1 Основы Kubernetes

Приветствуем вас на страницах книги «Kubernetes. Полное руководство»! Мы рады сопровождать вас в процессе освоения этой системы. Если вы занимаетесь разработкой программного обеспечения, то, вероятно, уже слышали о Kubernetes. Это неудивительно, так как за последние годы популярность этой системы значительно возросла.

Созданный компанией Google, Kubernetes представляет собой самый популярный и широко используемый инструмент для оркестрации контейнеров. Если вам нужна надежная система управления контейнеризованными приложениями в рабочей среде, будь то локальная инфраструктура или публичное облако, Kubernetes станет идеальным решением. Ключевым моментом здесь является масштабирование. Развертывание и управление контейнерами в больших масштабах — это весьма сложная задача, поскольку контейнерные движки (такие как Docker) не предусматривают встроенных механизмов поддержания доступности и масштабируемости контейнеров[1].

Система Kubernetes изначально разрабатывалась компанией Google для развертывания огромного количества контейнеров в ее распределенной инфраструктуре. Включив Kubernetes в свой технологический стек, вы получите открытую платформу, созданную одним из интернет-гигантов, предъявляющим строжайшие требования к стабильности.

Хотя система Kubernetes поддерживает множество сред выполнения контейнеров, в этой книге основное внимание уделяется ее работе с Docker и Podman.

Возможно, вы уже активно используете Docker в своей работе, но мир оркестрации контейнеров вам пока незнаком. Вы даже можете не видеть смысла в применении таких инструментов, если Docker и так позволяет вам справляться

---

[1] На самом деле проверка доступности и масштабируемость реализованы в Docker (режим swarm) и его альтернативе — Podman (который также позволяет запускать поды — несколько контейнеров в общем пространстве имен), однако их гибкость и функционал, несомненно, сильно уступают возможностям Kubernetes. — *Здесь и далее примеч. науч. ред.*

со своими задачами. Именно поэтому в первой главе мы не будем углубляться в детали Kubernetes. Вместо этого разберем, что собой представляет эта система и как она помогает управлять контейнерами в рабочей среде. Вам будет проще освоить эту новую технологию, если вы будете понимать, зачем она была создана.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter01.

Монолиты и микросервисы

Давайте на мгновение отвлечемся от Kubernetes и Docker и немного поговорим о том, как развивались Интернет и сфера разработки программного обеспечения на протяжении последних 20 лет. Это поможет лучше понять, какое место занимает система Kubernetes и какие задачи она решает.

Стремительное развитие Интернета, начавшееся в конце 1990-х годов

В конце 1990-х годов популярность Интернета начала быстро расти. Если в 1990-е и даже начале 2000-х годов Интернетом пользовались всего несколько сотен тысяч человек во всем мире, то сегодня почти 2 миллиарда людей регулярно выходят в Интернет, чтобы отправлять электронную почту, просматривать веб-страницы, играть в видеоигры и решать другие задачи. Их потребности удовлетворяются десятками приложений, развернутых на множестве устройств.

При этом количество подключенных устройств также существенно выросло: теперь у одного человека может быть несколько разных гаджетов с доступом в Сеть, например ноутбук, настольный компьютер, смартфон, телевизор, планшет и многое другое.

Сегодня Интернет можно использовать для работы, развлечения, совершения покупок, чтения и не только. Он проник почти во все сферы общественной жизни и за последние 20 лет привел к глубочайшему изменению парадигмы. Все это сделало разработку программного обеспечения как никогда важной.

Потребность в более частых релизах программного обеспечения

Чтобы справиться с постоянно растущим числом пользователей, которые требуют все больше в плане функциональности, индустрия разработки ПО была вынуждена измениться, увеличив скорость и частоту выпуска новых версий.

В 1990-е годы можно было создать приложение, развернуть его в рабочей среде и обновлять всего один-два раза в год. Сегодня компании вынуждены выпускать обновления до нескольких раз в день, например, чтобы добавить новую функцию, обеспечить интеграцию с платформой социальной сети, реализовать поддержку разрешения экрана нового смартфона или выпустить патч для устранения уязвимости, обнаруженной накануне. Поскольку все стало гораздо сложнее, скорость разработки должна быть существенно выше, чем раньше.

Потребность в постоянном обновлении программного обеспечения стала нормой, и теперь выживание компаний во многом зависит от частоты релизов. Но как ускорить жизненный цикл разработки, чтобы чаще доставлять новые версии ПО нашим пользователям?

ИТ-отделы компаний были вынуждены эволюционировать как в организационном, так и в техническом плане. Изменение подхода к управлению проектами и командами позволило переключиться на гибкие методологии разработки, а широкое распространение таких технологий, как облачные платформы, контейнеры и виртуализация, помогло согласовать техническую гибкость с организационной. Все это было направлено на повышение частоты выпусков программного обеспечения. Давайте подробнее рассмотрим, как происходила эта эволюция.

Переход к гибким методологиям разработки

В настоящее время стандартным способом организации работы ИТ-команд считаются такие гибкие (agile) методологии, как Scrum, Kanban и DevOps.

Типичный ИТ-отдел, не использующий гибкие подходы, часто состоит из трех отдельных команд, каждая из которых отвечает за один этап жизненного цикла разработки и выпуска продукта.

Хотя сейчас мы говорим о гибких методологиях и истории Интернета, эта книга на самом деле посвящена системе Kubernetes! Просто перед тем, как перейти к ее обсуждению, нам следует объяснить кое-какие проблемы, с которыми сталкивалась индустрия до ее появления.

До внедрения гибких методологий специалисты, отвечавшие за разработку и эксплуатацию ПО, часто существовали в изоляции друг от друга, что приводило к неэффективности и сбоям в коммуникации. Гибкие методологии помогли устранить эти разрывы и наладить сотрудничество. Ниже перечислены три команды, которые раньше работали независимо друг от друга (рис. 1.1).

- *Бизнес-команда* представляет интересы клиента. Ее задача — объяснить, какие функции нужно включить в приложение, чтобы удовлетворить запросы пользователей. Эта команда переводит бизнес-цели в понятные разработчикам инструкции.
- *Команда разработки* состоит из инженеров, которые воплощают приложение в жизнь. Они превращают запросы бизнес-команды в код, создавая функциональность, с которой предстоит взаимодействовать пользователям. Здесь особенно важна четкая коммуникация: расплывчатые инструкции чреваты недопониманием, задержками и тратой лишних ресурсов.
- *Операционная команда (команда эксплуатации)* состоит из хранителей серверов, чья главная задача заключается в поддержании стабильной работы приложения. Добавление новых функций может нарушить его работу, так как требует обновлений, которые несут с собой определенные риски. В прошлом эти специалисты могли даже не знать о готовящихся изменениях, потому что их не привлекали к планированию.

Рис. 1.1. Изолированные команды в типичном ИТ-отделе

Роли в такой структуре четко определены, сотрудники из разных команд взаимодействуют не слишком активно, а при возникновении проблем теряют время, пытаясь получить нужную информацию от нужного человека.

Такая изоляция:

- значительно увеличивает время разработки;
- повышает риск того, что новая версия при развертывании в рабочей среде может не заработать вовсе.

Гибкие методологии и DevOps были призваны решить именно эти проблемы. Их внедрение побудило людей к совместной работе и формированию междисциплинарных команд.

DevOps — это культура сотрудничества и набор практик, направленных на улучшение взаимодействия между командой разработки (development, Dev) и операционной командой (operations, Ops). DevOps предполагает совместную работу и автоматизацию на всех этапах жизненного цикла программного обеспечения — от разработки и тестирования до развертывания и сопровождения.

В состав agile-команды входит владелец продукта, который описывает конкретные функции в виде пользовательских историй, понятных разработчикам, сотрудничающих с ним. Разработчики должны иметь представление о рабочей среде и возможность развертывать в ней приложения, предпочтительно с использованием подхода *непрерывной интеграции и непрерывного развертывания* (continuous integration and continuous deployment, CI/CD). Тестировщики также должны входить в состав таких команд, чтобы иметь возможность писать тесты.

Благодаря совместной работе над проектом, проиллюстрированной на рис. 1.2, команды получают более полное и точное представление о нем.

Рис. 1.2. Совместная работа над проектом

Таким образом, внедрение гибких методологий и DevOps привело к появлению междисциплинарных команд, способных формализовать потребность пользователей, реализовать нужную функцию, протестировать ее, выпустить и сопровождать в рабочей среде. В табл. 1.1 сравнивается традиционный подход к разработке ПО с гибкими методологиями и DevOps.

Таблица 1.1. Сравнение традиционного подхода к разработке ПО с гибкими методологиями DevOps

Характеристика	Традиционный подход к разработке ПО	Гибкие методологии и DevOps
Структура команды	Изолированные отделы (разработка, эксплуатация)	Кросс-функциональные, междисциплинарные команды
Стиль работы	Независимые рабочие процессы, ограниченное взаимодействие	Совместная работа, итеративный подход к разработке
Ответственность	Разработчики передают готовое решение операционной команде для развертывания и сопровождения	Принцип «Кто создает, тот и запускает». Вся команда отвечает за весь жизненный цикл продукта
Фокус	Функциональные возможности	Бизнес-ценность, постоянное улучшение
Релизный цикл	Длительные релизные циклы, редкие развертывания	Короткие спринты, частые релизы со сбором обратной связи
Тестирование	Отдельная фаза тестирования после разработки	Интегрированное тестирование на протяжении всего цикла разработки
Инфраструктура	Статическая инфраструктура, управляемая вручную	Автоматизированное предоставление и управление инфраструктурой (DevOps)

Итак, мы рассмотрели организационные изменения, вызванные внедрением гибких методологий. Теперь обсудим техническую эволюцию, которая произошла за последние несколько лет.

Переход от локальной инфраструктуры к облаку

Наличие agile-команд — это замечательно, но гибкость должна распространяться не только на организацию работы, но и на способы создания и размещения программного обеспечения.

Чтобы увеличить частоту релизов, agile-команды были вынуждены пересмотреть такие важные аспекты процесса разработки и выпуска ПО, как:

- размещение приложений;
- архитектура программного обеспечения.

Современные приложения рассчитаны не на сотни, а на миллионы одновременных подключений. Рост количества пользователей потребовал соответствующего увеличения вычислительных мощностей. В результате организация хостинга приложений превратилась в серьезную технологическую задачу.

На заре развития веб-хостинга компании использовали два основных подхода к размещению своих приложений, одним из которых был локальный хостинг (on-premises hosting). Этот способ подразумевал физическое владение и управление серверами, на которых работали приложения. Для организации локального хостинга могли использоваться следующие средства.

1. *Выделенные серверы*. Этот вариант предполагал аренду физических серверов у специализированных провайдеров. При этом провайдер обеспечивал работу физической инфраструктуры (электропитание, охлаждение, безопасность), а ответственность за настройку серверов, установку ПО и текущее обслуживание брала на себя сама компания. Такой подход давал больше контроля и возможностей для настройки по сравнению с виртуальным хостингом, но по-прежнему требовал наличия в штате компании квалифицированных ИТ-специалистов.
2. *Собственный дата-центр*. Такой подход предполагал строительство и обслуживание частного центра обработки данных. Этот вариант требовал колоссальных инвестиций в создание и эксплуатацию собственной физической инфраструктуры, включающей серверное и сетевое оборудование, хранилища данных, а также надежные системы электропитания, охлаждения и обеспечения безопасности. Такой подход предоставлял максимальный уровень контроля и безопасности, но являлся крайне затратным и ресурсоемким, поэтому обычно применялся только крупными корпорациями, располагающими серьезными ИТ-ресурсами.

Стоит отметить, что локальный хостинг также предполагает управление ОС, обновление системы безопасности, создание резервных копий и разработку планов восстановления после сбоев в работе серверов. В связи с этим компаниям, выбравшим данный подход, часто приходилось организовывать отдельную ИТ-команду для обслуживания локальной инфраструктуры и управления ею, что дополнительно увеличивало затраты.

По мере роста пользовательской базы компании были вынуждены наращивать вычислительные мощности, арендуя более производительные серверы или устанавливая новое оборудование в собственном дата-центре. Такой подход не отличается гибкостью. И сегодня многие компании все еще используют такие малогибкие локальные решения.

Переломным моментом стало появление публичного облака, служащего альтернативой локальной инфраструктуре. Такие крупные компании, как Amazon,

Google и Microsoft, владеющие множеством дата-центров, создали слой виртуализации поверх своей масштабной инфраструктуры, предоставив доступ к виртуальным машинам через API. Благодаря этому виртуальную машину можно получить с помощью всего нескольких щелчков кнопкой мыши или путем ввода пары команд в консоли.

В табл. 1.2 перечислены основные выгоды облачных вычислений для организаций.

Таблица 1.2. Важность облачных вычислений для организаций

Характеристика	Локальная инфраструктура	Облако
Масштабируемость	Ограниченная — для увеличения ресурсов требуется закупка нового оборудования	Высокая — ресурсы легко добавляются или удаляются по запросу
Гибкость	Низкая — для внесения изменений требуется модернизация физического оборудования	Высокая — ресурсы можно быстро выделить и освободить
Стоимость	Высокие первоначальные затраты на покупку оборудования, лицензионного ПО и формирование ИТ-отдела	Низкие первоначальные затраты — ресурсы оплачиваются по мере использования
Обслуживание	Для обслуживания и обновления системы требуется отдельная команда ИТ-специалистов	Минимальные затраты на обслуживание, так как инфраструктурой управляет облачный провайдер
Безопасность	Практически полный контроль над безопасностью, но для его осуществления требуются квалифицированные специалисты	Надежные меры безопасности, реализуемые облачным провайдером
Время простоя	Восстановление после аппаратных сбоев может занимать много времени	Облачные провайдеры обеспечивают высокую доступность и возможности для аварийного восстановления
Расположение	Привязка к физическому местоположению дата-центра	Доступ из любой точки мира при наличии интернет-соединения

В следующем подразделе вы узнаете о том, как облачные технологии помогли организациям масштабировать свою ИТ-инфраструктуру.

Почему облако хорошо подходит для масштабирования

Сегодня практически любой человек может нажать буквально несколько кнопок, чтобы получить сотни или даже тысячи серверов в виде виртуальных машин или экземпляров, созданных на базе физической инфраструктуры, обслуживаемой такими облачными провайдерами, как *Amazon Web Services*, *Google Cloud Platform* и *Microsoft Azure*. За последние годы многие компании перенесли свои рабочие нагрузки из локальной инфраструктуры в облако, и масштаб этого перехода был колоссальным.

Благодаря этому в настоящее время вычислительные мощности являются одним из самых легкодоступных ресурсов.

Облачный хостинг превратился в стандартное решение для размещения приложений, доступное agile-командам, поскольку облако идеально соответствует современным подходам к разработке ПО.

Конфигурации виртуальных машин, процессоры, операционные системы, правила обработки сетевого трафика и многое другое доступно для просмотра и полностью настраивается, поэтому ваша команда точно знает, как устроена рабочая среда. Программируемая природа облачных платформ позволяет легко воспроизвести рабочую среду в среде разработки или тестирования. Это повышает гибкость и помогает командам справляться с проблемами, возникающими в процессе создания программного обеспечения. Данное преимущество особенно важно для agile-команд, работающих в соответствии с принципами DevOps и отвечающих за разработку, выпуск и сопровождение приложений в рабочей среде.

Использование облачных сервисов дает такие преимущества, как:

- эластичность и масштабируемость;
- помощь в преодолении изоляции команд и внедрении гибких методологий;
- полная совместимость с гибкими методологиями и DevOps;
- низкая стоимость и гибкие модели оплаты;
- отсутствие необходимости управлять физическими серверами;
- возможность беспрепятственно удалять и заново создавать виртуальные машины;
- большая гибкость по сравнению с арендой физических серверов.

Благодаря этим преимуществам облако стало ценнейшим инструментом в арсенале agile-команд разработчиков. По сути, вы можете создавать и многократно воспроизводить рабочую среду, не занимаясь при этом обслуживанием

физического оборудования. Облако позволяет масштабировать приложение в зависимости от количества пользователей или объема потребляемых ими вычислительных ресурсов. Это делает приложение высокодоступным и отказоустойчивым, что в конечном счете улучшает пользовательский опыт.

Kubernetes можно запускать как в облаке, так и в локальной инфраструктуре. Эта система чрезвычайно универсальна, ее можно развернуть даже на одноплатном компьютере Raspberry Pi. Хотя Kubernetes и публичное облако прекрасно сочетаются друг с другом, вы не обязаны использовать этот инструмент именно в облаке.

Теперь, когда мы обсудили изменения, связанные с появлением облачных технологий, обратимся к архитектуре программного обеспечения, так как и в этой области за последние годы произошло немало перемен.

Монолитная архитектура

В прошлом большинство приложений создавалось в виде монолитов. Типичное монолитное приложение представляет собой единый процесс, один исполняемый файл или один пакет, как показано на рис. 1.3.

Рис. 1.3. Монолитное приложение представляет собой один большой компонент, содержащий все программное обеспечение

Этот единый компонент отвечает за реализацию всей бизнес-логики, предусмотренной в программном обеспечении. Монолиты могут быть хорошим выбором, если вы разрабатываете простое приложение, которое не требуется часто обновлять в рабочей среде. Почему? Потому что у монолитов есть один серьезный недостаток: если он становится нестабильным или по какой-либо причине выходит из строя, все приложение оказывается недоступным.

Монолитная архитектура позволяет заметно сократить время разработки ПО, и, пожалуй, это ее единственное преимущество. Однако у нее есть и множество недостатков. Вот лишь некоторые из них:

- неудачное развертывание в рабочей среде может вывести из строя все приложение;
- масштабирование становится сложной задачей, а при его сбое все экземпляры приложения могут оказаться недоступными;
- любой сбой в монолите способен привести к полной остановке работы приложения.

В 2010-х годах эти недостатки начали вызывать серьезные проблемы. С ростом частоты развертываний возникла необходимость в архитектуре, способной поддерживать частые релизы и короткие циклы обновлений, а также снижающей риск возникновения полной недоступности приложения. Именно поэтому была разработана микросервисная архитектура.

Микросервисная архитектура

Микросервисная архитектура предполагает создание программного обеспечения в виде набора независимых микроприложений. Каждое из них называется *микросервисом* и имеет собственную систему версионирования, жизненный цикл, окружение и зависимости, а при необходимости и отдельный цикл развертывания. Каждый микросервис отвечает за ограниченное число бизнес-правил, а все они в совокупности составляют полноценное приложение. По сути, микросервис можно рассматривать как самостоятельное полнофункциональное ПО со своим жизненным циклом и процессом версионирования.

Поскольку микросервисы реализуют лишь часть функциональных возможностей всего приложения, для использования их функций к ним необходимо предоставить доступ. Как минимум нужно обеспечить возможность получать данные от микросервиса, а иногда и передавать их ему. Для предоставления доступа к микросервисам можно использовать такие широко поддерживаемые протоколы, как HTTP или AMQP. Кроме того, важно обеспечить взаимодействие микросервисов между собой.