Содержание

Аннотация

В рамках проекта мы рассмотрим преимущества микросервисной архитектуры, важность и актуальность использования DevOps-практик. Также создадим шаблон web-приложения, использующий kubernetes-манифесты и helm-charts - два наиболее популярных инструмента для автоматизации развертывания и управления приложениями в кластерах.

Полученный продукт может быть использован в качестве основы для дальнейших более масштабных проектов. Настроенные в нем инструменты автоматизации позволят ускорить процесс разработки и снизить количество ошибок в производственной среде. А как известно, время от идеи до реализации - является конкурентным преимуществом на рынке.

Ключевые слова

Микросервисы, DevOps, Kubernetes, CI/CD, Helm.

1 Введение

1.1 Предметная область

DevOps (сокр. от development и operations) - это одновременно методология, культура и набор инструментов. Главной целью является объединение процессов разработки и эксплуатации для повышения эффективности, скорости и безопасности разработки и доставки программного обеспечения (ПО), что обеспечивает конкурентное преимущество для компаний и их клиентов [1-2].

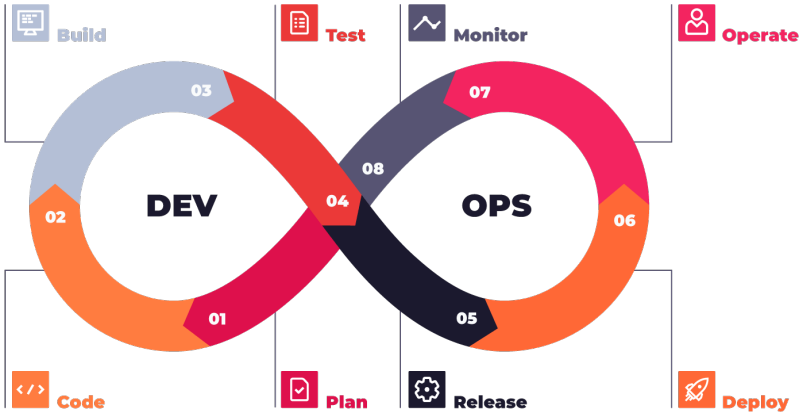

Рис. 1.1: Цикл DevOps [3].

Движение DevOps возникло в 2009 году: программисты создавали код и передавали его системным администраторам, которые занимались его эксплуатацией и поддержкой, чтобы ничего не сломалось. Такой подход удлинял разработку ПО - многие рабочие процессы были не согласованы, а коммуникация между двумя типами сотрудников усложнялась. Осознав эту проблему, разработчики, тестировщики и менеджмент ИТ-компаний решили объединиться и сформировать общий подход к разработке, тестированию и релизу, который позволил бы получить не просто готовый код, а еще и возможность представить его заказчику максимально быстро. Так и появился DevOps - связующий элемент между разработчиками, тестировщиками и системными администраторами.

1.2 Актуальность

Сегодня крайне важно поддерживать эффективный процесс разработки IT-продуктов, поскольку время от идеи до реализации - является конкурентным преимуществом на рынке. Поэтому все больше компаний начинают вднерять культуру DevOps - одну из самых быстрорастущих и революционных тенденций в области технологий в наши дни, благодаря которой можно ускорить весь жизненный цикл ПО. Его глобальная стоимость достигла 7 миллиардов долларов в 2021 году и, как ожидается, вырастет до 57,9 млрд долларов к 2030 году при среднегодовом темпе роста 24,2% [4].

По данным на 2017 год компании, которые приняли решение следовать принципам DevOps, сократили количество багов при развёртке приложения в пять раз, а время от фиксации до выпуска в 440 раз [5]. В настоящее время данная методология внедряется в ИТ-компании по всему миру: Amazon, Facebook, Netflix, Sony Pictures и другие.

При этом в крупных компаниях есть своя IT-инфраструктура, свои макеты приложений, которые можно использовать при создании новых. В рамках данной работы я попробовала разработать свой упрощенный шаблон, который в будущем можно взять за основу, чтобы ускорить процесс разработки и сфокусироваться на главном - на создании новой функциональности.

Основными преимуществами DevOps являются:

- Быстрая и качественная доставка продукции
- Большая масштабируемость и доступность
- Большая автоматизация

В своем проекте я сфокусируюсь на следующих DevOps-tools, которые помогают автоматизировать, улучшать и ускорять различные этапы жизненного цикла ПО [6-7]:

Микросервисы (Golang) - эффективный подход, при котором приложение строится как набор небольших сервисов, которые взаимодействуют друг с другом через определенный интерфейс, такой как API на основе HTTP. Микросервисы разрабатываются с учетом бизнес-возможностей, и каждый из них имеет единственную цель. Основными плюсами являются: масштабируемость, локализация отказов, гибкость в выборе инструментов, возможность развертывать их как отдельные службы или как кластер служб, более простой процесс включения в проект новых членов команды.

Контроль версий (Git) - включает в себя управление кодами в нескольких версиях с изменениями и историей изменений, поэтому становится легко просматривать коды, восстанавливать их, объединять изменения и тд.

Оркестрация контейнеров (Kubernetes). Kubernetes — самая популярная платформа для оркестрации контейнеров, которая стала важным инструментом для команд DevOps. Она автоматизирует развертывание, управление, масштабирование, сетевое взаимодействие и доступность контейнерных приложений. По сути, Kubernetes помогает разработчикам создавать «инфраструктуру как код», а также помогает им управлять конфигурациями среды кодирования как кодом.

Непрерывная интеграция и непрерывная доставка (CI/CD) (GitLab)

Непрерывная интеграция - это практика разработки ПО, при которой разработчики часто объединяют или интегрируют изменения кода в основную ветвь кода, где запускается автоматическое тестирование для быстрого выявления и устранения проблем с кодом. Затем происходит непрерывная доставка и развертывание: автоматизированный выпуск нового или измененного кода в промежуточную, а затем в рабочую среду. Данная практика позволяет командам устранить операционные накладные расходы, человеческие ошибки и повторяющиеся шаги.

1.3 Цели и задачи

Основная цель проекта - получить работающую связку микросервисов с масштабируемой и понятной архитектурой и возможностью автоматического развертывания в кластере Kubernetes.

Задачи:

1 Написать связку микросервисов (сервис авторизации, публикации постов и комментариев, real-time чат и frontend)

2 Ознакомиться с видами архитектурных диаграмм (в частности, с моделью C4), составить контекстную и контейнерную диаграммы.

3 Подготовить kubernetes-манифесты и helm-charts для развертывания сервисов в k8s-кластере.

4 Настроить CI/CD для автоматического деплоя и последующего развертывания новых версий приложения в Kubernetes.

1.4 Обзор литературы.

В качестве продукта было предложено реализовать приложение для ведения блога с поддержкой real-time чата. В качестве аналога можно выделить, например, следующий. Однако, здесь нет чата и поддержки DevOps (манифестов, версионирования и автодеплоя). Кажется, что сложно найти аналогичные решения с похожей на нашу задумкой в открытом доступе.

2 Выбор инструментов

2.1 Архитектура ПО. Преимущества микросервисной архитектуры

Большинство проектов, особенно стартапов сначала выбирают монолит, так как нужен быстрый старт при минимальных ресурсах и не ясны перспективы. Но с увеличением размеров и сложности приложения управлять монолитом становится всё труднее. Любое изменение в монолитном приложении может иметь каскадный эффект, оказывая влияние на другие части приложения, что существенно усложняет интеграцию и развертывание в промышленной системе. Также присутствует риск отказа всего приложения.

Альтернатива монолиту - микросервисы. Микросервисная архитектура представляет собой метод организации архитектуры, основанный на ряде независимо развертываемых служб. У этих служб есть собственная бизнес-логика и база данных с конкретной целью. Обновление, тестирование, развертывание и масштабирование выполняются внутри каждой службы.

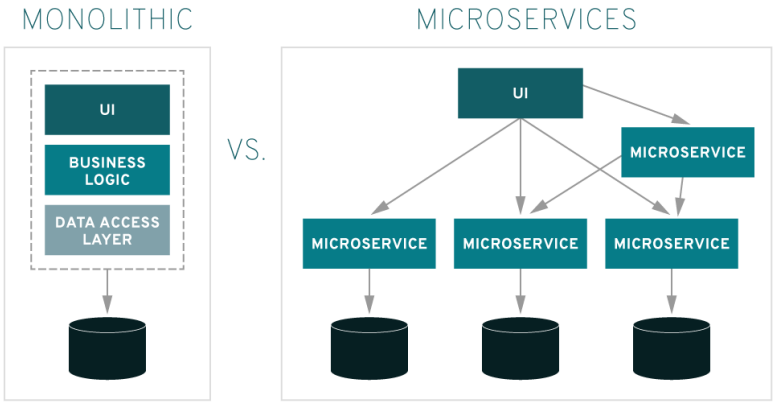

Рис. 2.1: Монолитная и микросервисная архитектура [8].

Давайте рассмотрим преимущества разработки приложения с использованием микросервисной архитектуры [9-10]:

- **Гибкое масштабирование**. Поскольку микросервисы изначально являются распределенными и могут развертываться в кластерах, становится возможным динамическое горизонтальное масштабирование через границы сервисов. Если нагрузка на микросервис достигает предела, можно быстро развернуть новые экземпляры этого сервиса в сопутствующем кластере и таким образом снизить нагрузку.
- **Локализация отказов**. В монолитных решениях ошибка или сбой в одном компоненте приводит к сбою всего приложения. С микросервисной архитектурой мы извлекаем

выгоду из так называемой изоляции сбоев: если какой-то компонент выходит из строя, остальные продолжают работать как обычно.

- **Раздельное развертывание**. В монолите любые изменения, внесенные в любую часть кодовой базы, потребуют повторного развертывания всего приложения целиком. В зависимости от стратегии развертывания это может привести к тому, что вся платформа будет недоступна в процессе развертывания. С микросервисами все становится намного лучше - вам нужно только повторно развернуть «сервис» (компонент), в котором были внесены изменения.
- **Частые релизы**. Одним из основных преимуществ микросервисной архитектуры являются частые и быстрые циклы релизов. Благодаря микросервисной архитектуре, которая выступает ключевым элементом непрерывной интеграции и непрерывной поставки (CI/CD), команды могут экспериментировать с новыми возможностями, а если что-то пойдет не так - выполнить откат и вернуться к предыдущей версии. Это упрощает обновление кода и ускоряет вывод новых возможностей на рынок.
- **Гибкость**. В монолите вам, скорее всего, придется использовать один единственный язык программирования на протяжении всей разработки вашего приложения. Хотя с этим проблем может не быть, в некоторых сценариях могут быть другие языки, которые работают намного лучше в определенном наборе задач, или например хочется воспользоваться библиотеками, доступными на определенном языке (допустим, библиотеками ML в Python). Архитектура микросервисов дает возможность сочетать лучшее из всех языков программирования.
- **Упрощенная координация команды**. При работе над средними и крупными проектами очень часто возникают проблемы с включением членов команды в существующий проект. С микросервисами это становится сделать гораздо проще, поскольку приложение разделено на гораздо меньшие независимые подмножества, и новым членам команды достаточно освоить подмножество, над которым им нужно работать.

Неудивительно, что микросервисная архитектура стала набирать популярность. Например, в 2009 году Netflix, страдавший от перебоев в обслуживании и проблем с масштабированием, начал постепенный процесс рефакторинга своей монолитной архитектуры, сервис за сервисом, в микросервисы [11].

Помимо Netflix также и другие, одни из самых инновационных и прибыльных предприятий в мире, такие как Amazon, Uber, Etsy, Spotify отчасти объясняют огромный успех своих ИТ-инициатив внедрением микросервисов.

2.2 Язык программирования

Микросервисы могут быть реализованы с помощью множества фреймворков, версий и инструментов. Java, Golang, Python, C++, Node JS и .Net - лишь немногие из них. Однако в Python, например, отсутствуют возможности многопоточности и параллелизма, что делает его менее масштабируемым. Многочисленные тесты показывают, что по этой же причине Go работает в 30 раз быстрее него. Изучив еще несколько статей, я остановила свой выбор на Go, который становится все более популярным языком для микросервисов [12-13].

Почему именно Go?

- Статическая типизация, сборка мусора, поддержка параллелизма, безопасность указателей, отражение, перегрузка функций, вывод типов, перегрузка операторов.
- Простота в освоении и использовании благодаря простому синтаксису и отличной документации.
- Чистый и читаемый код с четко определенным руководством по стилю, который легко понимать и поддерживать.
- Хорошая производительность из-за использования статической типизации и сборки мусора.
- Большое и активное сообщество с множеством доступных библиотек и фреймворков. Обширный набор инструментов для разработки, отладки, профилирования и развертывания.
- Golang хорошо масштабируется. Язык был разработан для поддержки горизонтального масштабирования, что позволяет разработчикам легко добавлять новые узлы в свою распределенную систему без необходимости переписывать код, гарантируя, что приложения останутся отзывчивыми даже при высокой нагрузке.
- Кроссплатформенная поддержка. Код может быть скомпилирован для работы на различных операционных системах, таких как Windows, Linux и macOS.

Таким образом, Golang - идеальный выбор для создания микросервисов благодаря простому синтаксису, надежному набору функций, кроссплатформенной поддержке, скорости и масштабируемости. Его использование в распределенных системах сделало его популярным выбором среди разработчиков, стремящихся быстро и эффективно создавать приложения и сервисы.

2.3 Формат обмена данными. Тип API

API – это механизмы, которые позволяют двум программным компонентам взаимодействовать друг с другом, используя набор определений и протоколов. Каждый тип API имеет свои особенности и подходит для конкретных приложений [14-15].

- **REST** - наиболее часто используемый архитектурный стиль для создания API, который имеет универсальную поддержку сторонних инструментов. Клиент и сервер полностью отделены друг от друга, что позволяет использовать уровни абстракции, которые помогают поддерживать гибкость даже по мере роста и развития системы. REST удобен для кэширования и поддерживает несколько форматов (простой текст, HTML, XML, YAML и JSON), что важно при создании общедоступных API.
- **WebSocket** - обеспечивает двустороннюю связь между клиентом и сервером. Это популярный выбор для приложений, обрабатывающих данные в реальном времени. Он отлично подходит для написания real-time чата.
- **SOAP** - протокол обмена структурированными сообщениями в распределённой вычислительной среде. Сообщения SOAP представляют собой XML-документы, которые часто бывают излишне громоздкими, особенно для простых систем. За последние годы стал менее популярным.
- **GraphQL** (Facebook) - очень гибок по отношению к своим клиентам, поскольку позволяет извлекать и доставлять только запрошенные данные. Но в простых приложениях может добавить ненужную сложность по сравнению с REST. Кроме того, GraphQL не использует методы кэширования HTTP, которые позволяют сохранять содержимое запроса и уменьшать объем трафика на сервер.
- **gRPC** (Google) - основан на клиент-серверной модели удаленных вызовов процедур. Клиентское приложение может напрямую вызывать методы серверного приложения, как если бы оно было локальным объектом. Однако его формат данных не читается человеком, поэтому для анализа полезной нагрузки и выполнения отладки требуются дополнительные инструменты. Кроме того, HTTP/2 поддерживается только через TLS в последних версиях современных браузеров.

Наш вариант использования не требует реализации gRPC или GraphQL, и рисковать их внедрением на зарождающемся этапе (до повсеместного внедрения) нецелесообразно и не нужно. Поэтому в своем проекте я решила использовать WebSocket для разработки real-time чата и REST для остальных сервисов, поскольку REST удобен, поддерживается повсеместно и покрывает все текущие потребности.

2.4 Система управления базами данных

PostgreSQL [16] - свободная объектно-реляционная система управления базами данных. Представляет собой набор элементов данных с предопределенными отношениями между ними, работать с которыми можно с помощью запросов на языке SQL. Элементы организованы в виде набора таблиц со столбцами и строками. Данная объектно-реляционная система является очень популярной, так как она бесплатная и простая в управлении.

MongoDB [17] - наиболее популярная на данный момент документо-ориентированная система управления базами данных. Она опирается на концепции коллекций и документов. В отличие от реляционных СУБД, MongoDB не требуются таблицы, схемы или отдельный язык запросов, что упрощает изменение структуры данных. Плюсом также является возможность горизонтального масштабирования.

Несмотря на рост популярности баз данных NoSQL, реляционные базы данных по-прежнему остаются предпочтительными для многих приложений. Это связано с их сильными способностями к запросам и их надежностью. Они лучше всего подходят данных, структура которых не часто меняется. Нас это вполне встраивает, поэтому выбираем PostgreSQL.

2.5 Оркестратор контейнеров

Сегодня Docker Swarm и Kubernetes — самые популярные платформы для оркестрации контейнеров. Оба они имеют свое специфическое использование и имеют определенные преимущества и недостатки [18].

Docker Swarm - стандартный оркестратор для docker контейнеров, доступный из «коробки», если у вас установлен сам docker. Плюсом является простота установки и поддержки, но при этом он работает только с Docker-контейнерами и совсем не поддерживает автоматическое масштабирование.

Kubernetes - открытое программное обеспечение для автоматизации развёртывания, масштабирования и управления контейнеризированными приложениями. Плюсами является то, что он способен поддерживать и управлять большими и сложными рабочими нагрузками, а также автоматизирован и поддерживает автоматическое масштабирование. Из минусов - процесс установки может быть сложен, особенно для новичков.

Мы будем использовать Kubernetes, поскольку его автоматическое масштабирование позволяет увеличивать или уменьшать масштаб для более быстрого удовлетворения спроса.

3 Разработка web-приложения

3.1 Описание функциональных и нефункциональных требований

Функциональные требования:

- Возможность регистрации и авторизации на сайте, создание/обновление/удаление своих постов или комментариев, общение в real-time чате.
- Сохранение всех пользователей, постов и комментариев в базу данных.
- Сайт должен быть реализован в виде веб-приложения, запускающегося в браузере. На странице будут представлены такие элементы, как страницы авторизации и регистрации, страница с чатом и страница с блогом.
- Сервис должен быть удобным в использовании.

Нефункциональные требования:

- Ответ должен быть дан в течение периода времени до 1 секунды.
- API должен стабильно работать на операционных системах MacOs, Linux, Windows.
- Пользователю, работающему с программой через веб-браузер должен быть предоставлен непрерывный доступ к веб-приложению, расположенному по определённому URL-адресу. Веб-сервис не должен непредвиденно прерывать свою работу.
- Система должна быть простой в использовании и понятной.
- Компоненты должны быть масштабируемыми.
- Система должна быть защищена от несанкционированного доступа.
- Для простоты внесения изменений компоненты должны быть настроены с использованием переменных среды.

3.2 Архитектура проекта

В проекте используется классическая трехзвенная архитектура - модульная клиент-серверная архитектура, которая состоит из трех уровней [19-20]:

- **Уровень представления** - самый верхний уровень приложения с интерфейсом пользователя, который связывается с другими уровнями посредством вызовов интерфейса прикладных программ (API). Главной функцией является представление задач и результатов, понятных пользователю.
- **Уровень приложения** - является логическим уровнем, управляет основными функциями приложения: обрабатывает команды, выполняет вычисления.
- **Уровень данных** - размещает серверы баз данных (локально или в облаке), где информация хранится и извлекается. Данные на этом уровне хранятся независимо от серверов приложений или бизнес-логики и управляются и доступны с помощью таких программ, как PostgreSQL, MongoDB, Oracle, MySQL и др.

Эти три уровня являются логическими, а не физическими, и могут работать как на одном физическом сервере, так и на разных машинах. Теперь опишем архитектуру нашего приложения по нотации C4 (посмотрим на две основные диаграммы) [21]:

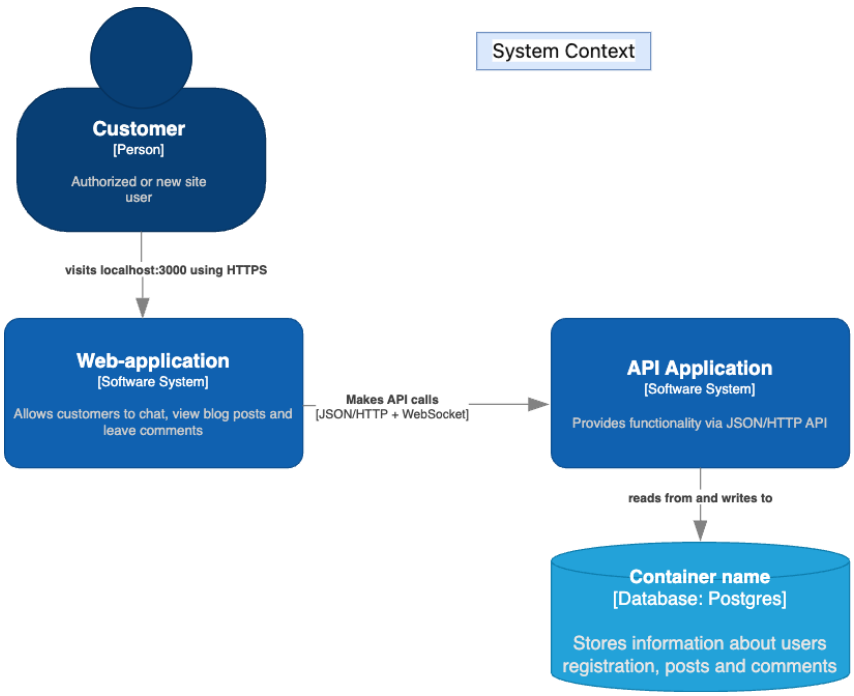

Рис. 3.1: Контекстная диаграмма - позволяет кратко и ёмко визуализировать назначение и границы системы, выявить и устранить коллективные расхождения в их понимании, показать и договориться о её масштабе. Контекстная диаграмма относится к категории диаграмм, описывающих систему на уровне «чёрного ящика» — а именно, только внешние свойства, но не содержание системы.

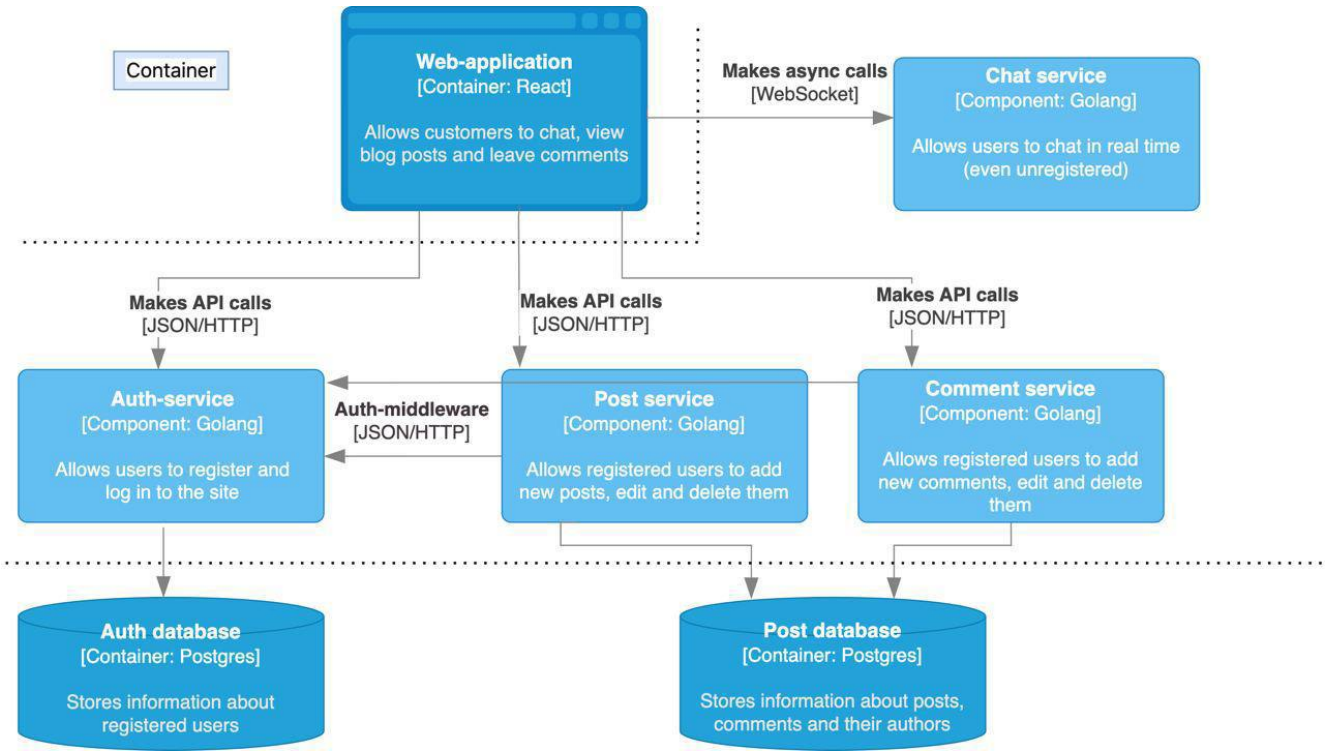

Рис. 3.2: Контейнерная диаграмма - содержит описание каждого сервиса и БД, а также способы их взаимодействия. Предназначена для пользователей, которым нужно понять архитектуру приложения без глубокого погружения в техническую часть. Это наиболее полезная и обязательная абстракция при документировании системы.

Помимо этого, стоит отметить, что в ходе проекта использовались принципы чистой архитектуры, основным из которых является разделение деталей реализации слоями абстракции. Конкретно код был разбит на сущности (models), сценарии (Usecases/actions) и интерфейсы-адапторы (Interface Adapters), что делает его удобным в обслуживании, наглядным и простым для понимания, а также легко тестируемым [22]:

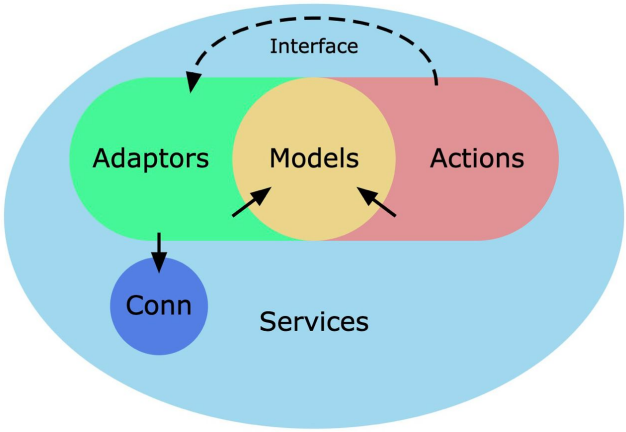

Рис. 3.3: Чистая архитектура на Go [23].

3.3 Написание связки микросервисов

Одной из ключевых задач было написать связку микросервисов с frontend частью, которая бы взаимодействовала с ними через REST API или веб-сокеты согласно описанной выше архитектуре. Отметим особенности каждого микросервиса:

Auth-service отвечает за регистрацию и аутентификацию пользователей.

- Помимо access JWT-токенов, он использует также и refresh токены, что повышает безопасность приложения. Токены подписываются секретным ключом и передаются клиенту, который в дальнейшем использует их для подтверждения подлинности аккаунта.
- Всех зарегистрированных пользователей храним в отдельной базе данных, с указанием их email-почты, логина и зашифрованного пароля.
- Также присваиваем новому пользователю роль, по умолчанию member, но при старте приложения в базе есть также модератор и админ с расширенными правами.

Post-service отвечает за публикацию постов.

- Видеть все посты может каждый посетитель сайта, но публиковать/редактировать/удалять свои посты – только зарегистрированный пользователь.
- Зарегистрирован пользователь или нет мы понимаем с помощью auth-middleware: клиент обращается к auth-service, передавая токены через cookies, там их проверяют на валидность, и если все хорошо и такой пользователь есть в системе, возвращают информацию о нем. После этого post-service проверяет, есть ли у нас права на запрашиваемое действие.
- Удалять и редактировать чужие записи могут только участники с ролью admin или moderator.
- Все посты хранятся в отдельной базе данных (вместе с комментариями к ним).

Comment-service разделяет одну БД с вышеупомянутым микросервисом и работает аналогичным образом, только оперирует комментариями вместо постов.

Chat-service - реализует real-time чат.

- Единственный сервис, работающий через веб-сокеты в реальном времени.
- Имена задаются случайным образом при подключении пользователя к чату.
- Есть разумные ограничения по длине одного сообщения и количестве сообщений в день.

Frontend-service – реализует пользовательский интерфейс.

- Написан на React (JavaScript-библиотеки) с использованием HTML, CSS.
- При нажатии на разные кнопки отправляются запросы к backend-сервисам и в случае успешного их выполнения меняется состояние страницы в зависимости от запрашиваемого действия (добавляет пост, и тд).
- При получении ошибки от сервиса отправляет pop-up оповещение с комментарием. Например, что пользователь с таким email уже зарегистрирован, введен неверный пароль и прочее.

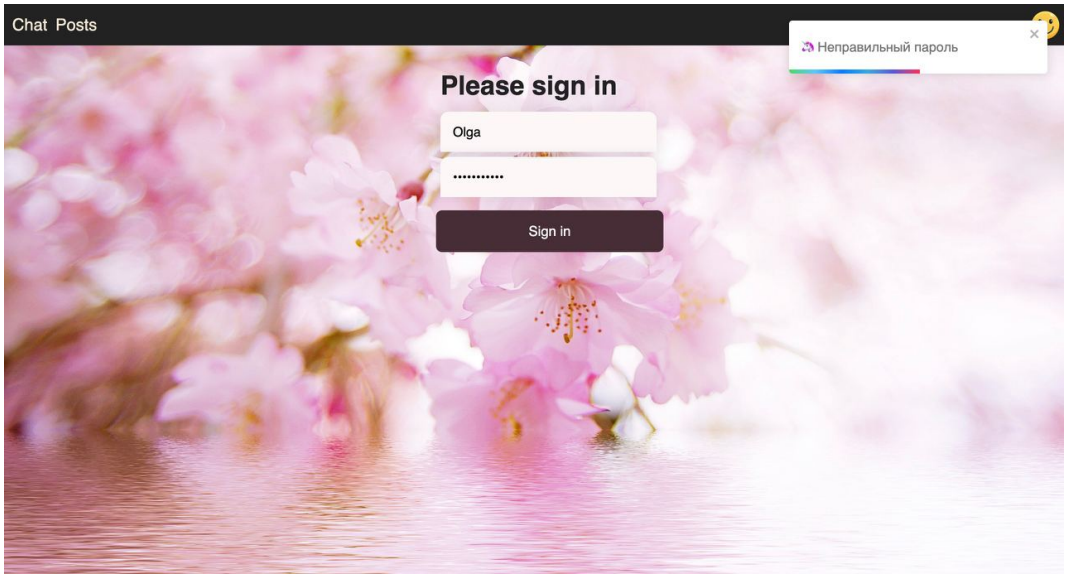

Рис. 3.4: Пример. Оформление login-страницы в Frontend-service. Больше фото по ссылке

На данном этапе запускать полученную связку микросервисов можно было с помощью docker-compose файла. Также стоит отметить, что все полученные образы были загружены в **GitLab Container Registry** - безопасный и закрытый реестр образов Docker. Он интегрирован с конвейерами GitLab CI/CD и предоставляет удобный способ отправки и получения изображений. В следующей главе мы, используя данные образы, научимся разворачивать приложение в Kubernetes.

4 Развертывание приложения в Kubernetes

4.1 Подготовительный этап

1. Контейнеризация приложения.

К каждому микросервису я создала Dockerfile с использованием **двухэтапной сборки**. Как известно, чтобы приложение на Go заработало, нам нужно его скомпилировать. При компиляции он создаст исполняемый файл (относящийся к этой ОС), и только этот исполняемый файл требуется для запуска приложения. Поэтому можно разделить процесс на две стадии.

Для среды этапа сборки я использовала образ Golang, который поставляется со всеми инструментами, необходимыми для запуска, тестирования и сборки приложения. Полученный исполняемый файл передала в среду выполнения, которая его запустит. Поскольку нам уже не нужны никакие инструменты Go, мы можем работать с базовым образом alpine. Итоговый образ составил около 20MB.

Чтобы проиллюстрировать преимущества многоэтапной сборки, я также создала одноэтапный Dockerfile, размер которого равнялся 1.09GB.

Name	Tag	Status	Created ↓	Size	Actions
test-one-stage/auth 5c23a163fa47	latest	Unused	6 minutes ago	1.09 GB	
test-two-stage/auth bd08321a900d	latest	Unused	2 days ago	20.11 MB	

Рис. 4.1: Сравнение одноэтапной и двухэтапной сборки в Docker

В итоге, многоэтапная сборка позволила сократить общий размер образа докера примерно в 55 раз. Теперь его легче переносить и можно легко развернуть в рабочей среде.

2. Настроить кластер.

Для проекта нам были выделены следующие ресурсы с уже настроенными кластерами:

Кластер Kubernetes:

- CPU: 6x
- RAM: 18 GB
- Накопитель: 220 Gb SSD (storageClass: mts-ssd-fast) 500 IOPS
- Версия Kubernetes: 1.24.6

Кластер PostgreSQL:

- CPU: 4 GB
- RAM: 8 GB
- SSD: 30 GB
- PSQL v15

4.2 Написание k8s-манифестов

После изучения основ Kubernetes [24] и выполнения подготовительного этапа, мне нужно было с помощью манифестов определить желаемое состояние объектов приложения, которое Kubernetes будет поддерживать при применении манифеста.

Манифест Kubernetes - это ваш личный гид по кластеру Kubernetes: файл конфигурации, написанный в формате YAML или JSON, который описывает ресурсы, которые вы хотите использовать в своем кластере. Этими ресурсами может быть множество вещей: pods (которые запускают ваши приложения), services (которые помогают вашим приложениям взаимодействовать) и deployments (которые управляют вашими приложениями). Вы также можете думать о файле манифеста как о списке пожеланий Kubernetes! Он сообщает, какие ресурсы вам нужны и как вы хотите их настроить.

Для каждого микросервиса я создала необходимые ему виды манифестов:

- **Pods** - наименьшие развертываемые вычислительные единицы, которые вы можете создавать и которыми можете управлять в Kubernetes. По сути, это обертка над группой из одного или нескольких контейнеров с общим хранилищем и сетевыми ресурсами, а также спецификацией того, как запускать контейнеры.
- **Service** - абстракция, помогающая предоставлять доступ к группам Pod по сети.
- **ConfigMap** - объект API, используемый для хранения неконфиденциальных данных в парах ключ-значение. Поды могут использовать ConfigMaps как переменные среды, аргументы командной строки или как файлы конфигурации в томе.
- **Secret** - объект, похожий на ConfigMaps, но содержащий конфиденциальные данные, таких как пароль, токен или ключ.

Также, требовалось создать **Ingress** - объект API, предоставляющий правила маршрутизации для управления доступом внешних пользователей к службам в кластере Kubernetes, обычно через HTTPS/HTTP. С помощью Ingress можно легко настроить правила маршрутизации трафика, не создавая кучу балансировщиков нагрузки и не раскрывая каждую службу на узле.

Помимо этого, чтобы получить образы из приватного реестра (в моем случае из Gitlab Container Registry), нужно было предоставить Kubernetes необходимые секреты для аутентификации в этом реестре. Для этого достаточно было создать Deploy Tokens и выполнить небольшую настройку согласно инструкции.

4.3 Шаблонизация манифестов с помощью Helm

Что такое хелм-чарты? Helm Charts - это инструмент, помогающий определять, устанавливать и обновлять приложения, работающие в Kubernetes. По сути, Helm - это механизм шаблонов, который создает манифесты Kubernetes. Как и манифесты Helm Charts написаны на YAML.

Для чего использовать Helm-chart?

Helm объединяет все связанные манифесты (такие как deployments, services, statefulset и т. д.) в диаграмму. Helm-chart становятся особенно полезными, когда у вас большое и сложное приложение, содержащее десятки объектов Kubernetes, которые необходимо настроить и изменить во время обновления. Это также применимо, если вы развертываете одно и то же приложение несколько раз. Helm может сделать этот процесс простым и воспроизводимым.

Helm объединяет шаблоны и значения по умолчанию на диаграмме с предоставленными вами значениями, а также с информацией из вашего кластера для развертывания и обновления приложений. Вы можете использовать диаграммы непосредственно из репозиториев, скачанных вами диаграмм или диаграмм, которые вы создали сами. Helm использует механизм шаблонов Go, поэтому, если вы знакомы с ним, вы поймете, как работают диаграммы.

Я разработала Helm-chart к проекту, когда работала с **minikube** [25] - инструментом для локальной разработки приложений Kubernetes, который поддерживает все подходящие функции Kubernetes.

5 Настройка Gitlab CI/CD

GitLab - репозиторий с открытым исходным кодом и платформа непрерывной интеграции/непрерывной доставки (CI/CD). Есть два основных требования для использования GitLab CI/CD: код приложения, размещенный в репозитории Git, и файл с именем .gitlab-ci.yml в корне репозитория, в котором указана конфигурация GitLab.

Файл .gitlab-ci.yml определяет сценарии, которые должны выполняться во время конвейера CI/CD, и их планирование, дополнительные файлы конфигурации и шаблоны, зависимости, кэши, команды, которые GitLab должен выполнять последовательно или параллельно, а также инструкции о том, где приложение должно быть развернуто. Я разделила сценарий на две стадии:

- **build**:
  - Из тегов к коммитам (которые мы автоматически делаем ниже) получаем последнюю версию приложения. В случае ее отсутствия устанавливаем начальную с значением v0.1.0.
  - Поднимаем ее, увеличивая значение Patch
  - Выполняем вход в Container Registry с помощью команды docker login
  - Собираем образ и с нужным тегом-версией сохраняем в Container Registry
- **deploy**:
  - Проверяем, что кластер доступен
  - Клонируем репозиторий с манифестами
  - Выполняем развёртывание, пробрасывая env переменные, где нужно

В результате, при получении каких-либо изменений в коде у нас в кластере Kubernetes автоматически поднимется новая версия приложения на основе нового, взятого из Container Registry образа с увеличенным тегом-версией.

6 Заключение

6.1 Выводы и направления дальнейшей работы

В ходе работы мы поняли, что DevOps-практики являются важной составляющей успешной разработки ПО. С современными веб-службами пользователи ожидают, что приложения будут доступны круглосуточно и без выходных, а разработчики хотят разворачивать новые версии приложений несколько раз в день. Использование DevOps позволяет сократить время релиза, улучшить качество продукта и повысить скорость разработки за счет автоматизированных процессов.

Именно поэтому сегодня важно изучать данную методологию и стараться применять ее принципы при разработке новых продуктов. Для облегчения внедрения DevOps-практик в новые проекты, я разработала рабочий макет web-приложения, в котором предусмотрены базовые сценарии его релиза и эксплуатации. Данный шаблон может облегчить работу над первой версией нового, более масштабного продукта, поскольку его легко можно адаптировать под себя.

Есть несколько **направлений дальнейшей работы** по улучшению проекта:

- Заменить существующее автоматическое развертывание через k8s-манифесты на автодеплой Helm-чартов
- Придумать более грамотный способ поднятия версии образа
- Изучить клиентские библиотеки Prometheus, ознакомиться с виджетами в Grafana и реализовать сбор и хранение метрик с микросервисов.

Источники

[1] DevOps. [Электронный ресурс] / ru.wikipedia.org. Режим доступа: свободный. (Дата обращения: 10.01.23)

[2] DevOps: как автоматизировать производство цифровых продуктов. [Электронный ресурс] / pro.rbc.ru. Режим доступа: свободный. (Дата обращения: 10.01.23)

[3] DevOps в SimpleOne. [Электронный ресурс] / simpleone.ru. Режим доступа: свободный. (Дата обращения: 10.01.23)

[4] Top 12 DevOps Trends for 2023: What to Expect. [Электронный ресурс] / alpacked.io. Режим доступа: свободный. (Дата обращения: 11.01.23)

[5] The DevOps adoption: a clear view on the state of affairs. [Электронный ресурс] / hackernoon.com. Режим доступа: свободный (с VPN). (Дата обращения: 11.01.23)

[6] Что такое DevOps? [Электронный ресурс] / aws.amazon.com. Режим доступа: свободный. (Дата обращения: 11.01.23)

[7] What is DevOps? [Электронный ресурс] / www.netapp.com. Режим доступа: свободный. (Дата обращения: 11.01.23)

[8] Все говорят о микросервисной архитектуре приложений. Чем она хороша и как на нее перейти? [Электронный ресурс] / www.tadviser.ru. Режим доступа: свободный. (Дата обращения: 07.02.23)

[9] Сравнение микросервисной и монолитной архитектур. [Электронный ресурс] / www.atlassian.com. Режим доступа: свободный. (Дата обращения: 07.02.23)

[10] Microservices In Review [Golang]. [Электронный ресурс] / medium.com. Режим доступа: свободный. (Дата обращения: 08.02.23)

[11] These giant tech companies has embraced the microservices architecture. [Электронный ресурс] / goldenowl.asia. Режим доступа: свободный. (Дата обращения: 09.02.23)

[12] Go vs Python in 2023: Which One to Choose? [Электронный ресурс] / doit.software. Режим доступа: свободный. (Дата обращения: 09.12.22)

[13] Is Golang the Go-To for 2023? [Электронный ресурс] / www.bairesdev.com. Режим доступа: свободный. (Дата обращения: 09.12.22)

[14] REST vs. GraphQL vs. gRPC – Which API to Choose? [Электронный ресурс] / www.baeldung.com. Режим доступа: свободный. (Дата обращения: 12.12.22)

[15] A Guide to the Most Popular APIs: REST, SOAP, GraphQL, gRPC, and WebSockets [Электронный ресурс] / blog.postman.com. Режим доступа: свободный. (Дата обращения: 12.12.22)

[16] PostgreSQL [Электронный ресурс] / ru.wikipedia.org. Режим доступа: свободный. (Дата обращения: 09.02.23)

[17] MongoDB [Электронный ресурс] / itglobal.com. Режим доступа: свободный. (Дата обращения: 09.02.23)

[18] Kubernetes VS Docker Swarm – What is the Difference? [Электронный ресурс] / www.freecodecamp.org. Режим доступа: свободный. (Дата обращения: 15.04.23)

[19] Трёхуровневая архитектура [Электронный ресурс] / ru.wikipedia.org. Режим доступа: свободный. (Дата обращения: 10.05.23)

[20] Трехуровневая архитектура приложения [Электронный ресурс] / it-matika.pro. Режим доступа: свободный. (Дата обращения: 10.05.23)

[21] Описание архитектуры системы с помощью C4 model [Электронный ресурс] / bool.dev. Режим доступа: свободный. (Дата обращения: 09.02.23)

[22] Clean Architecture with GO [Электронный ресурс] / manakuro.medium.com. Режим доступа: свободный. (Дата обращения: 08.04.23)

[23] Чистая архитектура на Go [Электронный ресурс] / vporoshok.me. Режим доступа: свободный. (Дата обращения: 08.04.23)

[24] Complete Kubernetes Tutorial for Beginners. [Электронный ресурс] / www.youtube.com. Режим доступа: свободный. (Дата обращения: 14.01.23)

[25] Minikube Documentation. [Электронный ресурс] / minikube.sigs.k8s.io. Режим доступа: свободный. (Дата обращения: 14.01.23)