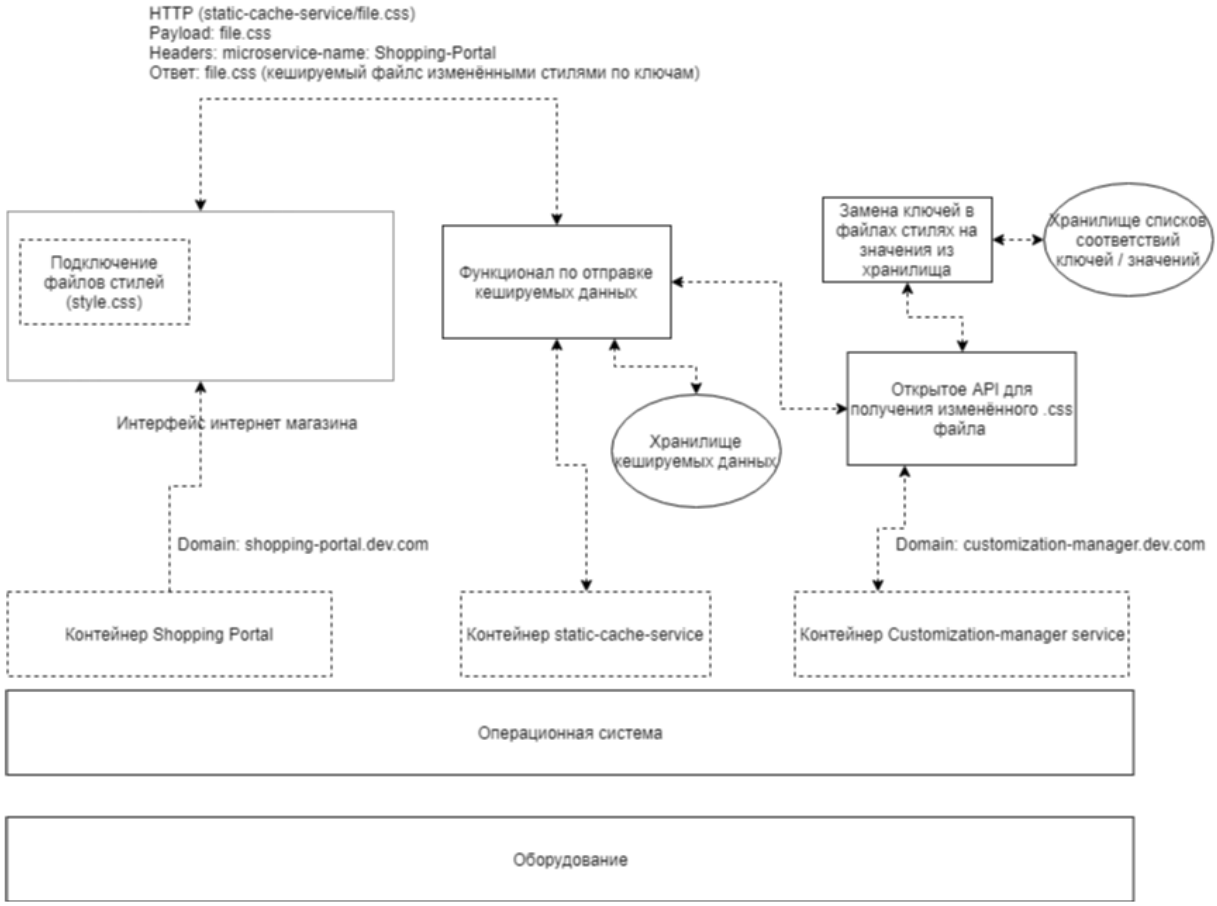

Рисунок 17 - Демонстрация получения стилей веб-портала от сервиса static-cache-service

Как показано на рисунке 17, при запросе за файлом по адресу static-cache-service/file.css, внутри имеется поле Headers, которое в протоколе HTTP используется для передачи метаданных текстового вида, для обработки их на стороне сервера. В случае если сервис кеширования не имеет в хранилище нужного файл, то самостоятельно делает запрос на customization-manager,

после чего кеширует файл в своём хранилище на 10 минут и отправляет ответ клиенту.

2.2 Модель взаимодействия микросервисов в облачной платформе, использующей каналы передачи данных компании Netcracker

В монолитном приложении компоненты вызывают друг друга посредством вызовов методов или функций на уровне языка. Напротив, приложение на основе микросервисов - это распределенная система, работающая на нескольких виртуальных машинах. Каждый экземпляр сервиса обычно является процессом. Следовательно, сервисы должны взаимодействовать с использованием механизма межпроцессного взаимодействия (IPC).

Существует множество стилей взаимодействия клиент-сервис. Их можно разделить на две категории. Первое измерение - является ли взаимодействие один-к-одному или один-ко-многим:

- один-к-одному - каждый клиентский запрос обрабатывается ровно одним экземпляром службы;
- один-ко-многим - каждый запрос обрабатывается несколькими экземплярами службы.

Второе измерение - является ли взаимодействие синхронным или асинхронным:

- синхронный - клиент ожидает своевременного ответа от службы и может даже заблокировать, пока он ждет;
- асинхронный - клиент не блокируется во время ожидания ответа, и ответ, если таковой имеется, не обязательно отправляется немедленно.

Существуют следующие виды взаимодействия один-к-одному:

- запрос / ответ - клиент отправляет запрос в сервис и ожидает ответа. Клиент ожидает, что ответ прибудет своевременно. В приложении на основе потоков поток, который выполняет запрос, может даже блокироваться во время ожидания;

- уведомление (так называемый односторонний запрос) - клиент отправляет запрос в сервис, но ответ не ожидается или не отправлен;
- запрос / асинхронный ответ - клиент отправляет запрос сервису, который отвечает асинхронно. Клиент не блокируется во время ожидания и рассчитан на то, что ответ может не прийти некоторое время.

Существуют следующие виды взаимодействия один-ко-многим:

- опубликовать / подписаться - клиент публикует уведомление, которое используется нулевым или большим количеством заинтересованных сервисов;
- публикация / асинхронные ответы - клиент публикует сообщение с запросом, и затем некоторое время ожидает ответов от заинтересованных сервисов.

Каждый сервис обычно использует комбинацию этих стилей взаимодействия. Для некоторых сервисов достаточно одного механизма IPC. Другие сервисы могут использовать комбинацию механизмов IPC. На рисунке 18 показано, как могут взаимодействовать сервисы в интернет-магазине, когда пользователь добавляет продукт в корзину и выводится уведомление.

Сервисы используют комбинацию уведомлений, запроса / ответа и публикации / подписки. Например, когда пользователь, взаимодействуя с интернет-магазином, отправляет запрос в сервис корзины интернет-магазина, сервис Shopping portal делает запрос через сервис Gateway API на службу отвечающая за корзину. Как только данный запрос поступил, сервис Shopping Cart делает запрос на Quote Storage для сохранения товара в корзине и проверки валидно ли данное действие. В то же время Shopping Cart ожидает ответа от сервиса Quote Storage, и когда ответ приходит, выводит уведомление, используя сервис Notification Service.

Рисунок 18 - Взаимодействие сервисов между собой при добавлении товара в корзину и вывод уведомления

Но в данной схеме имеется риск, частичного сбоя Gateway API Service. Поскольку клиенты и сервисы являются отдельными процессами, сервис может не иметь возможности своевременно отвечать на запрос клиента. Сервис может быть недоступен из-за сбоя или технического обслуживания. Или служба может быть перегружена и очень медленно реагирует на запросы.

Рассмотрим, например, тот же сценарий добавления продукта в корзину. Представим, что Quote Storage не отвечает. Наивная реализация клиента может блокировать бесконечно ожидание ответа. Это не только приведет к ухудшению взаимодействия с пользователем, но во многих приложениях потребует занимания потока выполнения запросов. В конце концов, во время выполнения будут исчерпаны потоки, и они не будут отвечать на запросы, как показано на рисунке 19.

Чтобы предотвратить эту проблему, было решено создать в Cloud Platform компании Netcracker, свои сервисы для обработки частичных сбоев.

Рисунок 19 - Пример сбоя сервиса Quote Storage

Стратегии борьбы с частичными сетевыми сбоями включают в себя:

- Таймауты сети. Ожидание запросов до определённого срока позволит повысить целостность системы;
- Ограничение количества невыполненных запросов. Установка верхней границы для числа невыполненных запросов, которые клиент может иметь с определенной службой;
- Шаблон автоматического выключателя сервиса. Если частота ошибок превышает настроенное пороговое значение, автоматический выключатель должен временно заблокировать сервис для одного клиента, чтобы последующие попытки немедленно завершились неудачей. Если большое количество запросов терпит неудачу, это говорит о том, что служба недоступна и что отправка запросов бессмысленна. По истечении времени ожидания клиент должен повторить попытку и в случае успеха выключить автоматический выключатель;
- Обеспечить запасные варианты - выполнить резервную логику при сбое запроса. Например, вернуть кэшированные данные или значение по умолчанию, например пустой набор рекомендаций.

При использовании обмена сообщениями процессы взаимодействуют посредством асинхронного обмена сообщениями. Клиент отправляет запрос в сервис, отправляя ему сообщение. Если ожидается, что служба ответит, она отправит отдельное сообщение клиенту. Поскольку связь является асинхронной, клиент не блокирует ожидание ответа. Вместо этого клиент ожидает, предполагая, что ответ не будет получен немедленно.

2.2.1 Виды каналов связи для взаимодействия микросервисов между собой

Сообщение состоит из заголовков и тела сообщения. Сообщения обмениваются по каналам связи и любое количество клиентов может отправлять сообщения на канал. Точно так же любое количество

потребителей может получать сообщения из канала. Существует два вида каналов: точка-точка и публикация-подписка. Двухточечный канал доставляет сообщение точно одному из потребителей, которые читают из канала. Сервисы используют двухточечные каналы для стилей взаимодействия один-к-одному, описанных ранее. Канал публикации-подписки доставляет каждое сообщение всем подключенным потребителям. Сервисы используют каналы публикации-подписки для описанных выше стилей взаимодействия один-ко-многим.

На рисунок 20 показано, как интернет-магазин и портал администрирования могут использовать каналы публикации-подписки.

Рисунок 20 - Пример взаимодействия порталов при использовании каналов публикации-подписки

На данной схеме графически представлено, как при изменении количества продуктов у сервиса Product Storage, должно по каналу публикации-подписки сообщить об изменении интернет-магазину и другим порталам.

Использование обмена сообщениями имеет много преимуществ:

- Отключает клиента от сервиса - клиент делает запрос, просто отправляя сообщение на соответствующий канал. Клиент полностью не знает об экземплярах службы. Не нужно использовать механизм обнаружения для определения местоположения экземпляра службы;
- Буферизация сообщений. При использовании протокола синхронного запроса / ответа, такого как HTTP, клиент и служба должны быть доступны на время обмена. Напротив, брокер сообщений ставит в очередь сообщения, записанные в канал, до тех пор, пока потребитель не сможет их обработать. Это означает, например, что интернет-магазин может принимать заказы от клиентов, даже если система выполнения заказов работает медленно или недоступна. Сообщения с заказами просто стоят в очереди;
- Гибкое взаимодействие клиент-сервис - обмен сообщениями поддерживает все стили взаимодействия, описанные ранее;
- Явное межпроцессное взаимодействие - механизмы на основе RPC пытаются заставить вызываемый удаленный сервис выглядеть так же, как и вызов локального сервиса. Однако из-за законов физики и возможности частичного отказа они на самом деле совершенно разные.

Есть, однако, некоторые недостатки использования обмена сообщениями:

- Дополнительная сложность в работе. Система обмена сообщениями - это еще один системный компонент, который необходимо установить, настроить и использовать. Важно, чтобы брокер сообщений был высокодоступным, иначе это повлияет на надежность системы;
- Сложность реализации взаимодействия между запросом и ответом. Взаимодействие типа запрос / ответ требует определенной работы для реализации. Каждое сообщение запроса должно содержать идентификатор канала ответа и идентификатор корреляции. Служба записывает ответное сообщение, содержащее идентификатор корреляции, в канал ответа. Клиент использует идентификатор корреляции для

сопоставления ответа с запросом. Часто проще использовать механизм IPC, который напрямую поддерживает запрос / ответ.

2.2.2 IPC на основе запросов и ответов

При использовании синхронного механизма IPC на основе запроса / ответа клиент отправляет запрос в службу. Служба обрабатывает запрос и отправляет ответ. Во многих клиентах поток, который делает запрос, блокируется во время ожидания ответа. Одним из самых популярных протоколов сетевого взаимодействия сервисов является REST.

На рисунке 21 показан один из способов, с помощью которого интернет-магазин может использовать REST.

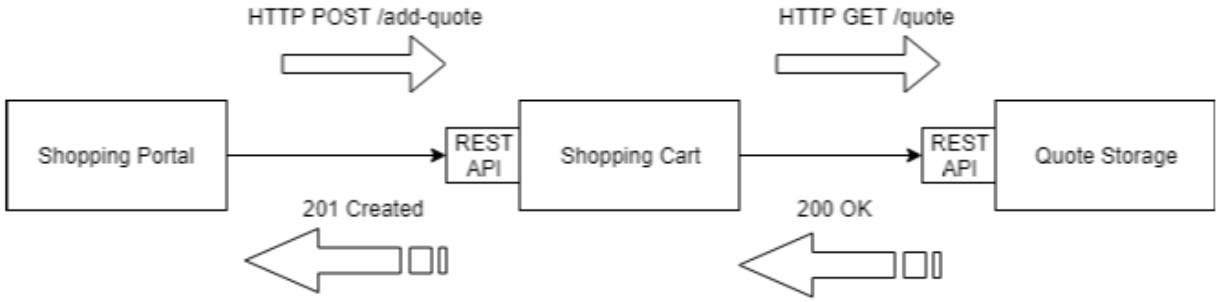

Рисунок 21 - Использование REST для добавления товара в корзину

Когда сервису Shopping Portal необходимо создать новую квоту, он отправляет POST запрос на сервис Shopping Cart. Данный сервис обрабатывает данный запрос и, для полного создания квоты, делает GET запрос на сервис Quote Storage. После успешного создания квоты на стороне Quote Storage, отправляется HTTP ответ со статусом 200, что говорит об успешном выполнении операции. Как только сервис Shopping Cart получает квоту, он проводит свои действия по обработке квоты и возвращает ответ сервису Shopping Portal с HTTP статусом 201, что говорит о том, что квота успешно создана.

Рассмотрим основные преимущества использования протокола, основанного на HTTP:

- HTTP достаточно простой в использовании;

- возможность легко протестировать HTTP API из браузера, используя расширение, такое как Postman, или из командной строки, используя curl (при условии, что используется JSON или другой текстовый формат);
- протокол напрямую поддерживает связь в стиле запроса / ответа;
- не требует промежуточного посредника, что упрощает архитектуру системы.

Есть также и несколько недостатков использования HTTP:

- данный протокол только напрямую поддерживает стиль взаимодействия запрос / ответ. Вы можете использовать HTTP для уведомлений, но сервер всегда должен отправлять HTTP-ответ;
- поскольку клиент и служба взаимодействуют напрямую (без посредника для буферизации сообщений), они оба должны работать в течение всего обмена;
- клиент должен знать местоположение каждого экземпляра службы.

В облачной платформе Netcracker взаимодействие между сервисами будет происходить, используя REST, поскольку в данный момент не требуется использовать канал публикации-подписки, и реализация REST гораздо проще. В будущем, при возникновении необходимости вывода уведомлений, всё также можно пользоваться REST, но в конечном итоге большое количество сервисов, которые будут изменяться одновременно, будут нуждаться в канале публикации-подписки, что будет реализовано в будущем.

2.2.3 Формат ответов сообщений HTTP запросов

Важно использовать формат сообщения на нескольких языках, так как в условиях разработки системы на микросервисной архитектуре есть большая вероятность того, что в будущем технология, на котором основан микросервис, поменяется.

Существует два основных типа форматов сообщений: текстовый и двоичный. Примеры текстовых форматов включают в себя JSON и XML. Преимущество этих форматов заключается в том, что они не только удобочитаемы, но и самодокументируются [8]. В JSON атрибуты объекта представлены коллекцией пар имя-значение. Аналогично, в XML атрибуты представлены именованными элементами и значениями. Это позволяет потребителю сообщения выбирать интересующие его значения и игнорировать остальные. Следовательно, незначительные изменения в формате сообщения могут быть легко обратно совместимы.

Структура XML-документов определяется XML-схемой. Одним из альтернативных вариантов является использование JSON-схемы, как автономной, так и в составе IDL, например, Swagger.

2.3 Основные критерии при работе с контейнерами

Рассмотрев модель облачной платформы компании Netcracker и проведя анализ, как между собой должны взаимодействовать различные компоненты системы, следующим шагом является непосредственно настройка и создание микросервисов, используя виртуализацию на уровне операционной системы, или контейнеры.

Контейнеры (containers) представляют собой средства инкапсуляции приложения вместе с его зависимостями. На первый взгляд контейнеры могут показаться всего лишь упрощенной формой виртуальных машин (virtual machines – VM) – как и виртуальная машина, контейнер содержит изолированный экземпляр операционной системы (ОС), который можно использовать для запуска приложений.

На данный момент существует несколько типов форматов контейнеров, такие как Docker, APPC и LXD. Контейнерная технология начала стандартизироваться с 2015 года в Open Container Initiative (OCI) – стандарты, разработанные компанией Docker и другими лидерами в области

производства контейнеров [14]. В данный момент стандарты содержат две спецификации:

- Спецификация времени выполнения - описывает, как запустить «пакет файловой системы», который распакован на диске;
- Спецификация образов.

На высоком уровне реализация OCI загружает образ OCI, а затем распаковывает этот образ в комплект файловой системы времени выполнения OCI.

Весь этот рабочий процесс должен поддерживать UX, которого пользователи ожидают от контейнерных движков, таких как Docker engine и rkt: в первую очередь, возможность запуска образа без дополнительных аргументов, например:

- docker run example.com/org/app:v1.0.0,
- rkt run example.com/org/app,version=v1.0.0.

Для поддержки этого UX формат изображения OCI содержит достаточно информации для запуска приложения на целевой платформе (например, команда, аргументы, переменные среды и т. д.). Эта спецификация определяет, как создать образ OCI, который обычно выполняется системой сборки, и выводить манифест образа, сериализацию файловой системы (уровня) и конфигурацию образа.

Как уже было сказано ранее, инициаторами создания единых стандартов по работе контейнеров стали несколько компаний, которые являются новаторами в разработке программного обеспечения для работы с контейнерами. Такими компаниями являются, к примеру, Docker и rkt. И это произошло, потому что в последние несколько лет происходит быстрый рост интереса и использования контейнерных решений. Почти все крупные поставщики ИТ-услуг и облачные провайдеры объявили о решениях на основе контейнеров, и в этой области также появилось множество новых компаний. Хотя распространение идей в этом пространстве приветствуется, обещание контейнеров как источника переносимости приложений требует

установления определенных стандартов в отношении формата и времени выполнения.

2.3.1 Выбор контейнерного движка

Для более чёткого понимания, что такое контейнерный движок, необходимо понять, что необходимо автоматизировать при работе с виртуальными машинами, работающими на технологии контейнеризации.

- Компиляция C-программы для системных вызовов и настройка правил SE Linux;
- Настройка и запуск виртуальных машин;
- Работа с жизненным циклом контейнера и работа с консолью для углубленной настройки, или мониторинга системных параметров контейнеров;
- Безопасность и изоляция контейнеров между собой.

Для данных целей рассмотрим 2 контейнерных движка: Docker engine и rkt. Docker - программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему с поддержкой cgroups в ядре, а также предоставляет среду по управлению контейнерами. Изначально использовал возможности LXC, с 2015 года применял собственную библиотеку, абстрагирующую виртуализационные возможности ядра Linux — libcontainer. С появлением Open Container Initiative начался переход от монолитной к модульной архитектуре. CoreOS выпустила rkt в 2014 году как более безопасную, совместимую и открытую альтернативу Docker. Предыдущие версии Docker запускались с правами суперпользователя и в результате уязвимости, существующие внутри контейнеров, могли потенциально дать злоумышленнику привилегии суперпользователя [15]. Еще одна сильная сторона CoreOS - открытость: rkt использует формат контейнера с открытым

исходным кодом, называемый appc, а Docker использует собственный проприетарный формат изображений. Сравнение контейнерных движков будет приведено в таблице А.3.

Подводя итог в выборе контейнерного движка, можно сказать, что у обоих контейнерных предложений есть свои уникальные преимущества, но кроме rkt и его уменьшающихся преимуществ безопасности по сравнению с Docker, предложения двух поставщиков по большей части дополняют друг друга. Например, на предприятиях довольно распространено развертывание контейнеров Docker на CoreOS с его диспетчером кластеров флота. А поскольку недавно предложенная спецификация OCI использует формат изображения Docker 2.2 в качестве основы для общих типов образов контейнера, Docker и CoreOS будут якобы меньше заботиться о дублировании стандартов контейнеров и будут более сосредоточены на создании всеобъемлющего, совместимого набора инструментов для управления всем контейнерная экосистема.

В проекте по работе с контейнерами мы будем использовать Docker, поскольку его преимущества более подходят к реализации облачной платформы, с его большей поддержкой, как со стороны компании разработчика, так и со стороны сообщества. К тому же исследование показывает, что Docker используется в крупных компаниях с большими кластерными системами и прекрасно себя показывает с хорошей стороны.

2.3.2 Образы и контейнеры

Чтобы понять взаимосвязь между образами и контейнерами, необходимо более подробно рассмотреть ключевой элемент технологии, лежащей в основе Docker, UFS (иногда используется термин «каскадно-объединенное монтирование» (unionmount)). Файловые системы с каскадно-объединенным монтированием позволяют подключать несколько файловых систем с перекрытием (или наложением друг друга), причем для пользователя они будут выглядеть как одна файловая система. Каталоги могут содержать файлы из нескольких файловых систем, но если двум

файлам в точности соответствует один и тот же путь, то файл, смонтированный самым последним, скроет все ранее монтированные файлы.

Docker поддерживает несколько различных реализаций UnionFS, включая AUFS, Overlay, devicemapper, BTRFS и ZFS. Реализацию, используемую в конкретной системе, можно определить командой docker info – смотреть содержимое заголовка «Storage Driver». Файловую систему можно заменить, но это рекомендуется только в тех случаях, когда вы точно знаете, что делаете, и хорошо знакомы со всеми преимуществами и недостатками используемых файловых систем.

Образы Docker состоят из нескольких уровней (layers). Каждый уровень представляет собой защищенную от записи файловую систему. Для каждой инструкции в Dockerfile создается свой уровень, который размещается поверх предыдущих уровней. Во время преобразования образа в контейнер (командой docker run или docker create) механизм Docker выбирает нужный образ и добавляет на самом верхнем уровне файловую систему с возможностью записи (одновременно с этим инициализируются разнообразные параметры настройки, такие как IP-адрес, имя, идентификатор и ограничения ресурсов). Поскольку ненужные уровни значительно увеличивают размеры образов (а для файловой системы AUFS установлен строгий лимит, равный 127 уровням), во многих файлах Dockerfile можно обнаружить попытку свести к минимуму количество уровней посредством записи нескольких команд Unix в одной инструкции RUN.

Контейнер может находиться в одном из следующих состояний:

- «создан» (created),
- «перезапуск» (restarting),
- «активен» или «работает» (running),
- «приостановлен» (paused) или «остановлен» (exited).

«Созданным» считается контейнер, который был инициализирован командой docker create, но его работа пока еще не началась. Состояние exited в общем случае соответствует состоянию «остановлен» (stopped), когда в

данном контейнере нет активно выполняющихся процессов (их нет и в «созданном» контейнере, но остановленный контейнер уже запускался, по крайней мере, один раз). Контейнер существует, пока существует его основной процесс. Остановленный контейнер можно перезапустить командой docker start. Остановленный контейнер – это не то же самое, что исходный образ. Остановленный контейнер сохраняет все изменения в его параметрах настройки, метаданных и файловой системе, в том числе и параметры конфигурации времени выполнения, например IP-адрес, которые не хранятся в образах. Состояние перезапуска на практике встречается редко и возникает в тех случаях, когда механизм Docker пытается повторно запустить контейнер после неудачной первой попытки. [30]

На рисунке 22 демонстрируется изоляция поверх файловой системы, осуществляемая технологиями уровня ядра, такими как cgroups, пространства имен и этот факт сделал docker такой многообещающей технологией. Процессы в этом пространстве процессов могут изменять, удалять или создавать файлы в файле «представления объединения», которые будут захвачены на уровне чтения-записи.

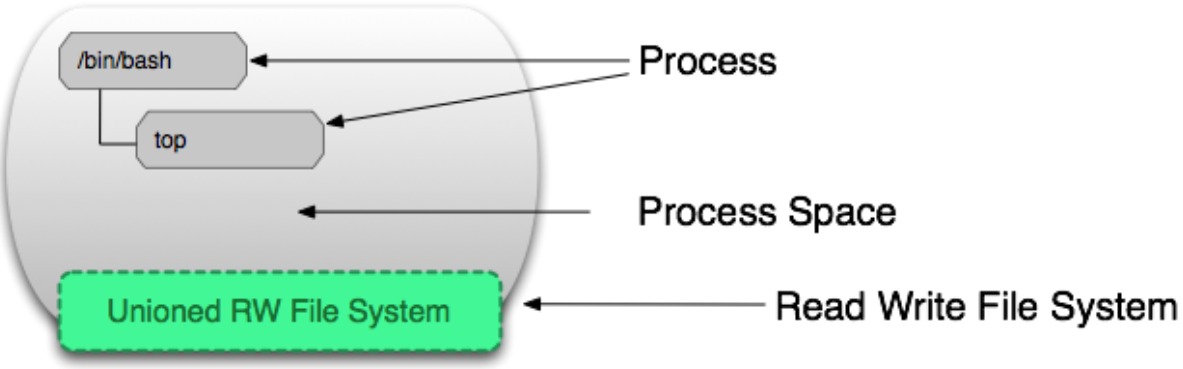

Рисунок 22 Схема взаимодействия между файловой системы и разделов

Наконец, чтобы связать некоторые свободные концы, мы должны определить слой образа. Рисунок 23 показывает слой образа и позволяет нам понять, что слой - это не только изменения в файловой системе. Метаданные

- это дополнительная информация о слое, которая позволяет docker собирать информацию о времени выполнения и времени сборки, а также иерархическую информацию о родительском слое. Оба чтения и чтения-записи слои содержат эти метаданные.

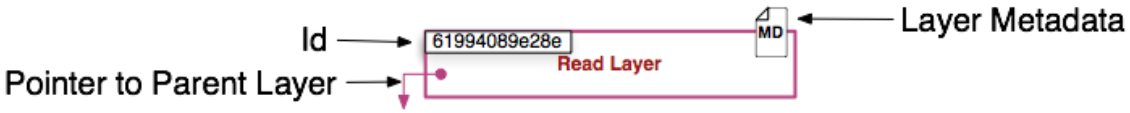

Рисунок 23. Модель docker образа

Кроме того, каждый уровень содержит указатель на родительский уровень с использованием идентификатора (здесь родительские уровни находятся ниже). Если слой не указывает на родительский слой, он находится в нижней части стека.

Выводы

В данной главе были продемонстрированы иллюстрации и диаграммы будущей облачной платформы Netcracker, в которой будут сочетаться технологии для построения полноценного веб-приложения, а также технологии, необходимые для создания полноценной среды, посредством выделения отдельных элементов системы в изолированные друг от друга сервисы.

В данной главе также был описан жизненный цикл запуска и работы контейнеров, с подробным описанием команд, необходимых для развёртывания и запуска докер образа.

Глава 3 Представление авторского решения поставленной в исследовании проблемы

После проведённого анализа проблемы, которую должна решить облачная платформа компании Netcracker, а также исследованию технического стека для решения данной проблемы, в работе были представлены логические модели проектирования системы. Теперь необходимо начать фазу создания инфраструктуры облачной платформы и привести описание производимых работ. Так как тема диссертации сосредоточена именно на контейнерной виртуализации, в данной главе будет подробно описан процесс настройки контейнеров с использованием Docker, а также приведены команды для запуска приложения в контейнерной среде и продемонстрирован процесс настройки оркестратора контейнеров для поддержания процессов CI (continuous delivery) [4].

3.1 Создание микросервиса Shopping Portal

Для создания первого образа контейнера необходимо:

- Установить на операционной системе программное обеспечение Docker Engine,

- Создать специальный файл, который носит название Dockerfile (без расширения).

Dockerfile – это обычный текстовый файл, содержащий набор операций, которые могут быть использованы для создания Docker-образа. В данном файле необходимо указать команду, которая определяет, на чём этот образ основан, и что необходимо выполнить в контейнере, в процессе его установки на виртуальную машину.

Исходя из требований, какую функцию будет выполнять приложение, работающее на контейнере, должно определяться в конфигурации Docker, для загрузки нужных библиотек и пакетов для работы приложения. В данном

случае будет рассматриваться микросервис Shopping Portal, и контейнер обязать иметь для работы этого приложения следующие инструменты:

- Spring framework - ниверсальный фреймворк с открытым исходным кодом для Java-платформы. Также существует форк для платформы .NET Framework, названный Spring.NET;
- Apache Maven - фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM, являющемся подмножеством XML. Проект Maven издаётся сообществом Apache Software Foundation, где формально является частью Jakarta Project. С помощью данного инструмента будут загружаться из сети пакеты необходимые для работы Spring framework;
- Angular - это открытая и свободная платформа для разработки веб-приложений, написанная на языке TypeScript, разрабатываемая командой из компании Google, а также сообществом разработчиков из различных компаний. Angular - это полностью переписанный фреймворк от той же команды, которая написала AngularJS;
- NPM - крупнейший в мире реестр программного обеспечения. Разработчики с открытым исходным кодом со всех континентов используют npm для обмена и заимствования пакетов, а многие организации также используют npm для управления частной разработкой. В случае Shopping Portal, данный инструмент необходим для загрузки и запуска клиентских фреймворков, таких как Angular.

Учитывая все критерии, приведённые выше необходимо сначала создать приложение интернет-магазина со всеми необходимыми компонентами и функциями по работе с бизнес-логикой проекта. Приложение не сразу будет ориентировано на микросервисную архитектуру и будет представлять из себя монолит, позже часть функционала будет

перенесена в отдельный микросервис, а пока полностью всё приложение будет располагаться на одном контейнере.

В целях фокусировки конкретно на технологии контейнеризации, в данной работе будет приведены только некоторые аспекты разработки приложения и больше внимания уделено настройке контейнера и конфигурирования Docker образа.

В первую очередь, приложение сразу делится на 2 части: клиенстку и серверную. Эти 2 части неразрывно связаны, и как правило разрабатываются отдельно друг от друга. В серверной части описывается логика взаимодействия с данными веб-сервера, работа с базой данных, обработка промежуточных данных и фильтрация клиентских запросов. В клиентской части разрабатывается интерфейс, с которым взаимодействует пользователь, описываются в каких случаях необходимо передать данные пользователя в серверную часть для обработки, проверка пользовательских данных и т. д.

Как говорилось ранее, для клиента необходим NPM и Angular, а для серверной части Maven и Spring Boot. Предустановленный npm должен быть уже на той операционной системе, на которой запускается приложение. Сделать это можно, скачав дистрибутив с официального сайта npm. Также npm устанавливается на окружениях, где запускается сборка проекта, о чём будет пояснение в следующих главах. Инструмент Maven необходимо настроить по такому же принципу, как и NPM.

После настройки инструментов для разработки, приложение должно разрабатываться по дизайну проекта, исходя из критериев, которые были приведены в предыдущих главах. Так как данный микросервис пока не разделён на несколько сервисов, в данном приложении должен быть создан компонент Quote Storage, Product Storage и Shopping Cart. В дальнейшем данные компоненты будут отделены от приложения в отдельные микросервисы, а делается это так, поскольку портал администрирования интернет-магазином ещё не процессе разработки, и данные блоки будут нужны только сервису Shopping Portal.

После того, как приложение заработало, необходимо позаботиться об успешной сборке всего приложения в архив для того, чтобы веб-сервер смог распознать приложение, и пользователь начал работу с ним. Для сборки клиентской части, как уже говорилось ранее, будет использоваться NPM. Данный сборщик скомпонует все компоненты клиентского приложения в единый отдельный файл, который должен будет подключён к странице, после запроса за страницей интернет-магазина.

Серверное веб приложение будет собрано с помощью инструмента Maven. С помощью данного инструмента все файлы веб проекта собираются в один веб архив (JAR), такие файлы как: xml, jsp, image, html, css, js файлы сервлетов и т. д.

3.2 Создание и запуск контейнера Shopping Portal.

Все операции по сборке проекта должны быть проделаны до создания Docker образа, так как для создания образа требуются файлы приложения, чтобы скопировать их в файловую систему контейнера.

После того, как проект собран, необходимо в созданный Dockerfile добавить следующую конфигурацию:

- FROM 13-ea-21-jdk-oraclelinux7
- MAINTAINER netcracker.com
- ADD shopping-portal/target/shopping-portal-1.0.0-SNAPSHOT.jar /app/
- EXPOSE 8080
- COPY start.sh /tmp/start.sh
- RUN chmod +x /tmp/start.sh

Инструкция FROM определяет базовый образ ОС. Данное наследование диктуется тем, какие именно технологии будут применимы в данном контейнере. Так как контейнер интернет-магазина должен использовать для запуска JAVA команды и в целом будет написан, используя

данный язык программирования, то выбор родительского образа становится очевидной.

Выбор между базовыми образами велик, поэтому стоит уделить время на изучение разнообразных достоинств и недостатков каждого базового образа. В идеальном случае создание нового образа вообще не потребуется – можно просто использовать существующий образ, объединив с ним свои конфигурационные файлы и/или данные. Во многих случаях такой подход применим для широко распространенного прикладного ПО, например, для СУБД и веб-серверов, для которых доступны готовые официальные образы. Вообще говоря, гораздо лучше воспользоваться официальным образом, чем пытаться сформировать собственный – вам предлагается успешный результат работы людей, которые обладают солидным опытом организации работы ПО внутри контейнеров. Если официальный образ не подходит для вашей работы по некоторой конкретной причине, то попробуйте сформулировать эту причину как тему для обсуждения в исходном проекте, и наверняка найдутся пользователи, встречавшиеся с подобными проблемам или знающие, как их решить.

Инструкция FROM является строго обязательной для всех файлов Dockerfile как самая первая незакомментированная инструкция. Инструкции RUN определяют команды, выполняемые в командной оболочке внутри данного образа. В нашем случае это команда установки пакетов JDK, которая ранее была выполнена вручную.

Команда MAINTAINER определяет метаданные об авторе «Author» для создаваемого образа в заданной строке. Извлечь эти метаданные можно с помощью команды docker inspect -f {{.Author}} IMAGE. Обычно используется для записи имени автора образа и его контактных данных.

Команда ADD копирует файлы из контекста создания или из удаленных URL-ссылок в создаваемый образ. Если архивный файл добавляется из локального пути, то он будет автоматически распакован. Так как диапазон функциональности инструкции ADD достаточно велик, в

общем случае лучше воспользоваться более простой командой COPY для копирования файлов и каталогов в локальном контексте создания или инструкциями RUN с запуском curl или wget для загрузки удаленных ресурсов (с сохранением возможности обработки и удаления результатов загрузки в той же самой инструкции). В данном случае команда ADD копирует и разархивирует веб-архив shopping-portal/target/shopping-portal-1.0.0-SNAPSHOT.jar в файловую систему контейнера, в созданную директорию /app/.

Команда EXPOSE сообщает механизму Docker о том, что в данном контейнере будет существовать процесс, прослушивающий заданный порт или несколько портов. Механизм Docker использует эту информацию при установлении соединения между контейнерами (см. параграф «Соединение между контейнерами» ниже) или при открытии портов для общего доступа при помощи аргумента -P в команде docker run. Но сама по себе инструкция EXPOSE не оказывает никакого воздействия на сетевую среду. В данном случае приложение будет расположено на сетевом порту 8080, что и указано командой EXPOSE в Dockerfile.

Команда COPY используется для копирования файлов из контекста создания в образ. Имеет два формата: COPY источник цель и COPY ["источник", "цель"] – оба копируют файл или каталог из «источника» в контексте создания в «цель» внутри контейнера. Формат JSON-массива обязателен, если путь содержит пробелы. Можно использовать шаблонные символы для определения нескольких файлов или каталогов. Следует обратить особое внимание на невозможность указания путей «источника», расположенных вне пределов контекста создания (например, нельзя указать для копирования файл ../another_dir/myfile). В нашем случае копируется файл start.sh, который хранит в себе пользовательские настройки, переменные окружения и данный скрипт должен быть запущен уже внутри контейнера, при его старте.

Команда RUN запускает заданную инструкцию внутри контейнера и сохраняет результат. В данном случае запускается скрипт файла start.sh с правами доступа чтения файла.

Теперь мы можем создать образ, выполнив команду docker build в том же каталоге, где расположен наш Dockerfile:

docker build -t shopping-portal.

Для команды docker build необходим Dockerfile и контекст создания образа (build context) (который может быть пустым). Контекст создания – это набор локальных файлов и каталогов, к которым можно обращаться из инструкций ADD и/или COPY в Dockerfile и которые обычно определяются как путь к нужному каталогу. Например, при использовании команды создания образа docker build -t shopping-portal, определяет контекст создания как '.', то есть текущий рабочий каталог. Все файлы и каталоги, расположенные по указанному пути, формируют контекст создания образа и передаются в демон Docker как часть процесса создания. В тех случаях, когда контекст не определен, – если задан только URL для Dockerfile или содержимое Dockerfile передается по программному каналу из стандартного потока ввода (STDIN), – контекст создания данного образа считается пустым. Флаг –t в данном случае определяет название контейнера, который уже позже можно запустить.

После ввода данной команды операция создания отображается в консоли командной строки и выглядит примерно так, как показано на рисунке Б.2.

Чтобы лучше понять, как устроена архитектура и создание образа, рассмотрим основные компоненты Docker, которые графически представлены на рисунке 24.

В центре расположен демон Docker (Docker daemon), ответственный за создание, запуск и контроль работы контейнеров, а также за создание и хранение образов. Контейнеры и образы представлены в правой части

диаграммы. Демон Docker запускается командой docker daemon, обычно об этом заботится операционная система хоста.

Клиент Docker, размещенный в левой части диаграммы, используется для диалога с демоном Docker по протоколу HTTP. По умолчанию это соединение устанавливается через сокет домена Unix, но также может использоваться TCP-сокет для поддержки соединений с удаленными клиентами или дескриптор файла для сокетов, управляемых systemd.

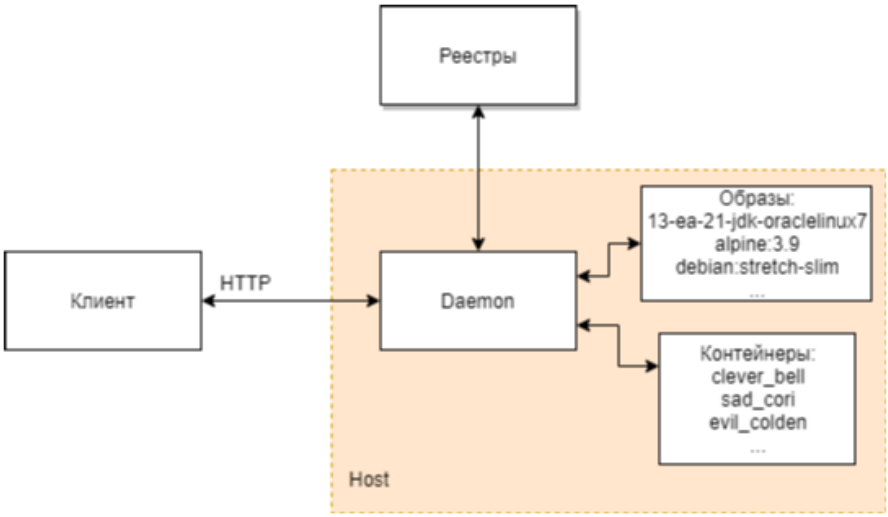

Рисунок 24 - Общая схема взаимодействия главных компонентов Docker

Так как все операции обмена данными выполняются по протоколу HTTP, можно без затруднений организовать соединение с удаленными демонами Docker и разработать привязки (bindings) к нужному языку программирования, но при этом следует учитывать особенности реализации этих возможностей, например, обязательное наличие контекста создания (building context). Интерфейсы прикладного программирования, используемые для организации обмена данными с демоном, четко определены и подробно документированы, что позволяет писать программы, взаимодействующие напрямую с демоном, без использования клиента

Docker. Клиент и демон Docker распространяются как отдельные независимые бинарные файлы.

Реестры Docker используются для хранения и распространения образов. Реестром, выбираемым по умолчанию, является Docker Hub, на котором хранятся тысячи общедоступных образов, а также управляемые «официальные» образы. Многие организации создают собственные реестры, которые используются для хранения коммерческих и приватных образов и для устранения накладных расходов, связанных с загрузкой образов через Интернет. Не исключением стала и компания Netcracker, но в целях безопасности, образы в данных примерах загружаются из Docker Hub. Демон Docker загружает образы из реестров по запросу docker pull. Кроме того, он выполняет автоматическую загрузку образов, указанных в запросе docker run и в инструкции FROM файла Dockerfile, если эти образы недоступны на локальной системе. В случае образа для контейнера shopping-portal, используется базовый образ 13-ea-21-jdk-oraclelinux7.

Для запуска контейнера основываясь на образе shopping-portal необходимо выполнить команду: docker run shopping-portal

Данная команда немедленно запускает контейнер без предварительной загрузки образа, так как образ создан локально. Выполнение команды создания контейнера происходит очень быстро несмотря на то, что внутри docker machine происходит множество операций:

Docker подготовил и запустил контейнер, выполнил выгрузку приложения и запустил команды по запуску приложения из start.sh и Dockerfile. Затем в консоль терминала вывел сообщения о работе контейнера, представление на рисунке Б.3.

Если попытаться сделать нечто подобное в обычной виртуальной машине, то придется ждать несколько секунд, а может быть, даже несколько минут. После выполнения данной команды контейнер с уникальным идентификатором 84a0b38750cc создаётся, в чём можно удостовериться,

введя команду в консоль терминала, результат которого приведён на рисунке Б.4:

docker ps –a

Остановить контейнер, или удалить его можно, зная его имя с помощью команд: docker stop shopping-portal и docker rm shopping-portal. Для создания других микросервисов

Несмотря на то, что логически микросервисы сильно отличаются между собой, образы контейнеров имеют общую структуру, которая может меняться в зависимости от некоторых требований, но в целом конфигурация образа остаётся без изменений.

3.3 Сопровождающие технологии и инструменты для сборки и управления контейнерами

Рассмотрев создание первого контейнера в облачной платформе Netcracker, можно предположить, что для запуска такого контейнера по большей части время уходит именно на создание самого приложения, а контейнер является некой базовой технологией, с помощью которой, совокупность приложений, работающих на данной технологии виртуализации, образует систему, архитектуру которой называют – микросервисной архитектурой. Но в данном случае речь шла о сборке и запуске лишь одного контейнера и для того, чтобы разрабатывать большую систему с множествами контейнерами потребуются некоторые инструменты для упрощения разработки с Docker контейнерами.

Сами по себе механизм Docker и реестр Docker Hub не предоставляют завершенного полноценного решения для работы с контейнерами. Для большинства пользователей потребуются сервисы поддержки и вспомогательное ПО, например система управления кластерами, инструменты обнаружения сервисов, расширенные сетевые функциональные возможности и проч.

Стратегия «заменяемых батареек» в первую очередь реализуется на уровне интерфейсов прикладного программирования, позволяя подключать компоненты непосредственно к движку Docker, но ее также можно наблюдать на примере формирования дистрибутивных пакетов Docker как независимых автономных бинарных файлов, которые с легкостью заменяются аналогами от третьих сторон. На текущий момент можно привести следующий список технологий поддержки, предоставляемых Docker:

- Docker Compose – инструмент для создания и выполнения приложений, скомпонованных из нескольких Docker-контейнеров. Такие компоновки используются главным образом при разработке и тестировании, но гораздо реже в производственной среде. В параграфе «Автоматизация с помощью Compose» можно найти более подробную информацию;
- Docker Machine устанавливает и конфигурирует Docker-хосты на локальных и удаленных ресурсах. Кроме того, Machine конфигурирует клиента Docker, упрощая процедуру переключения между средами;
- Kitematic представляет собой графический пользовательский интерфейс для операционных систем Mac OS и Windows, обеспечивающий запуск и управление контейнеров Docker;
- Openshift - это платформа облачной разработки как услуга (PaaS), разработанная Red Hat. Это платформа разработки с открытым исходным кодом, которая позволяет разрабатывать и развертывать свои приложения в облачной инфраструктуре. Это очень полезно при разработке облачных сервисов;
- Artifactory - это продукт JFrog, который служит менеджером бинарных репозиториев. Бинарный репозиторий является естественным расширением репозитория исходного кода в том смысле, что он будет хранить результаты вашего процесса сборки,

часто обозначаемые как артефакты. Так как образы контейнеров будут обладать разными версиями и их будет некоторое множество, то данный репозиторий также очень необходим для разработки. Также в Artifactory могут публиковаться не только образы контейнеров, но и другие библиотеки, например maven архивы и npm зависимости;

- GitLab - сайт и система управления репозиториями кода для Git, из дополнительных возможностей: собственная вики и система отслеживания ошибок. ПО доступно в системе управления пакетами Omnibus. Данный проект выполняет функцию системы контроля версий кода и его хранения. Данный проект был выбран благодаря его простому интегрированию с другими инструментами для сборки и запуска приложения, с такими как Jenkins;
- Jenkins - позволяет автоматизировать часть процесса разработки программного обеспечения, в котором не обязательно участие человека, обеспечивая функции непрерывной интеграции. Работает в сервлет-контейнере, например, Apache Tomcat. Поддерживает инструменты системы управления версиями, включая AccuRev, CVS, Subversion, Git, Mercurial, Perforce, Clearcase и RTC. Может собирать проекты с использованием Apache Ant и Apache Maven, а также выполнять произвольные сценарии оболочки и пакетные файлы Windows. Сборка может быть запущена разными способами, например, по событию фиксации изменений в системе управления версиями, по расписанию, по запросу на определённый URL, после завершения другой сборки в очереди. [16]

Процессом настройки и связки всех инструментов в одну экосистему должна заниматься отдельная команда, с опытом администрирования и поддержки таких систем. В совокупности инструменты Docker, Jenkins, Artifactory и GitLab образуют процесс сборки, который показан на рисунке 25.

На данном рисунке представлены блоки состояний, о которых ещё не было описаний:

SCM проверка - управленческая концепция и организационная стратегия, заключающаяся в интегрированном подходе к планированию и управлению всем потоком информации о продуктах, услугах, возникающих и преобразующихся в логистических и производственных процессах предприятия, нацеленном на измеримый совокупный экономический эффект (снижение издержек, удовлетворение спроса на конечную продукцию) [15]. Например, в нашем случае происходит проверка целостности модели продуктов в базе данных.

Интеграционные тесты необходимы для проверки приложения на работоспособность в условиях близкие к выпуску. Таким образом, среда Jenkins может запустить алгоритм по тестированию ПО в специально отведённой среде и если тесты успешны, можно опубликовать данный образ в Artifactory, а если нет, то это будет считаться ошибкой сборки.

Результатом схемы ниже станет либо сообщение об ошибке, чтобы можно было исправить приложение и запустить сборку ещё раз, либо публикация Docker образа в Artifactory. Графически Jenkins представляет из себя последовательность блоков, время выполнения, которого выводится в веб интерфейсе. Конфигурация, и последовательность данных блоков, настраивается в специальном Jenkinsfile, который формирует конфигурацию и импортируется в Jenkins. Данная конфигурация приведена на рисунке В.1, а интерфейс показан на рисунке В.2.

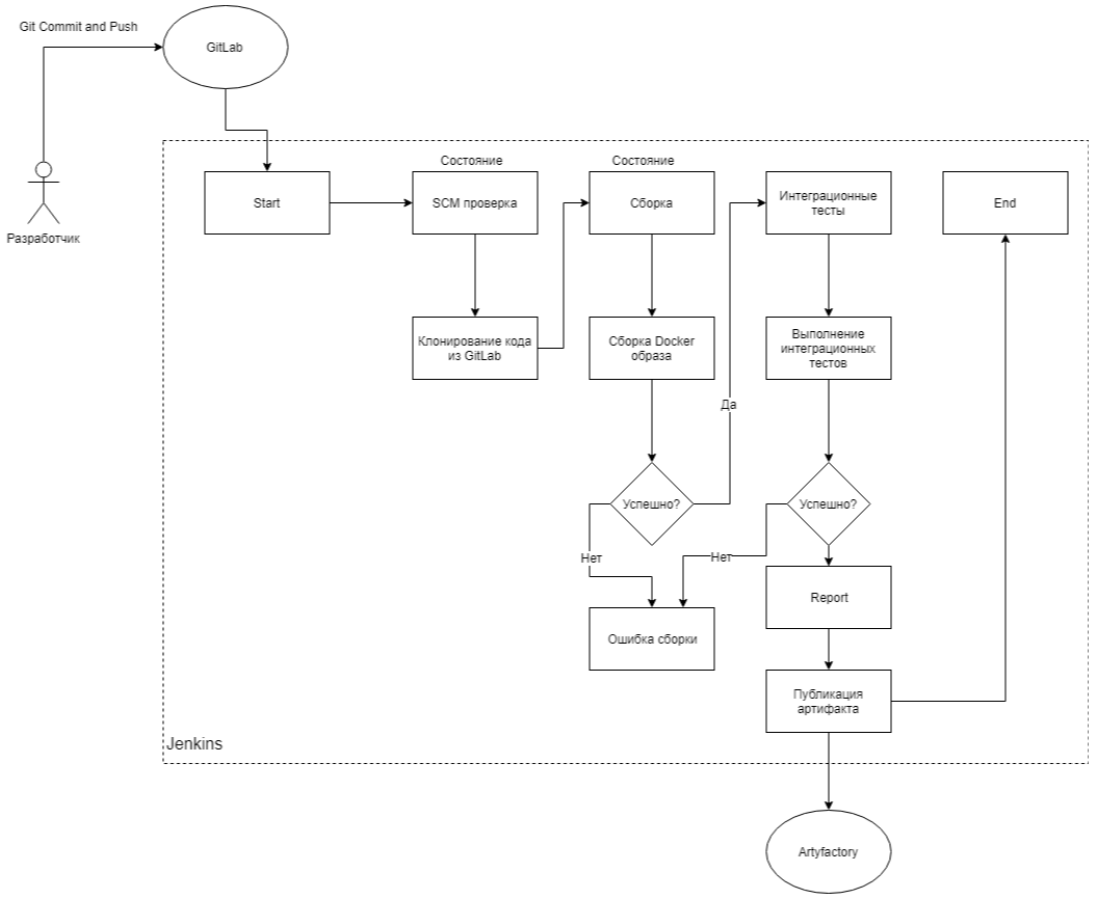

Рисунок 25 - Процесс сборки и публикации Docker образа

Таким образом, сборкой и публикацией образов контейнера будет заниматься автоматизированная система Jenkins, которая отлично подходит не только для сборки, но и для публикации в репозиторий Artifactory, а также тестирование приложение в специализированной изолированной среде.

3.4 Управление контейнерами в облачной платформе Netcracker

Обеспечение работы нестандартной системы всегда связано с большим количеством административных и прочих трудных задач, и проблем. Наблюдение и обслуживание для каждого отдельного элемента системы быстро становится невозможным, элементы должны быть стандартизированы, чтобы обеспечить их оперативный ремонт и обновление. Если элемент системы порождает проблему, то его следует удалить или заменить, а не исправлять, пытаясь вернуть в рабочее состояние.

Для решения этих серьезных проблем существуют разнообразные программные средства и решения, охватывающие в большей или меньшей степени, следующие области поможет оркестрация (orchestration) – обеспечение совместной работы всех элементов системы. Запуск контейнеров на соответствующих хостах и установление соединений между ними [25]. Организационная система также может включать поддержку масштабирования, автоматического восстановления после критических сбоев и инструменты изменения балансировки нагрузки на узлы. Примеры оркестраторов рассматривались ранее, но в данном случае для облачной платформы компании Netcracker будет использоваться OpenShift от компании RedHat.

3.4.1 Анализ оркестратора контейнеров Openshift

Поскольку мы можем запускать всю разработку, тестирование и производственную настройку в самом OpenShift, конвейер от кода к производству является автономным и простым в управлении. Поскольку спрос на эти сервисы меняется в вашей среде, мы можем легко масштабировать каждую услугу (включая Artifactory) независимо, чтобы максимизировать использование ресурсов и производительность.

Благодаря полной совместимости Artifactory Enterprise с OpenShift весь поток CI / CD может оставаться внутри нашего частного кластера OpenShift. Для разработки платформы можно использовать Artifactory для определения правильных артефактов и развертывания их в среде непрерывной интеграции (CI), работающей внутри OpenShift. Как только сборки продвигаются, их можно отправить обратно в Artifactory и развернуть / распространить оттуда. Типичный рабочий процесс может выглядеть примерно так:

- Разработчик передает код в любой VCS (в нашем случае GitLab),
- Это вызывает сборку на Jenkins, которая выбирает артефакты, которые ему нужны, из Artifactory,

- Когда сборка завершена, и она соответствует критериям приемлемости, она повышается и развертывается в Artifactory.

Это говорит о том, что соединение OpenShift Container Platform или OpenShift Online с реестром Artifactory может упростить облачную разработку и обеспечить путь для непрерывной интеграции от кода к кластеру.

Для настройки Openshift необходимо проделать следующие шаги:

- Перенаправить Red Hat OpenShift в локальные репозитории пакетов. Преимуществом Artifactory для проекта Red Hat OpenShift является его способность кэшировать наши внешние зависимости в локальный репозиторий. Это может ускорить сборки, предоставляя более надежный и согласованный доступ к удаленным ресурсам, помогая устранить любую зависимость от сети или внешних репозиториев. Artifactory обеспечивает интегрированную поддержку форматов пакетов, таких как Maven, npm, Python, NuGet, Gradle, Go и других. Языковая структура облачной платформы Netcracker может опираться как минимум на одного из этих менеджеров пакетов. Например, Spring Boot на основе Java требует Maven, а Angular требует npm. Red Hat OpenShift должен быть направлен на использование локального репозитория с учетом учетных данных, необходимых для доступа к ним. Эти учетные данные хранятся в виде секретов, созданных в проекте Red Hat OpenShift;
- Создание секретов доступа. Чтобы сохранить учетные данные в Red Hat OpenShift, создайте секрет-файл с помощью командной строки ОС. Как правило, эта учетная информация хранится в файле конфигурации менеджера пакетов, и мы можем извлечь нашу секретную информацию непосредственно из этого файла. Например, для Angular необходимо создать секрет-файл npm, который содержит учетные данные для доступа к вашему

хранилищу npm в Artifactory. Их можно извлечь непосредственно из файла .npmrc, настроенного для использования Artifactory;

- Наконец, мы должны создать сервисы сборки, развертывания, обслуживания и маршрутизации для платформы. Это делается с использованием файла YAML для каждого сервиса. В каждом из них свяжите секрет rt-docker-registry с действиями push и pull для службы. Например, фрагменте для настройки службы сборки приведённым на рисунке Г.1.

После выполнения всех шагов и правильной настройки репозиториев и систем сборки, в openshift должны быть применены следующие компоненты:

- поды – это группы контейнеров, которые развертываются и планируются совместно. В Openshift такие группы образуют неделимую единицу планирования в противоположность отдельным контейнерам в других системах. Группа обычно содержит от 1 до 5 контейнеров, совместно обеспечивающих работу некоторого сервиса. В дополнение к этим пользовательским контейнерам Openshift запускает вспомогательные контейнеры для сервисов ведения журналов и контроля. В Openshift такие группы считаются непостоянными – в процессе развития системы они могут создаваться и уничтожаться;
- flat networking space – в Openshift функционирование сетевой среды значительно отличается от работы сети с Docker-шлюзом, принимаемой по умолчанию. В сетевой среде Docker по умолчанию контейнеры существуют в закрытой подсети и не могут напрямую обмениваться данными с контейнерами на других хостах без перенаправления портов на хосте или без использования механизма прокси. В Openshift контейнеры в одной группе (pod) совместно используют один IP-адрес, но все адресное пространство является «плоским» для всех групп (pods), таким образом, все группы (pods) могут обмениваться информацией друг с другом без какого-либо

преобразования сетевых адресов (NAT). Это существенно упрощает управление многохостовыми кластерами, но при этом не поддерживаются внутренние каналы связи, а организация сетевой среды для одного хоста (или, более точно, для одной группы) становится чуть более сложной. Поскольку контейнеры одной группы (pod) совместно используют общий IP-адрес, они могут обмениваться данными, используя порты по адресу localhost (это означает, что вы сами должны управлять использованием портов внутри группы (pod));

- services – сервисы являются стабильными точками входа, к которым можно обращаться по имени. Сервисы могут устанавливать соединения с группами контейнеров (pods), используя селекторы ярлыков. Например, мой сервис cache может установить соединение с несколькими группами (pods) redis, определяемыми по селектору ярлыка "type": "redis". Этот сервис будет автоматически посылать циклические запросы, распределяемые по заданным группам (pods). Openshift настраивает для кластера DNS-сервер, который отслеживает появление новых сервисов и обеспечивает обращение к ним по имени как в коде приложения, так и в файлах конфигурации;
- шаблон - описывает набор объектов, которые могут быть параметризованы и обработаны для создания списка объектов для создания OpenShift. Объекты для создания могут включать в себя все, что пользователи имеют разрешения на создание в рамках проекта, например услуги, создание конфигурации и конфигурации развертывания. Шаблон также может определять набор меток для применения к каждому объекту, определенному в шаблоне.

Кроме того, возможны создание и настройка сервисов, которые указывают не на группы контейнеров, а на другие, уже существующие

сервисы, например на внешние прикладные программные интерфейсы или базы данных.

Для создания контейнера в кластере Openshift требуется обозначить его шаблон, после чего, с помощью консоли Openhsift загрузить его туда. После этого автоматическая система развёртывания развернёт контейнер в кластере, о чём будет сигнализировать созданная пода (Pods) в графическом интерфейсе. Пример шаблона показан в приложении А (Рис. А10).

Контроллеры репликации (replication controllers) представляют собой обычный способ создания экземпляров групп контейнеров в Openshift (как правило, при работе в Openshift вы не используете интерфейса командной строки Docker). Эти контроллеры предназначены для управления и отслеживания больших количеств работающих групп контейнеров (называемых репликами (replicas)), связанных с некоторым сервисом. Например, контроллер репликации может отвечать за поддержку в рабочем состоянии пяти групп Redis. Если одна из групп становится неработоспособной, то контроллер немедленно запускает в работу новую группу. Если количество реплик нужно сократить, контроллер остановит работу всех лишних групп. Несмотря на то, что применение контроллеров репликации для управления всеми экземплярами групп добавляет еще один уровень конфигурации, тем не менее при этом значительно улучшаются устойчивость к критическим сбоям и надежность системы.

На рисунке 26 показана часть кластера Kubernetes с двумя группами контейнеров, созданными контроллером репликации и объявленными некоторым сервисом. Сервис посылает циклические запросы, распределяемые по этим группам, которые выбираются по значению ярлыка tier. Внутри группы все контейнеры совместно используют один IP-адрес. Контейнеры внутри группы могут обмениваться данными, используя порты по адресу localhost. Сервису присвоен отдельный IP-адрес, доступ к которому открыт для всех. [7]

Пример конфигурации контроллера репликаций для сервиса shopping-portal приведён на рисунке Г.2.

Определение контроллера репликации состоит в основном из:

- Желаемое количество реплик (которое можно настроить во время выполнения),
- Определение модуля для создания реплицированного модуля,
- Селектор для идентификации управляемых модулей.

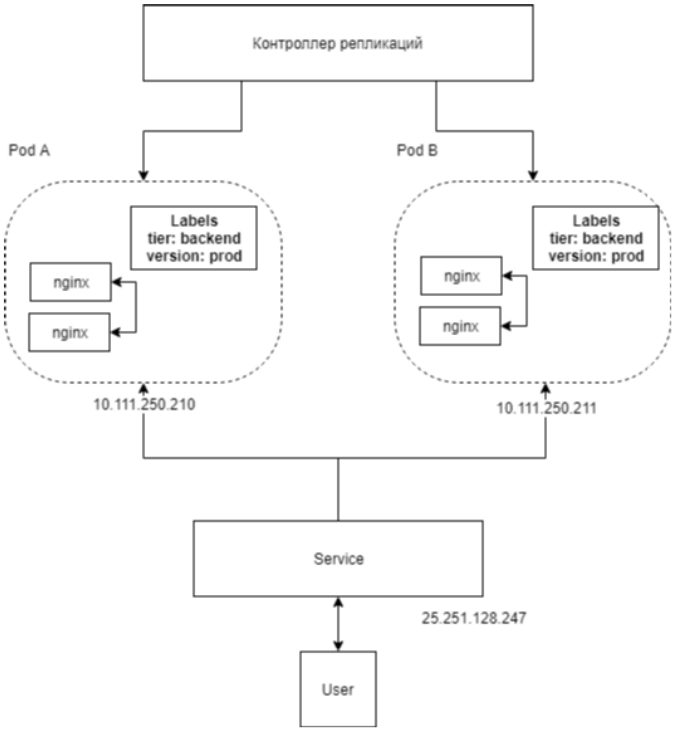

Рисунок 26 - Пример кластера в Openshift

Селектор - это простой набор меток, которые должны иметь все модули, управляемые контроллером репликации. Таким образом, этот набор меток включен в определение модуля, которое создает контроллер репликации. Этот селектор используется контроллером репликации, чтобы определить, сколько экземпляров модуля уже запущено, чтобы настроить его при необходимости. Данная конфигурация также настраивается в YAML файле конфигуратора контейнера в Openshift.

3.4.2 Взаимодействие между сервисами в openshift

Для маршрутизации сервисов в Openshift необходимо использовать специальные маршруты (routes). Маршрут OpenShift - это способ раскрыть сервис, дав ему внешнее имя хоста, например www.example.com. [5]

Определенный маршрут и конечные точки, определенные его сервисом, могут использоваться маршрутизатором для обеспечения именованного подключения, которое позволяет внешним клиентам получать доступ к нашим приложениям. Каждый маршрут состоит из имени маршрута (не более 63 символов), селектора служб и (необязательно) конфигурации безопасности.

В OpenShift можно развертывать маршрутизаторы в кластере OpenShift, что позволяет внешним клиентам создавать маршруты. Уровень маршрутизации в OpenShift является подключаемым, и два доступных подключаемых модуля маршрутизатора предоставляются и поддерживаются по умолчанию. Маршрутизаторы OpenShift обеспечивают сопоставление имен внешних хостов и распределение нагрузки для служб по протоколам, которые передают различающую информацию непосредственно в маршрутизатор; имя хоста должно присутствовать в протоколе, чтобы маршрутизатор мог определить, куда его отправить.

Подключаемые модули маршрутизатора предполагают, что они могут связываться с портами хоста 80 и 443. Это позволяет внешнему трафику направляться на хост и затем через маршрутизатор. Маршрутизаторы также предполагают, что сеть настроена так, что она может получить доступ ко всем модулям в кластере.

Маршрутизаторы поддерживают следующие протоколы: HTTP, HTTPS (с SNI), WebSockets, TLS с SNI. Маршрутизатор использует селектор сервисов, чтобы найти сервис и конечные точки, поддерживающие сервис. Предоставляемая сервисом балансировка нагрузки обходится и заменяется собственной балансировкой нагрузки маршрутизатора. Маршрутизаторы наблюдают за API-интерфейсом кластера и автоматически обновляют свою собственную конфигурацию в соответствии с любыми соответствующими

изменениями в объектах API. Маршрутизаторы могут быть контейнерными или виртуальными. Пользовательские маршрутизаторы могут быть развернуты для передачи модификаций объектов API во внешнее решение маршрутизации.

Чтобы реализовать маршрутизатор, необходимо в первую очередь понять, что запросы имен хостов должны разрешаться через DNS к маршрутизатору или набору маршрутизаторов. Предлагаемый метод состоит в том, чтобы определить облачный домен с помощью записи DNS с подстановочными знаками, указывающей на виртуальный IP-адрес, поддерживаемый несколькими экземплярами маршрутизатора на назначенных узлах. Для того чтобы сервисы были доступны извне, маршрут OpenShift позволяет связать сервис с внешним именем хоста. Это имя пограничного хоста затем используется для маршрутизации трафика в сервис.

Когда два маршрута требуют одного и того же хоста, побеждает самый старый маршрут. Если в одном и том же пространстве имен определены дополнительные маршруты с другими полями пути, эти пути будут добавлены. Если используется несколько маршрутов с одним и тем же путем, приоритет имеет самый старый. Конфигурация маршрута микросервиса shopping-portal, показана на рисунке Г.2.

Совокупность некоторого количества маршрутов облачной платформы Netcracker, после полной настройки и создании ещё нескольких контейнеров показано на рисунке Г.3. Для создания новых маршрутов нет необходимости в создании конфигурационного файла. Можно воспользоваться графическим инструментом Openshift. Данный процесс показан в рисунке Г.4.

Интерфейс после отправки формы автоматически подставляет значения в консоль Openshift и автоматически настраивает маршрутизацию между сервисами. Минусом данного подхода является перенастройка маршрутов, после перезагрузки некоторых сервисов. Например, при перезагрузке портальных сервисов, необходимо заново настроить маршруты.

Выводы

Благодаря сбору требований, анализу технологий и поэтапной разработке системы, получилось добиться создания прототипа облачной платформы компании Netcracker. Прототип платформы в данный момент состоит из 4 микросервисов: Shopping-portal, Admin-Portal, API Gateway, Quote Storage. Данные микросервисы обмениваются между собой данными по выделенному каналу, используя протокол HTTP и маршрутизацию оркестратора контейнеров Openshift, который также автоматически управляет контейнерами платформы, масштабировав систему под нужды клиентов. Также используя веб-интерфейс Openshift, можно производить мониторинг каждого микросервиса платформы.

Для автоматической сборки, тестирования и публикации образов контейнеров, были настроены такие инструменты, как Jenkins и Artyfactory. Их работа позволяет быстро и без риска установить новый образ контейнера Openshift, а также следить за процессом сборки и тестировать образы контейнеров до их официального выпуска. Имея в наличии всего несколько микросервисов, облачная платформа способна оформить заказ клиента, с помощью микросервисов Shopping Portal, Quote Storage и Api Gateway, а также, если требуется вмешательство в процесс покупки со стороны администратора портала, можно воспользоваться веб-порталом Admin Portal. Интерфейс приложения Shopping Portal, представлен на рисунке Б.5.

Заключение

В данной выпускной работе по теме проектирование виртуальных серверов на основе технологии контейнеризации, используя каналы передачи данных компании Netcracker, было разработано продуктовое решение, которое будет обладать высокой скоростью работы, а также, хорошо масштабируется и легко поддаётся изменениям.

Для реализации данного проекта в первую очередь было приведено подробное описание архитектуры проекта, схема взаимодействия сегментов приложения между собой и другие детали проектирования облачной платформы компании Netcracker. Далее во время проведения обзора технологий виртуализации, было пояснено понятие виртуализации, и основные типы виртуализации, существующие на данный момент, а также причины выбора определённого стека технологий для проектирования конкретно облачной платформы компании Netcracker. После этого, были изучены принципы работы программного обеспечения Docker, причины выбора именно этого решения. В конце, были подробно описаны технические детали реализации смоделированной архитектуры, а именно:

- Принципы сетевого взаимодействия контейнеров между собой в облачной платформе, используя каналы передачи данных;
- Описание механизма создания и запуска Docker образа контейнера;
- Краткое описание настройки автоматизированных систем сборки и хранения Docker образов в облачной платформе;
- Обзор принципов управления несколькими контейнерами и использование Openshift для оркестрации и кластеризации в технологии контейнеризации.

Несмотря на то, что облачная платформа компании Netcracker в данный момент функционирует и имеет хорошие производительные показатели, при увеличении количества контейнеров, будет увеличиваться и количество

инструментов и нагрузки на оркестратор и кластер в целом. Таким образом, в настоящее время необходимо производить исследование на предмет выявления слабых мест в системе и устранять их, применяя новые подходы для технологии контейнеризации, которые с каждым годом обновляются новыми инструментами и поддержкой от самых разных производителей решений основанной на технологии виртуализации на уровне ОС.

Также важно отметить следованию принципу изолированности микросервисов. Данный подход не только позволит улучшить кодовую базу, но и также поспособствует улучшению отказоустойчивости и масштабированию системы. Данные улучшения достигаются путём принципа изолированности сбоев: микросервисы должны уметь развёртываться и работать самостоятельно. Без этого принципа облачная платформа Netcracker лишится главных критериев – масштабируемости и отказоустойчевости.

В ходе проектирования облачной платформы были выявлены следующие положения:

- Каждый из контейнеров должен являться изолированной и самостоятельной частью системы.
- Технологии используемые в каждом контейнеры могут отличаться в зависимости от поставленных задач для конкретного контейнера. Для взаимодействия должны быть использованы общие стандарты, реализация которых также может отличаться.
- Выбор оркестратора для управления контейнерами является важным элементом проектирования системы при использовании контейнерной виртуализации. Процессы, которые внедряют оркестраторы, помогают производить мониторинг системы, масштабирование и полноценное управление жизненным циклом сервисов.

– При проектировании системы основанной на микросервсиной архитектуре, необходимо заранее продумать элемент масштабирования и версионирования сервисов.

Список используемой литературы

1. Андрей Маркелов Введение в технологии контейнеров и Kubernetes; изд. ДМК Пресс - 2019 – 194 с.
2. Арно Лорэ Проектирование веб-API; изд. ДМК Пресс – 2020 – 440 с.
3. Виникус Ф.П. Шаблоны микросервиса и лучшие практики; изд. ДМК Пресс - 2019 – 212 с.
4. В чем разница между непрерывной интеграцией, непрерывным развертыванием и непрерывной доставкой. [Электронный ресурс] – 2015 – Режим доступа: http://merrigrove.blogspot.com/2015/10/visualizing-docker-containers-and-images.html
5. Грант Шиплей, Грахам Дуплентон Openshift для разработчиков; ред. ДМК Пресс - 20178 – 182 с.
6. Дэвис Дженнифер, Дэниелс Кэтрин Философия DevOps. Искусство управления IT; изд. Питер, Серия «Бестселлеры O’Reilly» – 2017 – 416 с.
7. Джон Арундел, Джастин Домингус Kubernetes для DevOps: развертывание, запуск и масштабирование в облаке; изд. Питер, Серия «Бест-селлеры O’Reilly – 2020 – 384
8. Илья Корнипаев Требования для программного обеспечения: рекомендации по сбору и документированию; изд. Книга по требованию – 2019 – 118 с.
9. Клеппман Мартин Высоконагруженные приложения. Программирование масштабирование поддержка; изд. Питер, Серия «Бест-селлеры O’Reilly» – 2018 – 740 с.
10. Крис Ричардсон Микросервисы. Паттерны разработки и рефакторинга; изд. Питер – 2019 – 544 с.

11. Лапидус, Л. В. Технологии электронной коммерции и их влияние на формирование новых рынков и трансформацию традиционных бизнес-моделей//Экономика и предпринимательство – 2018 - №6

12. Лиз Райс Безопасность контейнеров. Фундаментальный подход к защите контейнеризированных приложений; изд. Питер, Серия «Бестселлеры O'Reilly» - 2021 – 224 с.

13. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения; изд. Питер – 2018 – 352 с.

14. Миллан, Сейерс Docker на практике; изд. ДМК-Пресс – 2020 – 516 с.

15. Парминдер Кочер Микросервисы и контейнеры Docker; изд. ДМК-Пресс – 2019 – 240

16. С.В. Назаров (2016) Архитектура и проектирование программных систем: монография / С.В. Назаров – 2-е изд., переаб. и доп. – Москва: ИНФРА-М, 2020 – 374с

17. Сэм Ньюмэн Создание микросервисов; ред. Питер, Серия «Бестселлеры O'Reilly - 2016 - 304 с.

18. Таненбаум, Э. С. Современные операционные системы / Э. С. Таненбаум. - 3-е изд. – ред. Питер – 2016 – 576 с.

19. Akentev, E., Tchitchigin, A., Safina, L., and Mzzara, M. Verified type checker for Jolie programming language [Электронный ресурс] – 2017 – Режим доступа: https://arXiv.org/pdf/1703.05186.pdf.

20. Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality - [Электронный ресурс] – 2019 – Режим доступа: https://www.researchgate.net/publication/331282866_Microservices_in_Industry_Insights_into_Technologies_Characteristics_and_Software_Quality

21. Burns Brendan Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services; изд. O'Reilly Media; 1st edition – 2018 – 166 с.

22. Černý, T., Donahoo, M.J., and Trnka, M. Contextual understanding of microservice architecture: current and future directions - [Электронный ресурс] – 2019 – Режим доступа: https://www.researchgate.net/publication/322842819_Contextual_understanding_of_microservice_architecture_current_and_future_directions

23. Inter-Process Communication in a Microservices - [Электронный ресурс] – 2016 – Режим доступа: https://dzone.com/articles/building-microservices-inter-process-communication-2

24. Jaehyeong Leea, Changyong Uma, Jinjae Shinb, Jongpil Jeong Dynamic Microservices to Create Scalable and Fault Tolerance Architecture, 23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems - [Электронный ресурс] - 2019 - https://www.researchgate.net/figure/CPPS-and-Edge-Computing-Architecture-based-on-OPC-UA-server-Fig-2-Detailed-module_fig1_335808750

25. Kalske M., Mäkitalo N., Mikkonen T. (2018) Challenges When Moving from Monolith to Microservice Architecture. In: Garrigós I., Wimmer M. (eds) Current Trends in Web Engineering. ICWE 2017. Lecture Notes in Computer Science, vol 10544. Springer, Cham - [Электронный ресурс] - 2019 - https://helda.helsinki.fi/bitstream/handle/10138/237054/challenges_moving_monolith.pdf?sequence=1&isAllowed=y

26. K.I. Volovicha, S.A. Denisova, S.I. Malkovskyb Deployment of parallel computing in a hybrid high-performance cluster based on virtualization technologies// Procedia Computer Science 186 - 2018 - 14th International Symposium «Intelligent Systems»

27. Larucces, X., Santamaria, I., Colomo-Palacios, R., and Ebert, C. «Microservices». In: IEEE Software, 35/3: 96-100. Kalske, M. Transforming monolithic architecture towards microservice architecture. M.Sc. Thesis, Univ. of

Helsinki – [Электронный ресурс] - 2019 - Режим доступа: https://helda.helsinki.fi/bitstream/handle/10138/234239/transforming-monolithic-architecture.pdf?sequence=2&isAllowed=y

28. Max Plauth, Lena Feinbube and Andreas Polze A Performance Evaluation of Lightweight Approaches to Virtualization, CLOUD COMPUTING: The Eighth International Conference on Cloud Computing, GRIDs, and Virtualization – [Электронный ресурс] - 2017 – Режим доступа: https://link.springer.com/chapter/10.1007/978-3-319-67262-5_3

29. Nikhil Pathania Learning Continuous Integration with Jenkins. A beginner's guide to implementing Continuous Integration and Continuous Delivery using Jenkins – изд. Packt Publishing – 2016 – 542 с.

30. Xia, C., Zhang, Y., Wang, L, Coleman, S., and Liu, Y. «Microservice-based cloud robotics system for intelligent space» In: Robotics and Autonomous Systems - Электронный ресурс] - 2018 – Режим доступа: https://www.researchgate.net/publication/328270568_Microservice-based_cloud_robotics_system_for_intelligent_space

31. Zimmermann, O. «Microservices Tenets: Agile Approach to Service Development and Deployment, Computer Science - Research and Development» - [Электронный ресурс] - 2017 - Режим доступа: https://ifs.hsr.ch/uploads/tx_icscrm/1_msa-pospaperzio4summersoc2016v15nc.pdf

Приложение А

Элементы необходимые для разработки системы работающей на микросервисной архитектуре

Таблица А.1. - Инструменты для развёртывания приложения микросервисной архитектуры

Категории инструментов	Примеры инструментов	Описание
Контейнерный движок	Docker LXC RKT	Автоматизированное создание контейнеров на хосте
Обнаружение сервисов	Zookeeper Eureka Etcd	Именования, поддержка информации о конфигурации, обеспечения распределенной синхронизации и предоставления групповых служб.
Мониторинг	Graphite InfluxDV Sensu	Периодическое отслеживание прогресса любой деятельности путем систематического сбора и анализа данных и информации.
Оркестровка контейнеров	Mesos Kubernetes Openshift	Контейнерная оркестровка - это управление жизненными циклами контейнеров, особенно в больших динамических средах.
Отказоустойчевость	Finagle Hystrix Proxygen	Сохранение полной работоспособности при отказе в работе отдельных модулей системы.

Продолжение таблицы А.1

Категории инструментов	Примеры инструментов	Описание
Непрерывный выпуск	Ansimble Jenkins Drone	Практика автоматизации всего процесса выпуска программного обеспечения.
Хаос-инжиниринг	Chaos Monkey Simian Army Pumba	Процесс тестирования распределенной вычислительной системы, чтобы убедиться, что система может противостоять неожиданным сбоям в работе.
Обнаружение сервисов	SmartStack Prada Envoy	Обнаружение, которое сводит к минимуму усилия по настройке для администраторов - автоматическое обнаружение устройств и предлагаемых услуг по сети.
Бессерверные вычисления	AWS Lambda Azure Functions GoogleCloud	Метод предоставления серверных услуг по мере использования.
Сервисная mesh сеть	Linkerd Istio Conduit	Способ предоставления сервисов, интегрированных в вычислительный кластер, через API, которые не требуют каких-либо аппаратных устройств.

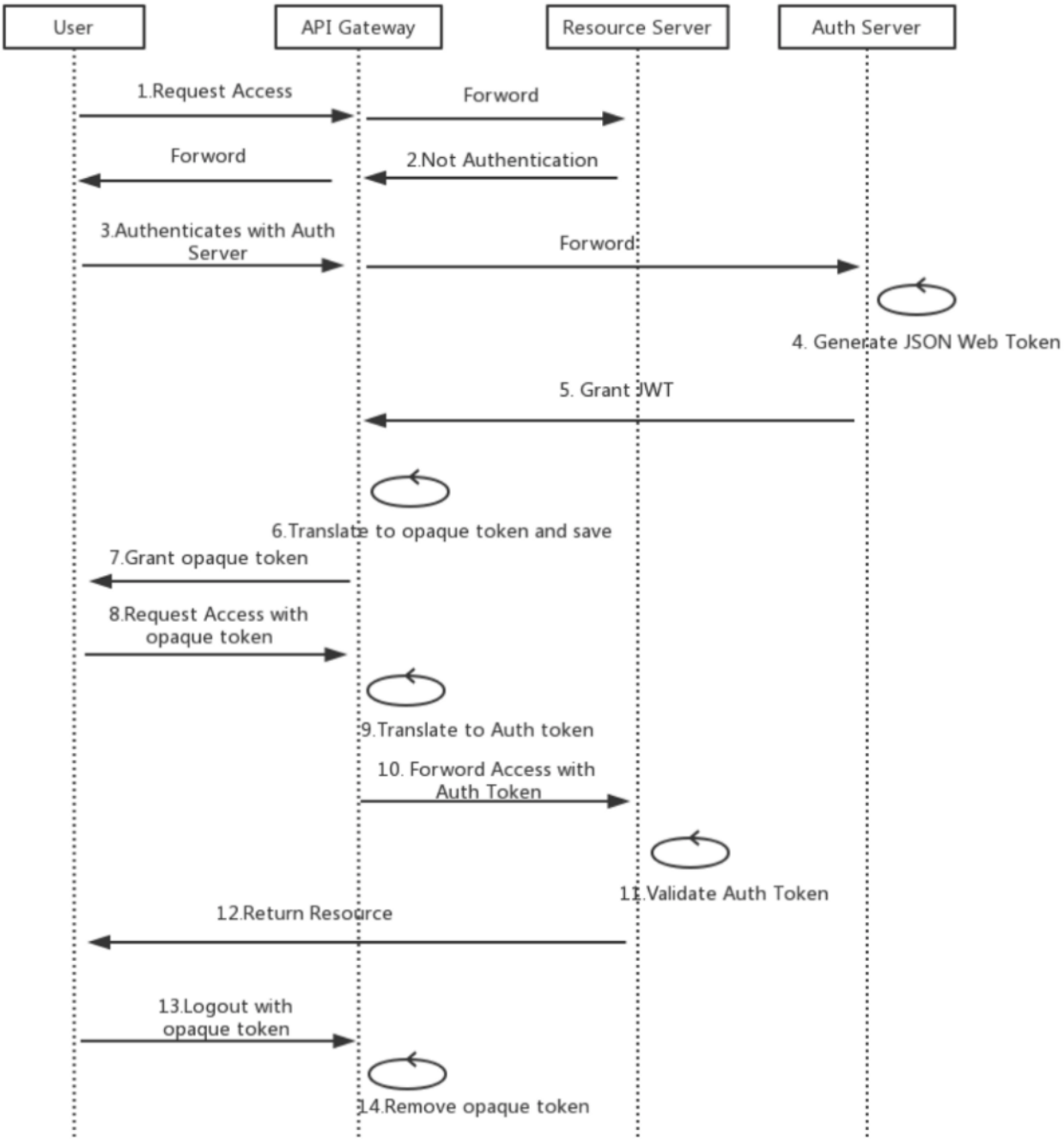

Рисунок А.1 – Схема взаимодействия пользователем с сервисами приложения

Таблица А.2 - Сравнение контейнерных движков

Критерий	rkt	Docker engine
Набор возможностей	CoreOS позиционирует rkt как более ориентированное на безопасность контейнерное решение. Кроме того, его контейнер Linux от CoreOS представляет собой облегченную операционную систему с открытым исходным кодом, основанную на ядре Linux.	Для расширенных возможностей решение Docker's Datacenter предлагает оркестровку корпоративных контейнеров, управление приложениями и защиту корпоративного уровня.
Простота использования	Для удобного запуска, остановкой, или удалением контейнеров, компания Core OS разработала платформу CoreOS Tectonic, которая позволяет визуально управлять контейнерами и кластерами CoreOS. контейнерами	В свою очередь компания Docker также сделала удобный в использовании Kitematic, который ни в чём не уступает аналогу от Core Os.

Продолжение таблицы А.2

Критерий	rkt	Docker engine
Поддержка сообщества	CoreOS поддерживает активный центр ресурсов сообщества	Аналогичным образом, портал Docker's Community и форумы являются популярными ресурсами самообслуживания среди пользователей Docker.
Цены и поддержка	Оплачиваемые варианты CoreOS полностью связаны с поддержкой, например, предложение поддержки Tectonic / Kubernetes стоит от 3000 долларов за 10 серверов.	Коммерческие продукты Docker включают Docker Datacenter (от 150 долларов в месяц) для контейнеров поддержки поставщиков за брандмауэром и Docker Cloud (от 7 месяцев / 5 репозиториев) для создания / доставки частных репозиториев.
API и расширяемость	CoreOS использует gRPC - высокопроизводительную универсальную среду RPC с открытым исходным кодом - для предоставления своим предложениям RESTful API.	Чтобы не отставать, Docker предлагает полный набор API-интерфейсов REST и SDK, которые позволяют контролировать каждый аспект стека контейнеров из пользовательских приложений.

Продолжение таблицы А.2

Критерий	rkt	Docker engine
Сторонние интеграции	CoreOS - это больше DIY / barebones в этом отношении, хотя все его проекты доступны на GitHub.	Docker Hub является компания облачных веб-сервисом, который предлагает более 100 000 бесплатных приложения, общественные и частные реестры, с официальными репозиториями от ведущих производителей: от сторонних разработчиков Nginx и Ubuntu в MongoDB и Redis.
Опыт других компаний в использовании данных движков	CoreOS используется известными фирмами, такими как CA Technologies, Verizon, Viacom, Salesforce. com , DigitalOcean	Docker используется многими ведущими современными предприятиями: ADP, PayPal, Ebay, BBC News, Spotify, Lyft, Expedia, Groupon, GE Appliances, ING и Uber.

Приложение Б

Элементы, относящиеся к облачной платформе компании Netcracker

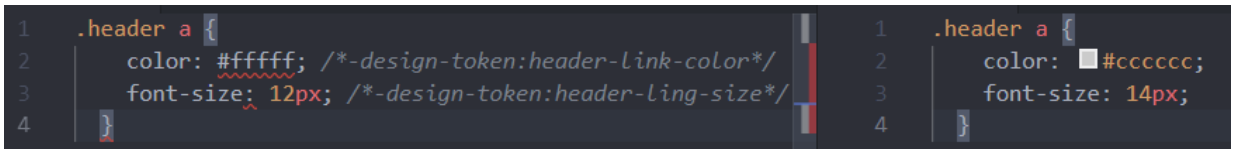

Рисунок Б.1 – Форматирование файла стилей с помощью сервиса кастомизации

Рисунок Б.2 – Информация об успешном создании docker образа в терминале

Рисунок Б.3 – Информация об успешном запуске контейнера shopping-portal в терминале

```
λ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
84a0b38750cc        shoping-portal      "/tmp/entrypoint.sh …"   3 minutes ago       Up 3 minutes        8080/tcp            epic_turing
```

Рисунок Б.4 – Информация о запущенном контейнере shopping-portal в терминале

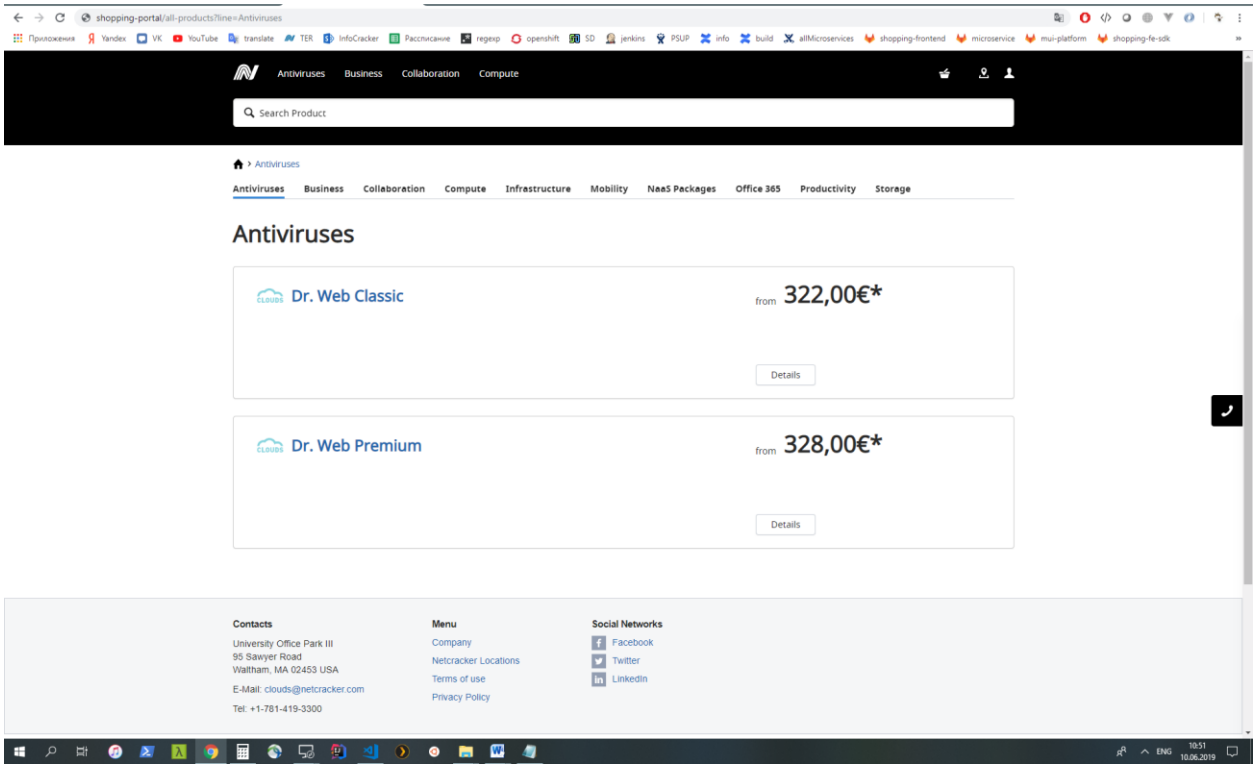

Рисунок Б.5 – Интерфейс приложения shopping-portal

Приложение В

Элементы относящиеся к инструментам автоматизации процесса сборки и деплоя контейнеров.

```
node('docker') {

    stage 'Checkout'
    checkout scm
    stage 'Build & UnitTest'
    sh "docker build -t accountownerapp:B${BUILD_NUMBER} -f Dockerfile ."
    sh "docker build -t accountownerapp:test-B${BUILD_NUMBER} -f Dockerfile.Integration ."

    stage 'Integration Test'
    sh "docker-compose -f docker-compose.integration.yml up --force-recreate --abort-on-container-exit"
    sh "docker-compose -f docker-compose.integration.yml down -v"
}

```

Рисунок В.1 - Конфигурация деплоя в Jenkins

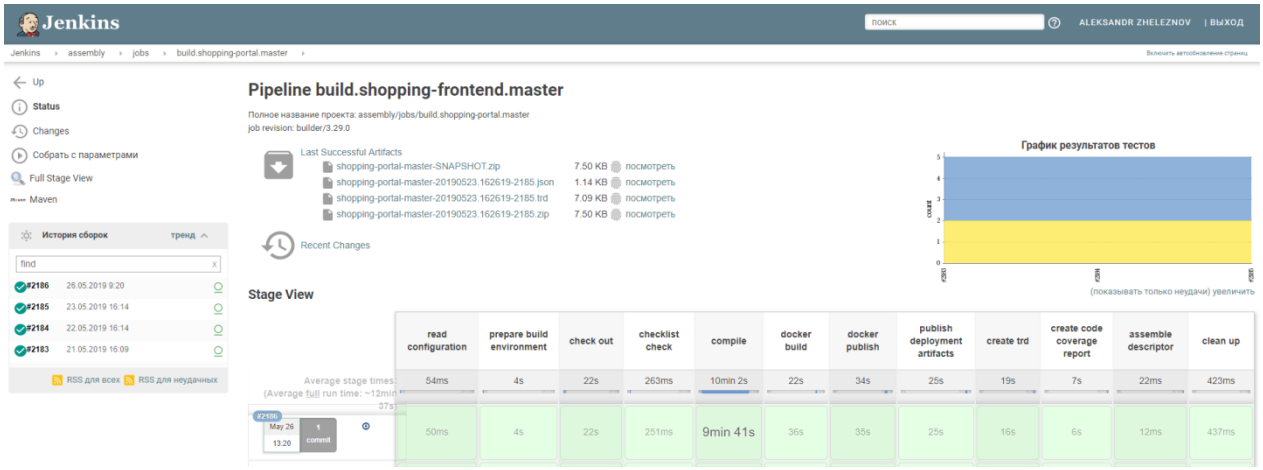

Рисунок В.2 – Информация об успешном деплое приложения shopping-portal в Jenkins

Приложение Г

Элементы платформы, относящиеся к конфигурации оркестратора автоматизированной работы контейнеров.

```yaml
  selector:
    name: shopping-portal
  template:
    metadata:
      creationTimestamp: null
      labels:
        name: shopping-portal
    spec:
      volumes:
        - name: client-credentials
          secret:
            secretName: microservices-ui-client-credentials
            defaultMode: 420
        - name: defaultsslcertificate
          secret:
            secretName: defaultsslcertificate
            defaultMode: 420
      containers:
        - name: microservices-ui
          image: >-
            artifactorycn.netcracker.com:17004/cloud-catalog/shopping-portal:build544
          args:
            - /usr/bin/bash
            - /tmp/start.sh
```

Рисунок Г.1 – YAML конфигурация контейнера shopping-portal

YAML JSON

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: shopping-portal
  namespace: dmp-dev1
  selfLink: /apis/apps/v1/namespaces/dmp-dev1/deployments/shopping-portal
  uid: c8d3ad29-b339-49e7-ad91-b3d885439f9c
  resourceVersion: '349425756'
  generation: 3
  creationTimestamp: '2021-11-24T10:16:30Z'
  labels:
    app_name: dmp
    app_version: master-20211126.202048-28811
    name: shopping-portal
    tier: backend
  annotations:
    deployment.kubernetes.io/revision: '2'
    kubectl.kubernetes.io/last-applied-configuration: >
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"labels":{"name"
        :"shopping-portal","tier":"backend"},"name":"shopping-portal","namespace":"dmp-dev1"},"spec"
        :{"replicas":1,"selector":{"matchLabels":{"name":"shopping-portal"}},"strategy"
```

Рисунок Г.2 - Пример конфигурации контроллера репликаций для сервиса shopping-portal

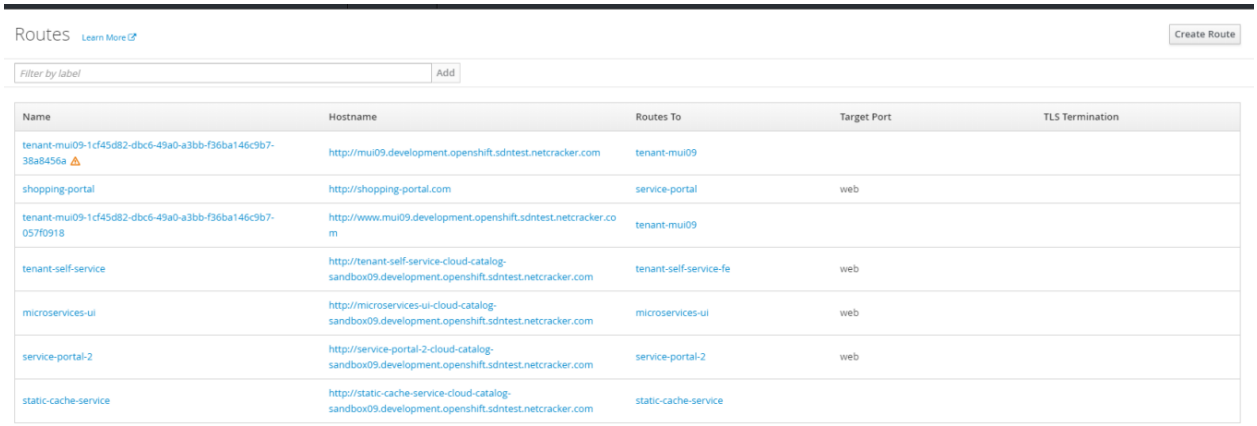

Рисунок Г.3 - Конфигурация маршрута микросервиса shopping-portal

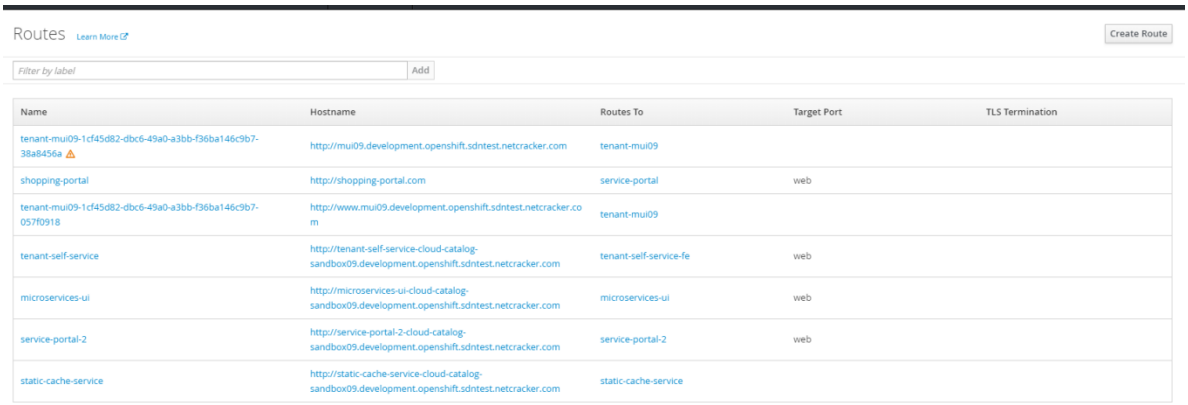

Рисунок Г.4 - Совокупность некоторого количества маршрутов облачной платформы Netcracker