ПРОГРАММИРОВАНИЕ CLOUD NATIVE

МИКРОСЕРВИСЫ, DOCKER И KUBERNETES

ИВАН ПОРТЯНКИН

Программирование Cloud Native. Микросервисы, Docker и Kubernetes

Разработка приложений и микросервисов в стиле Cloud Native. Упаковка микросервисов в контейнеры Docker, развертывание микросервисов с помощью Kubernetes. Управление развернутыми сервисами и их взаимодействие.

Иван Портянкин

Эта книга предназначена для продажи на http://leanpub.com/cloud-k8s

Эта версия была опубликована на 2020-03-06

Это книга с Leanpub book. Leanpub позволяет авторам и издателям участвовать в так называемом Lean Publishing - процессе, при котором электронная книга становится доступна читателям ещё до её завершения. Это помогает собрать отзывы и пожелания для скорейшего улучшения книги. Мы призываем авторов публиковать свои работы как можно раньше и чаще, постепенно улучшая качество и объём материала. Тем более, что с нашими удобными инструментами этот процесс превращается в удовольствие.

Оглавление

Введение

Разработка программного обеспечения и сервисов для сети Интернет в глобальном масштабе стала как никогда доступна. Если только у вас и вашей команды есть интересная новая идея или необычное решение для уже известной проблемы, вся мощь вычислительных облаков Cloud и обеспечиваемый ими легкий доступ к прорывным технологиям, легкость и скорость запуска контейнеров, точная настройка и изоляция их деталей с помощью Docker, и оркестрация работающих в контейнерах микросервисов с помощью Kubernetes даст вам возможность выйти на мировой уровень так, как если бы вы с полной уверенностью показали идеально настроенное демо приложения на вашем ноутбуке.

В этой книге мы взглянем на все с высоты птичьего полета, проанализируем популярную концепцию приложений, созданных работать в облаке (Cloud Native), вспомним как появились технологии виртуализации и масштабирования, разберем что именно принесут нам контейнеры и микросервисы, и увидим, как настройка и оркестрация Kunernetes позволяет развернуть систему любой сложности в любом облаке и мгновенно приспособить ее к растущим нагрузкам, при этом сделав ее надежной и устойчивой к отказам.

Актуальность и глубина информации. Онлайн-документация

Тема книги и технологии, которые мы изучаем и пробуем в ней, являются одним из самых популярных и динамичных направлений программирования и разработки последних пары лет. Уровень изменений и их скорость очень высока, и то что было актуально и важно полгода назад, может уступить свое место новой разработке или сервису.

Поэтому мы не стараемся максимально глубоко изучить все инструменты которыми пользуемся в данной книге, особенно это касается библиотек и программных сервисов API, предоставляемых известными публичными облаками (такими как Amazon AWS, Google Cloud и Microsoft Azure). Основное это понять процесс, который применяется при разработке в облаках, эффективно использовать базовые и главные возможности контейнеров Docker, и перейти “на ты” с Kubernetes.

Мы не станем перепечатывать массу документации из Интернета, прежде всего с сайтов docker.io и kubernetes.io. Большие компании, Google, Amazon и другие создают целый штаб качественных технических писателей, сопровождающих важные продукты, особенно если дело касается их коммерческих предложений и связанных с ними технологий, прежде всего Kubernetes. Хорошая документация, примеры, онлайн-лаборатории - все это к вашим услугам, и чем лучше качество и скорость начала работы с облаком, тем быстрее и больше оно привлекает клиентов.

На мой взгляд, основное - понять суть происходящего, увидеть “лес за деревьями”, узнать про краткую историю и эволюцию платформ, явлений, экосистем технологий, которые мы стараемся изучить, прежде всего иногда понять даже то, нужно ли нам вообще идти в этом направлении. Именно это очень трудно сделать в разношерстном море ссылок, блогов и статей Интернета, именно это мы и попробуем сделать в книге, уложившись в небольшой размер, и сделав ее путешествием по Cloud Native.

Аудитория книги

Эта книга прежде всего для программистов, которые на данный момент работают в привычной, не обязательно связанной с облаком среде - к примеру, запускают сервисы или работают с базами данных на собственных серверах или виртуальных машинах AWS, разрабатывают стандартные приложения для операционных систем (с интерфейсами командной строки, классические графические приложения, или вспомогательные приложения), или в основном сконцентрированы на пользовательских интерфейсах web и мобильных приложений. Подразумевается что вы знаете один или несколько языков программирования и основы сетей и протокола HTTP, но не более того.

Книга описывает, как применить классические знания о программировании в новой среде облачных вычислений (Cloud), где вместо ваших собственных настоящих физических серверов или управляемых вручную “тяжелых” мощных виртуальных вычислительных машин есть лишь эфемерная среда единого кластера (cluster), в которой будут исполняться сервисы, но среда эта способна практически мгновенно масштабировать приложение и сервисы для доступа миллионов пользователей, обеспечивать практически неограниченные ресурсы для вычисления и хранения данных, и эффективно обновлять приложения, используя множество полезных инструментов и сервисов Kubernetes, Docker и открытого сообщества вокруг них. При этом количество настроек самого приложения, изменений в нем будет не так велико - большая часть работы уже будет сделана для нас, особенно когда это касается непрерывного управления и масштабирования приложения.

По большому счету, это же относится и к программистам сервисов или больших монолитных приложений, уже работающих на выделенных (dedicated) для этого серверах, виртуальных машинах публичных провайдеров облака, или же в собственных центрах вычислений, как часть частных облачных решений. Даже если большая часть всего, что касается контейнеров, микросервисов, архитектуры приложения как сервисов, уже вам отлично известно, то книга может быть полезна просто посмотреть на то, что дополнительно может предложить вам Kubernetes и общая концепция Cloud Native (концепция приложений, созданных для облака).

Программирование и архитектура. Концепция Cloud Native

О чем именно эта книга? Мы попытаемся узнать большое количество связанных технологий, которые прекрасно сочетаются в единое целое, когда приложение или просто набор сервисов начинает работать в облаке. В общем и целом связаны технологии, которые мы будем рассматривать, *концепцией разработки приложений, созданных и приспособленных для работы в современном облаке (cloud native applications).* Концепция эта больше высокого уровня, можно назвать это архитектурой или дизайном (будем считать, как и многие эксперты, что эти два слова означают для программистов примерно одно и то же). В общем случае приложение, созданное для облака, способно быстро запускаться на любом облаке с поддержкой основных технологий концепции (Docker и Kubernetes), и при правильном подходе быстро масштабироваться при пиковой нагрузке, непрерывно обслуживать пользователей даже во время обновлений и фатальных ошибок, и позволять программистам использовать любые нужные им технологии и инструменты.

На словах концепция звучит прекрасно, однако между архитектурой (дизайном) и непосредственной разработкой большая работа и множество деталей, в которых, как известно, заключается иногда самая большая неприятность. В книге будет некоторое количество намеренно очень простых примеров, главная цель которых состоит в том, чтобы за минимальное количество шагов и настройки соединить теорию и историю изучаемого вопроса с практикой - непосредственным использованием довольно сложного рабочего инструмента (особенно когда речь идет о Kubernetes). После этого, уже зная основные детали, поняв их смысл, можно строить на этом уже реальную, сложную функциональность. Цельных, больших приложений “промышленного” уровня в книге мы рассматривать не будем - но вы всегда сможете найти их на GitHub, конференциях KubeCon, и разнообразных блогах.

Масштабирование на практике

Одним из основных преимуществ концепции Cloud Native и оркестратора Kubernetes является возможность автоматически масштабировать ваше приложение при пиковых нагрузках, и масштабировать лишь те его части, которые в этом нуждаются, если приложение разработано как набор микросервисов. На маленьких примерах это понять непросто, и лучше всего, если у вас в процессе чтения появится идея приложения или сервиса, которому понадобится быть в облаке, использовать впечатляющую мощь облачных библиотек и быть наготове для масштабирования и доступа для миллионов пользователей. В этом случае вся информация книги будет идти постепенно и в идеальном порядке. Если вы при этом еще будете использовать новый для себя язык программирования и небольшое руководство по нему, процесс станет особенно приятным и полезным.

Русскоязычные термины

В данный момент компьютерные технологии практически полностью находятся в зоне английского языка, как, впрочем, практически весь глобальный Интернет. Ничего плохого в этом нет, единый язык, который достаточно просто начать изучать, помогает мгновенному распространению информации, участию в конференциях программистов и архитекторов со всех уголков планеты, полезности таких сайтов как StackOverflow, и многому другому.

Тем не менее, нет ничего более легкого для усвоения, чем структурированная и правильно поданная информация на родном языке, который просто “работает” на уровне глубокого подсознания, не требуя ни минуты задержки, впитываясь в память и сознание. Эта книга постарается максимально структурировать и постепенно ввести вас в мир облачных технологий на родном языке, и мы будет использовать максимально близкие по смыслу русскоязычные аналоги английских названий технологий. Правда, многие библиотеки, алгоритмы, компании, сайты, паттерны проектирования, процессы разработки настолько влились в нашу жизнь на английском языке, что чтение их названий на русском языке будет даже мешать. В этом случае мы будем использовать максимально нейтральное название и включать в скобках английское название, чтобы не перемешивать языки и шрифты, и не коверкать книгу и предложения не слишком звучным транслитом (Докер? Кубернетес? А может быть Кюбэрнэтис? “Нода” и “поды”? В любом случае подобное выглядит просто неприятно - мы будем писать английский термин и использовать наиболее верный перевод, без излишней смысловой нагрузки).

Пользовательские интерфейсы

В этой книге мы в основном рассматриваем технологии для облака, особенно подходящие для быстрой разработки, от идеи до минимального работающего продукта (minimal viable product, MVP), и его последующей миграции в полноценное, масштабируемое до глобального мирового сервиса приложение или системы. Пользовательский интерфейс для системы это отдельная тема, чрезвычайно динамичная, вызывающая всегда горячие споры и как все что свойственно вкусу и внешности, то, что вызывает восторг у одних, кажется совершенно ужасным другим. Конечно у приложения будет web интерфейс, приложения для iOS и Android, возможно консоль администратора в дополнение ко всем диагностическим инструментам что мы затронем в этой книге. Тема чрезвычайно обширная и бесконечная и изучения требует в отдельном разговоре. Будем считать что отличный интерфейс у нас есть, и уделять внимания в данной книге ему не будем.

Примеры на Go и Java

Go как нельзя лучше подходит для микросервисов, особенно в качестве очень коротких, простых и быстрых процедур обработки данных. Этот язык специально был создан простым по

синтаксису, в нем по сути не присутствует полноценная система объектно-ориентированного программирования, и он во всем приглашает к простому, короткому, быстрому коду.

Крайне полезно то, что Go компилируется в машинный код для целевой операционной системы, то есть работает так эффективно, как это возможно без специальных оптимизаций. Работая в облаке, где каждую минуту исполнения, мегабайт оперативной памяти, использованную энергию необходимо будет оплатить, любые промежуточные этапы, использующие ресурсы, могут быть чуть дороже, чем интерпретируемые языки, например Python, или языки на основе виртуальных машин Node.js и Java, даже если они быстро компилируются (just in time, JIT) в машинный код в процессе работы. Это "чуть", помноженное на масштаб и количество пользователей (что в конечном итоге основной показатель успеха вашей идеи, стратегии и качества реализации), может вылиться в порядочные траты. Тем более что некоторые сложные участки программ на основе виртуальных машин оптимизировать крайне тяжело и они будут исполняться в разы медленнее.

Конечно, компиляция и сборка в машинный код всегда представляет собой проблему переносимости на конкретную платформу - что именно вы ожидаете на сервере в облаке, конкретный тип Linux? А если облачного провайдера нужно будет поменять или вы решите перейти на частное облако или свой собственный кластер? Это было бы огромным препятствием, но благодаря границам контейнеров Docker вы можете практически полностью изолировать детали реализации, настройку системы и ее пакетов и библиотек, от непосредственной операционной системы серверов в облаке или кластере. По сути, контейнеры полностью избавляют вас от зависимости от деталей облака и операционных систем, лишь бы Docker или подобная система контейнеров поддерживалась на серверах, что сейчас просто подразумевается как само собой разумеющееся требование к провайдеру облачных услуг.

Язык и платформа Java чрезвычайно популярны для классических корпоративных приложений, практически идеально переносятся между платформами и операционными системами, а количество фреймворков для веб-приложений, сервисов и микросервисов поражает воображение. Начать разработку может быть не так быстро и эффектно как на Go, но если у вас изощренный и сложный домен приложения (то есть описание области, в которой ваше приложение будет действовать), мощь объектного подхода, продуманный дизайн и стабильность языка, хорошо известные шаблоны проектирования и практики работы, и сотни разнообразных инструментов Java могут быть как нельзя кстати.

Вдобавок, главные игроки экосистемы Java и платформ для этого языка крайне заинтересованы оставить этот язык в лидерах разработки даже в мире микросервисов. Идет большая работа по уменьшению требований к оперативной памяти, размера виртуальной машины JRE, и разбиения платформы на модули, позволяющего исключить из приложения ненужные мегабайты библиотек JAR. Посмотрите на GraalVM от Oracle, оптимизирующей запуск программы и компилируемый код в бинарный, или библиотеки Quarkus и Microprofile, создающие минимальные микросервисы.

Сторонние библиотеки и инструменты

Все разработчики рано или поздно сталкиваются с похожими задачами и частыми проблемами. Так как хороший разработчик (и человек разумный в целом) как правило ленив, он пытается сразу же решить задачу так, чтобы никогда больше с ней не сталкиваться в будущем. Количество же инструментов, библиотек, блогов и стартапов в такой популярной и растущей стремительными темпами области как облачные вычисления просто поражает воображение. Многие задачи решены, решаются, а затем немедленно заново решаются с различной степенью эффективности, качеством документации и уровнем поддержки готовых решений.

Мы будем использовать многие из них, но это не совет на будущее и не “лучший инструменты или библиотека их всех”. Это лишь способ быстрее научиться главному, не изобретая уже много раз решенные вещи. Вполне может быть, что упомянутая в книге библиотека или инструмент выйдет из моды, обнаружится критические просчеты или просто будет заменен более удачным конкурентов. В это случае лишь поиск по основным тематическим сайтам поможет вам заменить их. Основы же останутся прежними.

Основные провайдеры облачных услуг - Amazon, Google, Microsoft

Основные технологии (контейнеризация, Docker, оркестрация сервисов Kubernetes) концепции приложений, созданных для облака Cloud Native, уже практически гарантированно работают на всех провайдерах облака, будь это три кита облачных вычислений Amazon Web Services (AWS), Google Cloud Platform (GCP), или Microsoft Azure, или провайдеры меньших размеров, часто также предлагающих интересные цены и услуги (российский #CloudMTS, Digital Ocean). Для нас же это означает, что на данный момент концепция стала стандартом, который очень широко поддерживается, и будет поддерживаться на многие годы вперед, а значит изучение ее основ, и технологий в ней, даст нам ключи к эффективной работе с любыми облаками. Созданные же единожды приложения можно будет и переносить между облаками, и запускать в гибридном облаке, работающем на нескольких провайдерах одновременно.

Дополнительные форматы книги на ipsoftware.ru

Я намеренно создал книгу, используя современные независимые платформы для издания - LeanPub, Lulu и российскую Ridero. Основная цель - снизить издержки классического издателя, в конечном итоге составляющие больше половины стоимости электронной книги, и особенно бумажной версий книги (в этом случае практически вся стоимость составляет траты на печать, хранение и распространения тиража). Книга также намеренно сделана

короткой, учитывая огромную доступность справочной информации в Интернете, и цена ее будет минимальной разрешенной электронными магазинами.

Различные платформы предоставляют разные уровень сервиса, форматы книги, и инструменты для работы с ней. База книги создана в формате Markdown на платформе LeanPub, сверстана для русских площадок с помощью Ridero. Вы можете найти и скачать все варианты и форматы книги на моей сайте www.ipsoftware.ru, особенно если вас не устроил изначально полученный вами вариант или формат. Более того, книга на сайте будет постоянно обновляться и меняться, в зависимости от найденных ошибок и необходимых добавлений.

1. Приложения, созданные для облака - концепция Cloud Native

> “Технологии Cloud Native позволяют создавать сложные системы в динамичной, современной среде частного, гибридного или коммерческого облака Cloud. Воплощают такой подход в жизнь контейнеры, сервисные сетки (service mesh), микросервисы, неизменная инфраструктура и декларативный способ управления ресурсами.“ Устав фонда Cloud Native Foundation, управляющего стандартами и общим направлением концепции.

Современная разработка классических приложений, вспомогательных сервисов, мобильных приложений и их серверных компонентов, и программного обеспечения в целом подразумевает постоянные изменения функциональности. Часто новые функции появляются несколько раз в день. При этом обновления, исправления ошибок и перезапуски не должны останавливать сервис и доступ к его функциям ни на секунду. Практически весь мир объединен глобальным скоростным доступом в Интернет, и одновременный доступ не только миллионов, но сотен миллионов пользователей к удачному приложению и сервису больше не является прерогативой технических гигантов, таких как Google, Apple и Facebook, и вполне доступен маленькому стартапу и индивидуальному программисту. Приложение должно быть готово к пиковой нагрузке, не уменьшая качество своего сервиса. Именно таким и будет приложение, с самого начала созданное для работы в облаке.

Основные положения концепции Cloud Native

Возможность быстро и эффективно наращивать функциональность приложения, не переписывая и не ломая уже существующие функции и компоненты, а также их взаимодействия, требует особого подхода к разработке в общем, и к выпуску готовых релизов и их запуску на серверах в частности. Более того, для каждой задачи хорош свой инструмент, что в мире программирования означает, что для каждой задачи чуть лучше может подходить свой собственный язык, его экосистема, и набор библиотек.

Реализовать это возможно с помощью так называемых “микросервисов” (microservices), слабо связанных между собой компонентов единой системы или приложения. Они обмениваются данными через сеть, используя стандартные сетевые протоколы, как правило это протокол HTTP и стандарт REST. Добавление и обновление одного микросервиса никак не затрагивает остальные части системы. Микросервисы связываются друг с другом через сетевые порты и абстрактные протоколы, и каждый из них может быть написан на любом подходящем

языке и технологии. Обновляются и перезапускаются они также независимо. Микросервисы часто противопоставляются единому, большому серверному приложению, так называемому “монолиту” (monolith).

Запуск на одной операционной системе разнородных приложений, написанных с помощью самых разнообразных технологий, как правило не сулит в себе ничего хорошего из-за конфликта системных зависимостей, библиотек и правил доступа. Эту проблему блестяще решают контейнеры (containers). Контейнеры - легкая форма виртуализации, они надежно изолируют приложения друг от друга, и в отличие от виртуальных машин, не требуют полной установки отдельной операционной системы. Запуск и остановка контейнеров практически мгновенна. Множество разнородных модулей и библиотек теперь смогут ужиться в одном сервере Linux, не мешая друг другу, и не требуя для запуска минут, как требуют полноценные виртуальные машины.

Одно из преимуществ приложения, разбитого на модули и микросервисы, работающих из собственных контейнеров - тонко настроенное горизонтальное масштабирование. Появляется возможность выделить наиболее нагруженную часть системы и запустить для ее сервисов и компонентов столько экземпляров, сколько необходимо для обработки текущей нагрузки. Для этого требуется практически неограниченная вычислительная мощность, растущая по требованию (ее еще называют эластичной) - эту мощь обеспечивают коммерческие провайдеры облака, такие как Google Cloud (GCP), Amazon Web Services (AWS), и Microsoft Azure.

Наконец, мощное, динамично меняющееся приложение, состоящее из сотен распределенных компонентов, соединенных между собой по сети, требует постоянного надзора и очень сложного управления, в том числе и для масштабирования и обновления компонентов. Здесь главную роль играет оркестратор контейнеров (orchestrator), самым популярным среди них без сомнения является Kubernetes. Для наблюдения трафика между компонентами, задержек, графиков исполнения запросов, и сбора и анализа журналов (logs) существуют целые комплексы программных решений, хорошо интегрированных с Kubernetes.

Первые выводы

Подводя краткий итог, мы поставили задачу создать максимально гибкое, устойчивое к отказам, всегда доступное приложение, способное выдержать пиковые нагрузки, и увидели, как эта задача может быть решена. Именно описанный подход и методы являются основой приложения, созданного для работы и развертывания в облаке (cloud native). Вот его ключевые атрибуты:

- Микросервисы (microservices) как способ максимально возможной слабой связи между подсистемами приложения. По сути эти компонентная разработка, с прицелом на абсолютно независимый друг от друга процесс разработки, свободный выбор технологии, а также независимые выпуски новых версий и их развертывание на сервере.

- Контейнеры (containers) - легкая виртуализация в пределах одной операционной системы (как правило Linux), не требующая огромных тяжелых виртуальных машин, включающих в себя полную отдельную операционную систему. Контейнеры позволяют множеству микросервисов незаметно друг для друга работать на одном сервере.
- Эластичная, практически бесконечно доступная при необходимости вычислительная мощность, то есть новые и новые сервера для запуска контейнеров. Эти сервера должны обладать эффективным, автоматическим, легко воспроизводимым способом запуска и конфигурации. Как правило, это обеспечивают коммерческие провайдеры облаков, владеющие большими центрами данных. Большие организации могут себе позволить собственные центры данных с работающими на их основе частными облаками.
- Оркестровка и управление контейнеров, внутри которых находятся микросервисы, в одном или множестве экземпляров. Основным инструментов управления является сейчас Kubernetes, мощный, расширяемый оркестратор, способный управлять, обновлять, масштабировать, настраивать взаимодействие для сотен микросервисов. Оркестратор работает с набором физических или виртуальных серверов в кластере.
- Наблюдение за сложной сетью микросервисов и их взаимодействием, в том числе за состоящими из множества мелких сетевых вызовов транзакциями и комплексными операциями. Необходимы эффективные инструменты для сбора и анализа журналов (logs). В динамической, распределенной среде любой мелкий вызов может таить в себе причину общего сбоя.

Теперь давайте взглянем чуть подробнее, какие технологии, подходы и архитектура обеспечивают успех каждого из столпов концепции cloud native.

Микросервисы - быстрый цикл разработки и постоянный выпуск

Микросервисы (microservices) - очередной виток развития компонентной разработки программных комплексов и приложений. Разбиение сложной задачи на составные более простые части, изоляция сложности, и поиск абстракций, позволяющих упростить и сделать задачу управляемой и решаемой - основа программирования в целом. Разбиение программы на пакеты, функции, классы, а затем и на совершенно независимо работающие друг от друга компоненты логически вытекает из анализа задачи.

Микросервисы - это компоненты вашего приложения, независимо друг от друга работающие в облаке и соединенные между собой не прямыми вызовами внутри одного процесса, а передачей данных по сети, используя заранее оговоренные протоколы (обычно HTTP или gRPC) и порты.

Эластичность и практически неограниченная вычислительная мощность облака дает нам возможность разбить приложение на логические компоненты и запускать их и управлять ими индивидуально. При необходимости легко увеличить пропускную способность приложения, увеличив количество экземпляров компонентов (работающих в виде микросервисов),

испытывающих наибольшую нагрузку. Это так называемое горизонтальное масштабирование - при работе в облаке его возможности практически безграничны, при условии выбора удачной архитектуры приложения, дающей возможность разбить его вычислительные потоки на независимые части. Вертикальное масштабирование же подразумевает рост мощности одного сервера и его аппаратных возможностей, что крайне ограничено, и более того, самые мощные серверы обычно очень дороги.

Передача данных между микросервисами осуществляется по сети, по хорошо известным протоколам, поддерживаемыми практически всеми известными языками и их библиотеками. Микросервисы больше не являются частью единого проекта и репозитория в системе контроля версия, и разрабатывающие их команды теперь свободны делать любой выбор, эффективно позволяющий решить задачу, стоящую перед компонентом. Это открывает двери для быстро меняющегося мира технологий, и когда-то сделанный выбор архитектуры и языка для одного компонента больше не диктует того же новым компонентами и сервисам.

Гораздо меньший размер и менее связанная с другими компонентами функциональность позволяет программистам быстро проводить в жизнь новые идеи, рефакторинг кода, и пробовать новые подходы и процессы разработки. Разумный размер кода делает процесс разработки быстрым и удобным. Это же позволяет проще и настроить системы постоянного контроля качества и развертывания сделанных изменений на сервере (CI/CD, continuous integration and delivery), и сделать их работу быстрой, позволяя программистам быстро проверить, было ли их последнее изменение удачным.

Обратной стороной компонентной разработки в распределенной среде является отсутствие гарантии работоспособности - любой сетевой вызов, в отличие от вызова функции внутри единого процесса, подвержен отказам и сбоям, иногда в течение долгого времени. Размытые границы между микросервисами диктуют аккуратный выбор протоколов и передаваемых структур данных. Тестировать взаимодействие микросервисов, взаимодействующих по сети, иногда бывает крайне сложно.

Мы подробнее рассмотрим некоторые аспекты дизайна и разработки микросервисов и похожих на них компонентов в отдельной главе.

Контейнеры - изоляция и гарантия неизменяемости версий

Мы только что увидели, как много потенциальных преимуществ может принести с собой разбиение приложения на независимые компоненты, или микросервисы. Особенно они важны для программистов, получающих намного больше свободы в своих экспериментах и выборе технологии. Однако запуск таких компонентов должен быть быстрым, а взаимодействующие технологии должны уживаться на одних и тех же серверах (в пределах кластера) без конфликтов и сложных конфигураций.

Решить эту задачу можно виртуальными машинами, запуская на них микросервисы. Однако виртуальная машина требует установки и запуска отдельной, самостоятельной операцион-

ной системы, и время ее запуска делает быстрое масштабирование и перезапуск компонентов практически невозможным. Как мы уже видели, эту задачу берут на себя контейнеры с их легкой виртуализацией с помощью возможностей Linux (другие операционные системы уже также их поддерживают). Время запуска контейнера практически то же, что для обычного процесса Linux, а изоляция приложений, их файловых систем, и ограничение их ресурсов мало чем отличается от полноценной виртуальной машины.

Все содержимое (файлы и зависимости приложения или его части), необходимое для запуска контейнеров, упаковывается в образы (image). Важным свойством образа является его неизменность (immutability), для каждой отдельной метки, или версии, этого образа. Поменять помеченный определенным образом образ с известной контрольной суммой уже невозможно. С практической точки зрения это означает, что созданная когда-то система, настроенная и работающая с определенным набором микросервисов, упакованных в образы для запуска в виде контейнеров, теперь всегда может быть заново воспроизведена в любой необходимый момент. Это важное качество воспроизводимости (reproducibility) гарантирует уверенность в текущем состоянии сложной, составной системы. Мы можем быть уверены в том, что работающая система не была запущена давно потерянным и никому больше не известным набором эзотерических скриптов.

В главе про контейнеры мы подробнее узнаем историю виртуальных машин и контейнеров, чуть подробнее взглянем на механизмы их работы, и на основной инструмент работы с контейнерами - Docker.

Облако - неизменная эластичная инфраструктура. “Феникс” вместо “снежинки”

Эта книга для разработчиков, и для нас, после того как мы создаем серверное приложение или сервис, зачастую начинается довольно туманный период его реальной эксплуатации (production), на основных серверах компании. Классически управлением и запуском готового выпуска приложения заведуют администраторы, или операторы (operators), заведующие всеми деталями настройки и управления серверами. Операторы могут использовать совершенно отдельный от разработки процесс запуска, и свои собственные инструменты для управления настройками серверов.

Для разработчиков в подобном процессе эксплуатации исправление и анализ ошибок или нестандартных ситуаций может стать настоящей головной болью. Если управление эксплуатацией совершенно отделено от выпуска и тестирования новых версий, анализ и воспроизведение ошибок особенно сложны, так как настройки и версии операционных систем и их зависимостей могут значительно отличаться от тех, что используются при тестировании или локальной отладке.

Особенно тяжело управлять и анализировать поведение большой серверной системы в случае, если каждый сервер представляет собой уникальную “снежинку” (этот термин предложил Мартин Фаулер), то есть обладает уникальным набором настроек и конфигураций

операционной системы и ее аппаратного обеспечения. В этом случае функциональность системы сливается с уникальностью сервера и становится очень трудно воспроизводимой, и довольно нестабильной.

Гораздо проще восстанавливать и копировать сервер, если он представляет собой “феникса”, способного быстро восстановиться из заранее подготовленного образа (“пепла”, если выражаться в терминах легенды). Еще лучше, если этот образ не бинарная копия диска, а список четких инструкций, по шагам восстанавливающих состояние сервера из известных проверенных компонентов. Эта инструкция хранится в системе контроля версий с историей всех изменений. Сервер, созданный по инструкции, всегда будет одинаковым, и таким образом, обеспечит неизменяемую инфраструктуру приложения (immutable infrastructure).

Если и разработчики, и операторы имеют доступ к легко читаемой, легко восстанавливаемой конфигурации своих серверных систем и кластеров, процесс передачи выпущенных сервисов из разработки в эксплуатацию становится прозрачным и легко поддерживаемым. Восстановление среды для тестирования или эксплуатации не представляет собой проблем. Слияние процессов разработки и управления иногда еще называют процессом DevOps (девопс, development + operations).

Облачные сервера как нельзя лучше подходят для реализации упомянутых выше “фениксов”, проверенных, неизменяемых серверов с прозрачной историей. Публичные провайдеры облака, такие как Amazon AWS, Google GCP и Microsoft Azure, создают свои виртуальные или реальные сервера из тщательно проверенных, безопасных версий известных операционных систем, которые не будут внезапно меняться в процессе работы сервера. Ваша команда DevOps затем может использовать подготовленные заранее инструкции для дополнительной автоматической настройки этих серверов.

В мире контейнеров все становится еще проще. Провайдеры облака как правило предоставляют оптимизированную операционную систему для запуска контейнеров (как правило особая версия Linux). Все зависимости и дополнительные настройки вы производите уже внутри самого контейнера, и указываете их при создании его образа, помечая его версией. Эта версия затем может многократно запускаться в виде контейнера и уже никогда не меняется. Управление сложными серверными конфигурациями значительно упрощается, и по сути, любой член команды, работающей в формате DevOps, способен без особого труда построить и восстановить любую конфигурацию системы, от тестирования до реальной эксплуатации, просто используя доступное в системе контроля версий описание развертывания и запуска системы.

Наконец, основные провайдеры облака как правило обладают мощными центрами данных. При недостатке вычислительной мощности и росте популярности вашего приложения вы сможете динамично расширить свой кластер, добавить в него новые сервера, и запустить на них необходимое количество экземпляров ваших микросервисов. Автоматизирует этот процесс оркестратор Kubernetes.

Оркестровка Kubernetes - декларативное описание состояния

Получив неизменную, эластичную инфраструктуру, и все преимущества изоляции и быстрого запуска контейнеров, необходим инструмент, обладающей достаточной мощью гибкостью для управления ими. Оркестратор Kubernetes, созданный на основе проверенной годами в компании Google системы управления контейнерами Borg, обладает всем необходимым для запуска и управления сложными системами, развернутыми в облаке. Все основные коммерческие провайдеры облака, Google Cloud, AWS, Azure, российские #CloudMTS и остальные, в обязательном порядке предоставляют сервисы на основе Kubernetes.

Исходный код Kubernetes открыт и находится на GitHub, это один из самых популярных проектов с огромным количеством программистов и компаний, работающих над ним. Запустить Kubernetes возможно на своем собственном, частном облаке и кластере, а начать эксперименты можно и вовсе на своем ноутбуке, используя совершенно полные локальные реализации, такие как Docker for Desktop и Minikube.

Как мы увидели в уставе фонда Cloud Native, декларативное описание является неотъемлемой частью этой концепции. В случае с Kubernetes это означает, что вместо развертывания и запуска приложений (упакованных в образы контейнеров) явными командами из скриптов и терминала, предпочтительным является *желаемое состояние кластера* (desired state). Состояние описывает все системы, микросервисы и сетевые настройки системы, а управляющая система Kubernetes заботится о том, чтобы кластер представлял собой именно желаемое состояние. Состояние описано в файлах YAML. В главах, посвященных Kubernetes, мы узнаем все детали нем, и научимся описывать систему декларативным способом.

Инструменты для сбора журналов и наблюдения

Развернув систему из множества микросервисов, общающихся между собой по сети, зачастую асинхронно, мы получим взрывной рост разнообразных условий, при которых система будет вести себя отлично от идеала. В отличие от монолитного приложения, журналы (logs) которого можно анализировать в относительном порядке, и не волноваться о сбоях вызовов внутри одного процесса, в распределенной, микросервисной среде каждый вызов может сорваться или внести каскадный сбой.

Вокруг концепции Cloud Native сложилась огромная, динамичная экосистема решений и продуктов, помогающих решить неизбежные проблемы распределенных асинхронных систем. Прежде всего это сервисные сетки (service mesh, смысл термина состоит в упорядочении и управлении сетевых переговоров микросервисов, но слово “сеть”, и мы используем сетку чтобы уменьшить путаницу), такие как Istio и Linkerd. Сервисные сетки решают многие проблемы, дают возможность наблюдать за сетевыми вызовами, строить графы, получить задержки, и многое другое. Системы сбора метрик, такие как Prometheus, позволяют

интегрировать метрики вашей системы в единые центры наблюдения. Управление журналами, например Fluentd, справляется со сбором и упорядочением десятков разнообразных журналов, полученных от микросервисов. Мы еще раз вспомним эти инструменты, когда будем рассматривать микросервисы в следующей главе.

Разработка на практике - 12 факторов облачного приложения

Теория и концепция Cloud Native, то есть приложений, созданных для облака, пока выглядит стройно и логично, и остальную часть книги мы посвятим практическому применению ее основных частей. Однако всегда интересно узнать "выжимку" опыта компаний, команд и программистов, которую уже попробовали разработку таких приложений, и увидели всю подноготную проблем, с которыми придется столкнуться - как мы знаем, в программировании многие неприятности скрываются именно в мелких деталях.

Команда облачного сервиса Heroku, популярного выбора для небольших команд и систем, собрала свои наблюдения в каталог из 12 факторов (12 factor app), наличие которых в дизайне и реализации системы резко повышает его шансы на успешную работу в облаке и простую поддержку готовой системы. Отсутствие этих факторов, или, что хуже, выбор противоположных решений, может впоследствии усложнить разработку и развертывание облачного приложения. Давайте посмотрим на эти факторы, и увидим, как они соотносятся с положениями концепции Cloud Native.

1 - Единая база кода

Весь код приложения, или микросервиса должен находиться в своем отдельном репозитории (GitHub или что-то еще). Использовать один репозиторий для нескольких сервисов не рекомендуется из-за возрастающей сложности сборки и связанности между компонентами системы. В главе про микросервисы мы подробнее узнаем, почему это хорошая мысль.

2- Явное описание и изоляция зависимостей

Облачное приложение ни в коем случае не должно рассчитывать, что на серверах кластера что-то будет доступно или предустановлено. Этот фактор отлично накладывается на рекомендуемый способ работы с контейнерами - вы всегда способны заново построить образ контейнера своего приложениях (обычно с помощью Dockerfile), и он включает в себя все необходимо для работы - любые библиотеки JAR, пакеты Node.js, и так далее.

3 - Управление конфигурацией

Гибкое приложение избегает включения любых элементов конфигурации в свой исходный код - это пароли, адреса баз данных, даже порты микросервисов-партнеров. Большую

помощь в реализации этого фактора оказывает Kubernetes. Он, хоть и не может вас заставить вынести всю конфигурацию в переменные окружения (environment variables), предоставляет удобные инструменты, такие как “секреты” (secrets) и карты конфигурации (config maps). Они описываются декларативно, в файлах YAML. Меняя карты конфигурации, вы с легкостью можете развернуть свою систему в совершенно другом окружении, например, для отладки, или у нового клиента на его собственных серверах.

4 - Вспомогательные ресурсы через конфигурацию

Система, использующая дополнительные внешние системы и ресурсы, такие как базы данных, хранилища неструктурированных данных, почтовые и СМС-сервисы, в идеале максимально абстрагирует свои связи с ними. Выделение точного интерфейса для работы с ними, и вынесение в переменные окружения всех параметров для доступа и соединения с такими ресурсами поможет системе уменьшить количество зависимостей и легкость работы в разнообразных облаках и окружениях. Вновь, карты конфигурации Kubernetes отлично справятся. Для более сложных случаев можно описать ресурс в виде объекта Kubernetes (CRD, custom resource definition).

5 - Строгое разделение построения и запуска системы

Система не должна запускаться из непроверенных изменений в коде или конфигурации. Собранная система помечается версией или меткой (tag), все собранные бинарные и конфигурационные файлы доступны для перезапуска в случае проблемы. Этот фактор прекрасно обеспечивают образы (image) контейнеров - они неизменны после сборки, вы знаете историю версию в репозитории образов (обычно Docker Hub), и можете строго воспроизвести любое состояние системы, не откатывая никаких изменений в коде.

6 - Сервисы без состояния

Микросервисы облачного приложения в идеале не обладают вообще никаким состоянием и стараются не хранить никаких промежуточных результатов для выдачи другим серверам (stateless, share-nothing). Это позволяет добиться легкой масштабируемости и восстановления системы. Необходимо рассчитывать на динамичность облака и то, что любой сервер или диск может быть перезапущен в любую минуту. Данные должны храниться в специализированных сервисах для данных, обычно управляемых облаком - облачных базах данных (Cloud SQL, Amazon RDS), кэшах Memcached, и других. Как мы увидим, именно микросервисы без состояния намного проще создавать с помощью Docker и управлять Kubernetes.

7 - Доступ через сетевые порты

Микросервисы общаются через сетевые порты, обычно с помощью HTTP в стиле REST, отсылая данные в формате JSON или XML, или используют бинарный протокол GRPC. Если

микросервис вызывает другие микросервисы-партнеры, адреса доступа к ним и их порты хранятся отдельно, в конфигурации. Данное требование идеально исполняется контейнерами, которые объявляют, по какому порту они будут ожидать соединений, и сервисами (service) Kubernetes, описывающим, как эти порты будут доступны в кластере. Все необходимое для работы HTTP сервера находится внутри контейнера (встроенные сервера, например Netty для Java).

8 - Масштабирование через запуск дополнительных экземпляров

Концепция микросервисов позволяет снизить количество ресурсов для четко выделенного компонента системы, и провести его точечное горизонтальное масштабирование - это возможно только в случае микросервисов без состояния (фактор 6). Kubernetes делает масштабирование, в том числе автоматическое в зависимости от загрузки системы, тривиальной задачей, и поддерживает желаемое количество экземпляров микросервиса.

9 - Быстрый запуск и надежная остановка

Микросервисы должны запускаться как можно быстрее, вновь в целях быстрой адаптации к возрастающей нагрузке, и более эффективного использования освобождающихся ресурсов. Легковесная виртуализация контейнеров Linux делает запуск новых контейнеров практически мгновенной, почти неотличимой от запуска обычного процесса. Отсутствие состояния и данных в основных компонентах бизнес-логики позволяет быстро их остановить, обновить, без потери данных и функциональности.

10 - Одинаковая среда разработки и эксплуатации

В больших распределенных системах, особенно если используется сложная авторизация, роли, базы данных, облачное хранилище данных, сразу же возникает вопрос, как организовать среду разработки с похожим поведением, для отладки и проверки нового кода. Часто в целях экономии производственные облачные системы заменяют на менее мощные, или даже на локальные эмуляторы, которые отдаленно напоминают среду эксплуатации (production), но все же имеют множество мелких различий, и называют это средой разработки (dev environment).

Авторы 12 факторов яростно протестуют против подобного подхода - среда разработки, среда тестирования, и среда эксплуатации должны полностью совпадать, даже если придется платить за дополнительные ресурсы. В долгосрочном плане это сэкономит множество ресурсов на поиске проблем и сделает возможным более быстрый выпуск надежного нового кода. Я думаю, многие из нас сталкивались не с полноценными средами разработки, и абсолютно не поддающимися воспроизведению в них ошибками реальной эксплуатации. Анализ ошибки в таком случае значительно усложняется.

Хотя контейнеры и Kubernetes не смогут автоматически предоставить вам идентичные среды, они сделают это намного проще, благодаря неизменности образов контейнеров, работающих в системе, и легкой переносимости конфигураций YAML. Следование факторам 2, 3, 4 и 6 также сделает создание идентичной среды разработки проще. Более того, если среды абсолютно идентичны, то любой член команды сможет выполнить развертывание, приближая команду к DevOps.

11 - Журналы logs в виде потока событий

В классических монолитных системах журналы пишутся на диск, в файлы. Используется заранее выбранный формат, архивация и инструменты для их обработки (например, Log4J для Java). Ситуация кардинально меняется для контейнеров и системы из распределенных микросервисов. Контейнеры эфемерны и их файловая система пропадает вместе с их остановкой, разные технологии применяют различные форматы журналов, а понять что происходит с системой целиком по разнородным журналам крайне сложно.

В облачном приложении журналы не сохраняются и не обрабатываются. Все записи делаются в стандартный вывод (standard output), тот самый, что выводится в терминал при ручном запуске. Именно стандартный вывод используется в контейнерах и Kubernetes. Дополнительные решения (ELK, Fluentd), работающие под управлением Kubernetes, собирают журналы с различных микросервисов, анализируют и хранят их, и предоставляют инструменты для полного анализа.

12 - Администрирование как часть приложения

Дополнительные административные задачи, такие как миграция данных или удаление неудачных записей из распределенного кэша, могут исполняться только из среды эксплуатации, эти задачи тестируются вместе с построением и выпуском системы, и поставляются вместе. Уверенность в том, что дополнительное администрирование сделано проверенным способом и в нужной среде уменьшит количество ошибок. Неизменность контейнеров и легкий откат к предыдущим версиям развертываний (deployment) в Kubernetes позволят исправить неудачный выпуск системы.

2. Микросервисы

> “Невозможно описать термин “микросервис”, потому что не существует даже словарного запаса для этой области. Вместо этого мы опишем список типичных черт микросервисов, но сделаем это со следующей оговоркой - большинству микросервисных систем присущи лишь некоторые из приведенных черт, но не всегда, и даже для совпадающих черт будут значительные различия от канона.” Мартин Фаулер (Martin Fowler), одно из первых выступлений, посвященных глубокому анализу микросервисов).

Как мы выяснили из первой главы, обзора концепции и технологий, “созданных для облака” (cloud native), практически неотъемлемой частью проектирования и разработки приложений для работы в облаке стали “микросервисы” (microservices), особенно бурно ворвавшиеся в тренд популярности на волне успеха стека технологий и способа разработки Netflix, Twitter, Uber, и до этого идей от Amazon.

Определить точно, что это за архитектура, и чем она формально отличается от очень известного до этого подхода SOA (service oriented architecture), то есть архитектуры ориентированной на сервисы, довольно сложно. Многие копья сломаны на конференциях и форумах, создано множество блогов, можно сделать определенные выводы. Прежде всего микросервисы отличаются от “монолитов” (monolith), приложений, созданных с помощью единой технологии или платформы, внутри которой находятся вся деловая логика системы, анализ данных, обслуживание и выдача данных пользовательским интерфейсам. Любое взаимодействие модулей, сервисов и компонентов внутри монолита как правило происходит в рамках одного или максимум несколько процессов.

Плюсы монолита очевидны - мгновенная скорость общения между сервисами и компонентами, зачастую в рамках одного процесса, общая база кода, меньшее количества ограничений на общений между компонентами и модулями, менее общие, более точные и выделенные интерфейсы между ними.

Однако с развитием облачных вычислений, и особенно легких контейнеров, изолирующих любые технологии, возрастанием скорости обмена данных по сети, и общей надежности и встроенной устойчивости к отказам, предоставляемых основными провайдерами облака, стало особенно удобно разбивать приложение на множество более мелких приложений. Они предоставляют друг другу сфокусированные, маленькие услуги и сервисы с помощью обмена информацией (как правило, это текстовый формат HTTP/JSON, или двоичный формат gRPC), независимые от использованных микросервисом технологий.

Подобное разбиение идеально ложится на разделение бизнес-функций в общем приложении, а что еще лучше, великолепно разделяет обязанности большой команды инженеров

на независимые, маленькие команды, способные к экспериментам, быстрым изменениям, и использованиям любых технологий.

Монолиты

Красивое слово монолит (monolith) описывает хорошо известный, наиболее часто используемый способ разработки программного продукта. Ваша команда определяется с набором требований к продукту и делает примерный выбор технологий и архитектуры. Далее вы создаете репозиторий для исходного кода (чаще всего GitHub), выделяете общую функциональность и библиотеки (пытаясь сократить количество повторного кода, DRY - don't repeat yourself!), и вся команда добавляет новый код и функциональность в этот единственный репозиторий, как правило, через ветви кода (branch). Код компилируется единым блоком, собирается одной системой сборки, и все модульные тесты прогоняются также сразу, для всего кода целиком. Рефакторинг и масштабные изменения в таком коде сделать довольно просто.

Однако, если брать разработку в облаке, и зачастую мгновенно и кардинально меняющиеся требования современных Web и мобильных приложений, описанные удобства грозят некоторыми недостатками.

Склонность к единой технологии

Единый репозиторий кода и одна система сборки естественным образом ведут к выбору основной технологии и языка, которые будут исполнять большую часть работы. Компиляция и сборка разнородных языков в одном репозитории неудобны и чрезмерно усложняют скрипты сборки и время этой сборки. Взаимодействие кода из разных языков и технологий не всегда легко организовать, проще использовать сетевые вызовы (HTTP/REST), что еще сильнее может запутать код, который находится рядом друг с другом, однако общается посредством абстрактных сетевых вызовов.

Тем не менее, для каждой задачи есть свой оптимальный инструмент, и языки программирования не исключение. Микросервисы, максимально разбивая и изолируя код частей системы, дают практически неограниченную свободу в выборе языка, платформы и реализации задачи, без взрывной сложности сборки проекта. Как мы вскоре увидим, контейнеры с блеском справляются с задачей легковесной виртуализации, и совершенно различные технологии способны с легкостью взаимодействовать друг с другом.

Сложность понимания системы

Часто говорят, что большая, созданная единым монолитом система сложна для понимания для новых членов команды. В мире технологий нередко размер команды резко растет, требуется срочно создать новую функциональность, и ключевым фактором становится скорость

начала работы с ней и ее кодом ранее незнакомых с ней программистов. Мне кажется, что это довольно неоднозначный момент, и качественно сделанная система с разбиением на модули, правильной инкапсуляцией и скрытием внутренних винтиков системы будет не сложнее для понимания, чем сеть из десятков микросервисов, взаимодействующих по сети. Все зависит от дисциплины и культуры команды.

Но в общем случае стоит признать, что созданная командой (с ее внутренней дисциплиной и культурой) система скорее будет более прозрачной и понятной в виде микросервисов и качественно разделенных друг от друга репозиториев, чем в виде огромного кода размером в сотни тысяч строк, особенно если новый программист начинает работу над четко определенной задачей в одном микросервисе.

Трудность опробования инноваций

Монолитная, сильно связанная система, где код может легко получить доступ к другим модулям, и начать использовать их в тех же благих целях не делать ту же работу заново (общие модули и библиотеки), может в результате затруднить создание новых возможностей, не способных идеально вписаться в существующий дизайн.

Представим себе, что мы написали отличную биржу для криптовалют. Неожиданно появляется новая валюта, но правила работы с ней совершенно не похожи, и просто не будут работать с нашими системами обработки заказов. В случае монолитной системы нам надо вносить изменения в общий код, и продумать множество граничных случаев, чтобы обработка всех валют могла сосуществовать. В концепции микросервисов идеально было бы обработку новой валюты отвести совершенно новому микросервису. Да, придется заново писать много похожего кода, но в итоге эта работа будет идеально соответствовать требованиям. Более того, если эта новая криптовалюта окажется “пустышкой”, ненужный микросервис удаляется, нагрузка на систему и ее сложность немедленно снижается - сделать такой рефакторинг в сильно связанном коде монолита может быть непросто, да и будет это и не самым первым приоритетом.

Дорогое масштабирование

Монолитное приложение собирается в единое целое и, в большинстве случаев, начинает работать в одном процессе. При возрастании нагрузки на приложение возникает вопрос увеличения его производительности, с помощью или *вертикального масштабирования* (vertical scaling, усиления мощности серверов на которых работает система), или *горизонтального* (horizontal scaling, использования более дешевых серверов, но в большем количестве для запуска дополнительных экземпляров (replicas, или instances).

Монолитное приложение проще всего ускорить с помощью запуска на более мощном сервере, но, как хорошо известно, более мощные компьютеры стоят непропорционально дороже стандартных серверов, а возможности процессора и размер памяти на одной машине ограничены. С другой стороны, запустить еще несколько стандартных, недорогих серверов

в облаке не составляет никаких проблем. Однако взаимодействие нескольких экземпляров монолитного приложения надо продумать заранее (особенно если используется единая база данных!), и ресурсов оно требует немало - представьте себе запуск 10 экземпляров серьезного корпоративного Java-приложения, каждому из них понадобится несколько гигабайт оперативной памяти. В коммерческом облаке все это приводит к резкому удорожанию.

Микросервисы решают этот вопрос именно с помощью своего размера. Запустить небольшой микросервис проще, ресурсов требуется намного меньше, а самое интересное, увеличить количество экземпляров можно теперь не всем компонентам системы и сервисам одновременно (как в случае с монолитом), а точечно, для тех микросервисов, которые испытывают максимальную нагрузку. Kubernetes делает эту задачу тривиальной.

Архитектура на основе сервисов (SOA)

Более гибким решением является разработка на основе компонентов, отделенных друг от друга, прежде всего на уровне процессов, в которых они исполняются. Архитектуру подобных проектов называют ориентированной на сервисы (service oriented architecture, SOA).

Разработка приложения в виде компонентов, и стремление свести сложные приложения к набору простых, хорошо стыкующихся между собой компонентов известна видимо с тех самых времен как программы стали разрабатывать. По большому счету подобная техника применима во многих областях человеческой деятельности.

Часто говорят, что классическая архитектура на основе сервисов отличается от ставших популярными сейчас “микро”-сервисов тем, что многое отдает на откуп “посредникам” (middleware), например системам обмена, настройки и хранения сообщений между компонентами и сервисами, так называемым “интеграционным шинам” (ESB, enterprise service bus). Микросервисы же эпохи облака минимизируют зависимость от работы со сложными посредниками и их правилами и проводят свои операции напрямую друг с другом.

Микросервисы по Мартину Фаулеру

Мартин Фаулер знаменит своими, как правило, очень успешными попытками найти структуру и систему в динамичном, хаотичном мире программирования, архитектуры, хранения данных, и особенно в мире сложных, высоконагруженных, распределенных приложений эпохи Интернета. Конечно же он попытался проанализировать и упорядочить волну популярности микросервисов.

Если попытаться вчитаться в цитату Мартина в начале этой главы, становится понятно, что микросервисы - явление вне рамок известных ранее архитектур, они не совсем укладываются в стандарты, и довольно разнообразно применяются на практике. Тем не менее, характерные черты микросервисов, используемые Мартином, очень хороши для первого, вводного анализа этой архитектуры, что нам для знакомства и нужно. Давайте посмотрим.

Сервисы как компоненты системы

Компонент (component) - одна из основополагающих идей в разработке программ. Идея проста - компонент отвечает за определенную функцию, интерфейс API, или целый бизнес модуль. Главное - компонент можно убрать из системы, заменив на другой, со сходными характеристиками, без больших изменений для этой системы. Замена компонентов целиком встречается не так часто, но независимое обновление (upgrade) компонента без потери работоспособности системы в целом очень важно для легкости ее обновления и поддержки. Классический пример - качественная библиотека (к примеру, JAR), модуль, или пакет, в зависимости от языка, которую можно подключить и использовать в своем приложении.

Микросервисы - это использование компонентов (зависимостей, библиотек и модулей) своего приложения не внутри одного, единого процесса приложения, а запуск их в отдельных, собственных процессах, и доступ к ним не напрямую, а через сетевые вызовы RESTful API / GRPC.

Логика в микросервисах, но не в сети и посредниках

Данная черта микросервисов противопоставляется предыдущим версиям распределенных систем (часто их называют SOA, service oriented architecture). В архитектуре SOA большая роль отводилась компонентам-“посредникам” (middleware), таким как сложные очереди сообщений (message queue), адаптерам данных, и общему набору логики между различными компонентами системы, называемому *интеграционной шиной ESB* (enterprise service bus). Зачастую львиная доля сложной логики всей системы находится не в самих компонентах, а именно в шине ESB.

В мире микросервисов это явно выраженный анти-шаблон (anti-pattern). Вся логика в обязательном порядке должна находиться внутри микросервисов (“разумные” точки доступа к сервису, smalt endpoint), даже если она повторяется в различных микросервисах. Сеть (и любой механизм передачи данных в целом), должна только передать данные, но не обладать никакой логикой (как еще говорят, “неразумные” линии передачи данных, dumb pipes).

Децентрализованное хранение данных

В монолитной архитектуре данные зачастую хранятся в единой, обычно реляционной, базе данных, со строго определенной схемой и обширным использованием запросов SQL. Микросервисы снова требуют противоположного подхода - в идеале никакого разделения данных, особенно через единую базу данных. Весь доступ, любые изменения данных происходят только через программный интерфейс API, предоставляемый микросервисом.

Микросервисы, таким образом, свободны в своем выборе способа хранения данных. Это может быть любой тип базы данных, например, нереляционная база NoSQL, база на основе документов, графовая база, или что-либо еще. Конечно, возникает значительная избыточность данных, но хранение данных не так дорого, а автономность и независимость развития каждого микросервиса, в идеале, должна окупить избыточность данных.

Культура автоматизации

Учитывая лавинообразный рост зависимостей между компонентами, и независимый набор технологий в каждом из микросервисов, надежный, автоматизированный выпуск системы и ее быстрые обновления - это жизненно необходимый элемент микросервисной архитектуры. Ручная сборка и управление микросервисами практически невозможны.

Непрерывная интеграция и тестирование (CI, continuous integration), непрерывное развертывание новых версий (CD, continuous delivery) - это обязательный атрибут команд, создающих микросервисы. Здесь огромную помощь оказывают контейнеры, со своей быстрой, легковесной виртуализацией, и “чистым”, изолированным пространством, в котором запускаются микросервисы, а развертывание значительно упрощает оркестратор Kubernetes, предоставляя мощный декларативный подход. Их мы тщательно рассмотрим в следующих главах, представить микросервисы без них уже практически невозможно.

Расчет на постоянные сбои

Микросервисы приводят к распределенной, гетерогенной, динамичной системе. Ей просто предначертано испытывать постоянные сбои, из-за сетевых вызовов, нагрузок, несовместимости версий, и многого другого. Этого не избежать, и на это нужно просто рассчитывать изначально, планируя каждый вызов к остальным компонентам системы как изначально не способный гарантировать успешное завершение.

Планировать сбои можно не только на этапе дизайна кода и отдельных тестов, но и более активным способом. Так, компания Netflix использует “обезьяну с гранатой” (Chaos Monkey, перевод не дословный, но наша фраза на русском языке вполне подходит!), отдельный сервис, постоянно выводящий из строя случайный набор микросервисов. Остальные компоненты системы должны подстроиться к ситуации, и правильно реагировать на сбои системы, проводя разумную политику отката своих действий, или же используя вспомогательные компоненты и альтернативные способы исполнения логики.

Небольшая команда, акцент на своей бизнес-области

Микросервисы как правило разрабатываются небольшой командой (известен практически анекдот от компании Amazon, что команда, работающая над микросервисом, всегда сможет насытиться двумя пиццами. Пиццы скорее всего большие, возможно, с разными наполнителями - так что в итоге это может быть и чуть больше 10 человек). Идея состоит в том, что небольшой команде проще координировать свои действия и быстро проводить новые идеи и изменения в жизнь.

Второй аспект - команда больше не состоит только из программистов серверной части (back end), пользовательских интерфейсов (front end, или UI), или же системных администраторов. Они работают вместе над одной, сконцентрированной бизнес-областью. Классический пример - онлайн магазин, и сервисы по доставке, оформлению заказов, и проведению платежей.

В мире микросервисов все эти сервисы разрабатываются, выпускаются в эксплуатацию, и развертываются отдельными командами, в своем ритме.

Более того, у каждой команды могут быть отдельные требования, процесс, планирование, и даже заказчики - внутренние или внешние. Общаются эти команды независимо, и микросервисы работают независимо, предоставляя четко очерченные программные интерфейсы API.

Интересно, что подобное требование лежит за пределами чисто технической, программистской зоны ответственности - такое распределение обязанностей должно лежать в основе всей организации целиком, с множеством автономных зон ответственности, без единого, “жесткого” центра управления. Как гласит известный закон Конвея (Conway’s law), структура организации обязательно проявит себя в планировании и производстве любых продуктов и сервисов этой организации.

Владение продуктом, а не реализация проекта

Зачастую, в классической модели разработки, команда программистов и дизайнеров реализует проект, исполняя поставленные перед ней требования (requirements). Что происходит после того, как проект “сдан”? Команда начинает работать над чем-то новым, переходит в другие проекты, а в случае субподрядчиков и контрактных услуг, работа с ней может быть полностью закончена. Обслуживание системы, исправление ошибок, ее обновление нередко попадает совершенно в другие руки.

Парадигма микросервисов предпочитает, чтобы команда разработчиков “владела” (*own*) своим проектом в начале его дизайна, в процессе создания и настройки микросервисов, и обязательно после формальной сдачи системы, непрерывно продолжая работу над ее эволюцией. Постоянный контакт с пользователями системы, наблюдение за ней в условиях реальной эксплуатации позволяет максимально эффективно понять, как ее развить и улучшить. Как вариант подобной системы популярна концепция DevOps (developer + operations, совместная работа как над разработкой, так и над поддержкой и эксплуатацией системы). Оркестратор Kubernetes сам по себе подталкивает работать в концепции DevOps.

Эволюционный дизайн

> “Эволюционный дизайн (evolutionary design) - попытка добиться результата не с одной попытки, чудесным образом дающей идеальное попадание, а в результате многих постепенных изменений, приводящих к эволюции дизайна и продукта”, Джошуа Кириевски (Joshua Kerievsky), адепт Agile/XP-разработки.

Микросервисы дают большую свободу приверженцам эволюционного дизайна, противникам тщательного предварительного анализа системы, и попытки понять ее целиком еще до того, как написана первая строка кода, и сделан выбор первых технологий для ее реализации. Вместо классического планирования и дизайна всей необходимой функциональности

системы, создается первое “примитивное” приближение (primitive whole). Система выглядит практически реально, но умеет очень мало. Пользователи получают возможность впервые посмотреть на систему, и конечно же, немедленно просят поменять или улучшить ее, не целиком, но значительно. В подобных итерациях и рождается истина.

Если создание монолитной системы немедленно приводит к сильными зависимостям от выбранной технологии, системы хранения данных и библиотек, то добавка новых микросервисов, и даже быстрое удаление “неудачных” или “временных” микросервисов становятся делом техники. У разработчиков развязаны руки - они могут свободно обращаться с выбором технологий, а также передвигать границы микросервисов - если изначально они были выбраны неверно, в первых итерациях их легко объединить или разбить.

Краткие итоги

Первый обзор типичных черт микросервисов внушает оптимизм. Четкое разделение обязанностей, абсолютная свобода в выборе технологий и хранении данных, точно настроенное масштабирование, и помощь контейнеров и Kubernetes выглядят очень заманчиво. Однако не стоит забывать о сложности распределенной системы, присущей любой, даже самой простой такой системе. Сетевые вызовы часто приносят с собой неизвестные заранее задержки (latency), и для быстрой работы приходится выполнять работу асинхронно (asynchronous), заранее не зная, в какой момент приходит ответ от остальных компонентов системы.

Сложность асинхронной, распределенной, динамично развивающейся системы в разы превышает сложность единого, монолитного приложения. Разделить систему на микросервисы зачастую очень сложно, если в компании “все делают все”, и нет четко очерченных бизнес-областей. В таких случаях вместо микросервисов получается “распределенный монолит” (distributed monolith), неповоротливая, сложная система, вместо которой мог бы иметь место более эффективный монолит.

Не забывайте, что два остальных столпа концепции Cloud Native совершенно безразличны к битве монолитов, микросервисов и SOA! Вы можете совершенно спокойно развернуть классическое, монолитное приложение Enterprise Java в контейнере под управлением Kubernetes, получив многие преимущества без излишней сложности.

Один из давних соратников Мартина, Сэм Ньюмен (Sam Newman), написал отдельную, и кстати, не слишком “толстую”, книгу по микросервисам, которую можно рекомендовать для чтения, и существует ее перевод на русский язык.

Разбиение системы на микросервисы

Если осмыслить основные качества системы, созданной на основе микросервисов, начинает казаться, что их использование - совершенно универсальное, великолепное решение, практически панацея, или как любят говорить в технологиях, “серебряная пуля”. Запустив систему в облаке под управлением Kubernetes, где многие неприятности и специфичные

проблемы микросервисов решаются за нас, можно ли спокойно получить все их преимущества?

Проблема же заключается больше в попытке понять, что будет микросервисом в вашей системе, а что будет лишь частью или библиотекой, работающей в составе большого сервиса. Неверное определение границ микросервисов (boundary) приведет к запутанному, сложному коду, чрезмерно раздутым программным интерфейсам API, и может сорвать все сроки разработки, а то и к поспешной попытке “склеить” все обратно в монолит.

Качественный процесс дизайна и архитектуры приложения подразумевает разделение компонентов, сервисов и объектов, представляющих собой данные, согласно области бизнеса (domain), для которого приложение разрабатывается. Это основа DDD, дизайна архитектуры приложения на основе области его применения (domain driven design). Однако именно здесь для микросервисов кроется неприятные подводный камень. Дело в том, что разбить компоненты и сервисы заранее очень тяжело, требования, как водяные знаки, появляются лишь по мере проявления приложения, пользователи зачастую полностью меняют свое мнение как только видят первые варианты приложения.

Это меньшая проблема для монолитов - рефакторинг компонентов и их интерфейсов довольно просто сделать внутри одного процесса и одной базы технологий и языков. В случае микросервисов перенести часть функциональности в другой сервис, зачастую написанный на другом языке или платформе, полностью поломать и поменять крупные интерфейсы REST/gRPC между ними очень непросто.

Отсюда вытекает один из начальных этапов разработки, направленный на выявление границ микросервисов. Разработка идет в соответствии с базовыми принципами DDD, и приложение разбивается на модули согласно выявленным *ограниченным контекстам* (bounded context) - области работы, автономной самой по себе. В примерах с вездесущими электронными магазинами это может быть обслуживание склада и инвентаря магазина, отдельно от них существует система доставки, обработка счетов и так далее. Но, модули работают как часть монолита, первого приближения эволюционного дизайна, упомянутого нами выше.

После получения первых, очень грубых приближений системы в целом, с минимумом возможностей, начинают проявляться более четкие границы ее ограниченных контекстов. Этот момент прекрасно подходит для разбиения модулей на микросервисы, если нужно, смену технологий в них, и децентрализацию данных. Принятые решения еще нужно будет пересмотреть, однако они уже основаны на реальном коде, дизайне, и не являются просто плодом сухой теоретической подготовки.

Обратная сторона медали

Подчеркнем еще раз - за блеском и преимуществами микросервисов, лежащих на поверхности, легко не заметить всех сложностей и совершенно другой парадигмы именно работы, эксплуатации всей системы в целом. Даже если удастся удачно разбить систему по ограниченным контекстам и минимизировать их зависимости, создать хорошо настроенные

программные интерфейсы API, изучить Kubernetes и успешно развернуть и масштабировать свое приложение, то понять в итоге, что же происходит в работающей системе, будет в разы сложнее.

Можно забыть о прямой отладке в вашем редакторе IDE, если ваш код взаимодействует с несколькими микросервисами, и все они отвечают асинхронно. Интеграционные тесты будут очень сложны и поддержка их требует много ресурсов. Понять по журналам (logs) одного компонента, что происходит в системе в целом, невозможно. Производительность системы будет настолько распределена между отдельными микросервисами и сетевыми вызовами, что измерять нужно будет все сразу. Распределенные транзакции между различными хранилищами данных. Расчет на сбои всего и вся. Доверие образам контейнеров. Задача защиты трафика между микросервисами системы, обслуживание сертификатов SSL, авторизация, безопасность, роли… Продолжать можно еще долго.

Есть хорошие новости - экосистема созданных для облака приложений Cloud Native буквально наводнена инструментами и решениями для перечисленных нами вызовов. Зачастую они бесплатны и с открытым кодом, и каждый день появляются новые решения. Самая распространенная проблема - управление сетевыми вызовами между микросервисами, отслеживание задержек, шифрование трафика - неплохо решается так называемыми *микросервисными сетками* (service mesh) - такими как Istio и Linkerd. Мы еще вспомним про них в дальнейших главах. Сбор распределенных журналов также отлично решается, например стеком ELK (Elastic, Logstash, Kibana), или Fluentd. Стандарт OpenTracing, метрики Prometheus, и отчеты Graphana уже встроены во многие библиотеки для создания микросервисов, и просто используя их, вы получите мощнейший центр наблюдения на своей системой.

Тем не менее, все это богатство надо изучить, выбрать нужное и подходящее вам, и настроить - эти издержки надо обязательно добавить в общую стоимость разработки реальной системы из микросервисов.

Резюме

Микросервисы выглядят заманчиво, а вместе с контейнерами и оркестрацией Kubernetes и вовсе как очевидный выбор. Тем не менее, существует высокая цена эксплуатации системы, созданной на их основе, и экосистема для работы с ними требует инвестиций времени и ресурсов. В этой главе нет примеров - каждая система уникальна, и зависит прежде всего от области своего применения (domain). Архитектурные решения высокого уровня трудно описывать без сложного конкретного примера, поэтому мы увидели все с высоты птичьего полета, но все концепции данной главы можно попробовать перенести на свой собственный проект.

Чтобы понять архитектуру и философию микросервисов чуть лучше, можно посоветовать следующие книги и ресурсы:

- Сэм Ньюмен (Sam Newman), “Создание микросервисов”.

- `www.cncf.io` - главный сайт фонда Cloud Native Foundation, посмотрите раздел проектов (projects), многие посвящены работе микросервисных систем, в том числе OpenTracing и Prometheus.
- Некоторые видео с конференций KubeCon удачно описывают микросервисы для людей с разной степенью подготовки, найдите их канал на YouTube.
- Эрик Эванс (Eric Evans), “Предметно-ориентированное проектирование (DDD).”
- `www.martinfowler.com` - статьи про микросервисы и связанные концепции. Кое-что доступно и на русском языке.

3. Контейнеры и Docker

Контейнеры (containers) - относительно новое слово и концепция, мгновенно захватившая мир разработки программного обеспечения за последние несколько лет. Это относительно новое достижение в попытке разработчиков и системных инженеров максимально использовать доступные им вычислительные ресурсы.

Если кратко вспомнить историю, то серверные приложения, сервисы и базы данных изначально располагались на выделенных для них физических серверах, в подавляющем большинстве случаев под управлением одного из вариантов операционной системы Unix (или ее клона Linux). С взрывным ростом вычислительных мощностей использовать один мощнейший сервер для одного приложения стало и расточительно, и неэффективно - на одном сервере стали работать несколько приложений, внутренних сервисов или даже баз данных. При этом незамедлительно возникли серьезные трудности - различные версии приложений использовали разные версии основных библиотек Unix, использовали разные несовместимые между собой пакеты расширений или дополнительные библиотеки, соревновались за одинаковые номера портов, особенно если они были широко используемы (HTTP 80, HTTPS 443 и т.п.)

Разработчики работали над своим продуктом и тестировали его на отдельно выделенных серверах для тестирования (среда разработки, development environment, или же дальше в среде тестирования, QA environment). На этих серверах сочетание приложений и сервисов было хаотичным и постоянно менялось в зависимости от этапа разработки, и как правило не совпадало с состоянием производственной (production) среды. Системным администраторам серверов пришлось особенно тяжело - совмещать созданные в изоляции приложения необходимо было развертывать и запускать в производственной среде (production), жонглируя при этом общим доступом к ресурсам, портам, настройкам и всему остальному. Надежность системы во многом зависела от качества настройки ее производственной среды.

Следующим решением, популярным и сейчас, стала виртуализация на уровне операционной системы. Основной идеей была работа независящих друг от друга операционных систем на одном физическом сервере. Обеспечивал разделение всех физических ресурсов, прежде всего процессорного времени, памяти и дисков с данными так называемый гипервизор (hypervisor). Делал он это прямо на аппаратном уровне (гипервизор первого типа) или же уже на уровне существующей базовой операционной системы (гипервизор второго уровня). Разделение на уровне операционной системы радикально улучшило и упростило настройку гетерогенных, разнородных систем в средах разработки и производства. Для работы отдельных приложений и баз данных на мощный сервер устанавливалась отдельная виртуальная операционная система, которую и стали называть виртуальной машиной (virtual machine), так как отличить ее “изнутри” от настоящей, работающей на аппаратном обеспечении ОС невозможно. На мощном сервере могут работать десятки виртуальных машин, имеющих

соответственно в десятки раз меньше ресурсов, но при это совершенно независимые друг от друга. Приложения теперь свободны настраивать систему и ее библиотеки, зависимости и внутреннюю структуру как им вздумается, не задумываясь об ограничениях из-за присутствия других систем и сервисов. Виртуальные машины подсоединяются к общей сети, и могут иметь отдельный IP-адрес, не зависящий от адреса своего физического сервера.

Именно виртуальные машины являются краеугольным камнем облачных вычислений. Основные провайдеры облачных услуг Cloud (Amazon, Google, Microsoft и другие) обладают огромными вычислительными мощностями. Их центры обработки данных (data center) состоят из большого количества мощных серверов, соединенных между собой и основным Интернетом сетями с максимально возможной пропускной способностью (bandwidth). “Арендовать” себе целый сервер, постоянно работающий и присутствующий в сети Интернет, было бы очень дорого и особенно невыгодно для только начинающих компаний-стартапов, или проектов в стадии зарождения, которым не нужны большие мощности. Вместо этого провайдеры облака продают виртуальные машины разнообразных видов - начиная от самых микроскопических, по сути “слабее” чем любой современный смартфон - но этого зачастую более чем достаточно для небольшого сервера, обслуживающего не более чем несколько простых запросов в минуту или того меньше. Более того, виртуальные машины оптимизированы - новая виртуальная машина создается по запросу в течение нескольких минут, версия операционной системы всегда проверена на уязвимости и быстро обновляется. Если виртуальная машина больше не нужна, ее можно быстро остановить и перестать платить за использование облачных ресурсов.

Тем не менее, виртуальные машины, несмотря на то что выглядят совершенно универсальным инструментом облака, обеспечивающим полную изоляцию на уровне операционной системы, имеют существенный недостаток. Это по прежнему полноценная операционная система, и учитывая сложность аппаратного обеспечения, большое количество драйверов, поддержку сети, основных библиотек, встроенное управление пакетами расширения (package manager), и наконец интерпретатор команд (shell), все это выливается во внушительным размер системы, и достаточно долгое время первоначальной инициализации (минуты). Для некоторых случаев это может не являться препятствием - в том случае если ожидается что количество виртуальных машин фиксировано или скорость их инициализации не является критической, и на каждой из них работает большое приложение, останавливать и перезапускать которое часто нет необходимости.

Однако, подобное предположение все чаще является препятствием для приложений с высокой скоростью разработки и постоянным появлением новой функциональности, и особенно это критично для архитектуры на основе микросервисов.

- Микросервисы в идеале очень малы, и даже самая слабая виртуальная машина может быть слишком неэффективна для них, как по избыточной мощности, так и по цене
- Одно из основных теоретических преимуществ микросервисов - быстрое, почти мгновенное масштабирование при увеличении нагрузки на них. В мире где удачное приложение собирает миллионы запросов в секунду, ожидание нескольких минут для по-

явления следующей копии сервиса просто недопустимо и практически лишает легкую масштабируемость смысла.
- Опять же из-за их малого размера микросервисы могут иметь намного меньше зависимостей и требований к операционной системе в которой они работают - полная операционная система, ограниченная гипервизором, является для них чрезмерным ресурсом.

Именно так на свет появилась следующая идея - контейнеры. Контейнеры позволяют перейти на следующий уровень разделения вычислительных ресурсов, не налагая на приложение, работающей в облаке, необходимость нести с собой операционную систему виртуальной машины и связанные с этим издержки.

Контейнеры - это Linux

Давайте сразу определим для себя, что представляют собой контейнеры, и чем они отличаются от виртуальных машин, чтобы избежать путаницы которая часто случается между ними. Контейнер - это набор ограничений для запуска приложений, которые поддерживаются ядром (kernel) операционной системы Linux. Эти ограничения заставляют приложение исполняться в закрытой файловой системе, со своим пространством процессов (приложение не видит процессы вне своей группы), и с квотами на использование памяти, мощности процессоров CPU, дисков, и возможно сети. При этом у приложения в таком ограниченном пространстве существует свой сетевой IP-адрес и полный набор портов, а также полная поддержка ядра системы - устройств ввода/вывода, управление памятью и процессором, многозадачность, и наконец самое главное, возможность установить любые расширения и библиотеки, не беспокоясь о конфликтах с другими приложениями.

Можно сказать, что приложение, запускающееся в контейнере Linux, “видит” стандартное ядро (kernel) операционной системы, так, как если бы ничего кроме этого приложения, и этого ядра, больше не существовало. Файловая система пуста, нет никаких дополнительных пакетов и библиотек, интерпретаторов shell и тем более никакого намека на графический интерфейс GUI. Примерно так же “ощущает” себя приложение, запускающееся в виртуальной машине, на которой установлена операционная система Linux, только в крайне урезанном варианте.

Иметь только возможности ядра Linux для большинства современных приложений недостаточно, они как правило зависят от множества расширений и библиотек, а также имеют свои файлы с данными. Здесь контейнеры также хороши - приложение свободно распоряжаться своим пространством контейнера так, как если бы оно находилось на своем отдельном виртуальном (или реальном) сервере. Возможно установить любые пакеты, расширения, библиотеки и скопировать файлы и данные внутрь контейнера, не опасаясь конфликтов. Все это будет закрыто внутри контейнера и недоступно другим приложениям из других контейнеров.

Запуск и настройка контейнеров может вылиться в большое количество связанных между собой системных команд, и требует определенных знаний команд и архитектуры Linux. Здесь свою нишу занял инструмент Docker, который великолепно справляется с настройкой и запуском контейнеров, именно он стал еще одной причиной взрывной популярности контейнеров, которые отныне не требуют расширенных знаний Linux.

“Что же, если я использую Mac OS или Windows?” спросите вы? Благодаря инструментам Docker контейнеры Linux доступны на любых популярных операционных системах. Еще раз вспомним, что для работы контейнеров требуется доступ к минимальному ядру Linux - именно это и предоставляет Docker, как правило с помощью скрытой внутри него минимальной виртуальной машиной. Отличная возможность не только использовать контейнеры, но и получить доступ к настоящему ядру Linux за считанные секунды на любой операционной системе и лэптопе без запуска тяжелых виртуальных машин с полной операционной системой!

Хотя идея контейнеров и сама их суть - это операционная система Linux, их растущая популярность и прекрасно подходящая для облачных приложений архитектура не могла пройти мимо всех провайдеров облака, особенно облака Microsoft Azure с большим фокусом внимания на серверные варианты операционных систем Windows. Компания Microsoft уже поддерживает контейнеры (и Docker) в своих операционных системах без использования виртуальных машин. Конечно это уже не Linux, но если вы работаете с кодом .Net, это неоценимая возможность использовать архитектуру контейнеров не переписывая свои приложения.

Docker

Контейнеры - отличная замена виртуальным машинам. Не требующие гипервизора, и всей тяжелой массы полной операционной системы, им нужна лишь уже установленная подходящая версия Linux с поддержкой контейнеров. Остается “только лишь” вопрос их настройки, запуска, остановки, управления ресурсами, файловыми системами, совместными данными и их томами, и многим другим. В принципе для этого не требуется ничего кроме инструментов Linux, но кривая обучения и время настройки всего этого будет весьма значимы. Все эти функции берет на себя инструмент Docker, и именно его легкость и логичность его команд и общей модели помогли контейнерам выйти на авансцену облачных вычислений намного быстрее.

Вы легко можете установить инструменты Docker на любые операционные системы, и он при необходимости обеспечит вам минимальную виртуальную машину с ядром Linux (смотрите сайт docker.com, вам понадобится бесплатная версия Docker CE, в ней есть все что необходимо для работы с любыми контейнерами).

После установки Docker можно начинать запускать контейнеры. Еще раз вспомним, что контейнер - ограниченная часть пространства операционной системы Linux, в которой есть ограниченный и прозрачный для приложения доступ ко всем функциям ядра (kernel)

системы. Никаких интерпретаторов команд shell, известных всем команд `ls`, `ps` или `curl` там просто нет. Соответственно, выполнить там можно только приложение без зависимостей или библиотек (статически скомпонованное, static linking, что означает что все возможные зависимости и библиотеки находятся внутри исполняемого файла программы). Зачастую этого слишком мало, особенно если вы рассчитываете на отладку приложения, хотите исследовать его журналы (logs), или посмотреть список процессов и использование ими ресурсов.

Чтобы не копировать каждый раз нужные инструменты вручную или сложными скриптами, в Docker ключевую роль играет уже готовый набор файлов и библиотек, который можно одной командой перенести в свой контейнер и спокойно там получать к ним доступ. Это так называемые образы (image) контейнеров.

Образы (image) Docker

Как мы уже поняли, сам контейнер в своей основе - лишь возможность вызывать системные функции и использовать сервисы ядра Linux, и в него необходимо перенести файлы с приложением, его зависимости, и возможно инструменты для отладки и диагностики приложения, включая интерпретатор команд shell, если понадобится взаимодействовать с контейнером в интерактивном режиме. Весь этот набор в архитектуре Docker хранится в образе (image). Образ - это статический набор файлов, инструментов, директорий, символических ссылок symlink, словом всего того, что требуется приложению и нам как его разработчикам, чтобы успешно его запустить и при необходимости отладить или диагностировать проблему.

Созданные один раз образы могут использоваться как заготовка для запуска контейнеров вновь и вновь. По умолчанию они хранятся в хранилище на вашем рабочем компьютере, а также могут иметь уникальные метки (tag, для отметки особенно важных или удачных образов). Все это удивительно напоминает систему контроля версий, и именно так работает хранение образов Docker на вашей машине. Более того, образы можно хранить в удаленном репозитории в сети, аналогично тому как храним версии и ветки исходного кода, например в GitHub. Создатели Docker поняли полную аналогию между хранением образов, необходимость разнообразных версий с помощью меток, и практически полностью скопировали команды Git для управления образами, а общедоступное для всех удаленное хранилище для них назвали конечно же… Docker Hub.

Чуть позже мы подробно узнаем про управление образами, а пока самое интересное для нас заключается в том, что в Docker Hub, также как в GitHub, есть открытые (public) образы, на основе которых можно запускать свои контейнеры, ничего не меняя в них, или же строить на основе них новые образы. Самые популярные образы как правило позволяют запустить в контейнере основные базы данных, кэши, веб-серверы и прокси-серверы, а также эмулировать популярные операционные системы, такие как Ubuntu или CentOS. Это прекрасный способ получить в свое распоряжение закрытое виртуальное пространства контейнера для любых экспериментов с использованием Linux, не волнуясь при этом за сохранность своей главной операционной системы.

Как и для репозиториев с исходным кодом на GitHub, образам на Docker Hub можно добавлять “звезды”, чтобы их отметить или выделить, а также увидеть сколько раз открытые для всех образы были скачаны (pull). Набравшие самое большое количество звезд и использований образы находятся в начале списка открытых образов, который вы можете увидеть на странице `hub.docker.com` - именно там вы увидите, что сейчас в мире наиболее популярно для запуска контейнеров, или в качестве базового образа для запуска своего приложения.

Интерактивные контейнеры - запуск и управление на примере образа Ubuntu

Если мы зайдем на открытое хранилище образов Docker Hub и посмотрим самые популярные образы (страница Explore), мы сразу же увидим там образ операционной системы Ubuntu - один из самых используемых. Интересно, что это не отдельное приложение, не веб-сервер и не прокси-сервер и совершенно точно не база данных, то есть это не что-то, что обычно требуется для разработки приложений или сервисов. Какой же смысл запускать операционную систему из контейнера?

В основном этот образ используется для экспериментов - вы можете запустить свое приложение из окружения Ubuntu со всеми ее стандартными зависимостями, пакетами и библиотеками, а также интерпретатором команд shell и всеми удобными инструментами Linux просто для того, чтобы иметь возможность отладить и отследить его работу в комфортном окружении, или же для того чтобы просто проверить его на работу в Linux, если у вас к примеру лэптоп с Mac OS. Отсюда еще один популярный вариант использования образа Ubuntu - возможность работать с легким контейнером внутри Linux, который запускается в течение секунд, вместо тяжелой виртуальной машины с гипервизором, на системах где основной операционной системой не является Linux.

И повторим еще раз, основное, что стоит помнить при запуске контейнеров, особенно таких как Ubuntu, Debian или CentOS - это не совсем операционная система! Работать ваш контейнер будет с общим ядром Linux, или с основной операционной системой, если вы уже работаете на Linux, или с минимальной виртуальной машиной Linux. Просто он будет ограничен своей “песочницей” со своей закрытой файловой системой, ограниченным пространством процессов и пользователей, и, возможно, ресурсов. Все что вы увидите внутри такого контейнера - это набор команд, файлов, символических ссылок, характерных для Ubuntu или другой операционной системы, но ниже этого уровня будет единое ядро. Другими словами, если вы запустите контейнеры Ubuntu, CentOS и Debian одновременно, все они будут работать с ядром и под управлением основной операционной системы Linux, и таким образом это просто удобный инструмент для эмуляции их инструментов и окружения.

Зная это, приступим наконец к запуску контейнера. Все команды Docker выполняются инструментом командной строки `docker`, а для запуска контейнера нужно просто выполнить `docker run <метка образа>`.

```
docker run ubuntu
```

Эта команда запустит контейнер на основе самой последней метки образа Ubuntu (`latest`). Если это первый запуск, она к тому же загрузит файлы этого образа из хранилища Docker Hub.

Интересно, что после запуска контейнер практически мгновенно завершит свою работу. Это его базовое свойство - если внутри контейнера не исполняется работающее приложение, он мгновенно завершается - контейнер должен быть эффективным и сразу освобождать свои ресурсы как только выполнение программы внутри него заканчивается. При запуске же Ubuntu ничего кроме интерпретатора bash по умолчанию не исполняется.

Здесь нам пригодится интерактивный режим запуска (ключ `-i`) - он присоединяет к контейнеру консоль для ввода и вывода и пока они активны, контейнер будет продолжать работать даже если в нем ничего не исполняется. Для эмуляции стандартного терминала пригодится также ключ `-t`.

```
docker run -it ubuntu
```

На этот раз Docker запустит контейнер на основе образа Ubuntu, и предоставит нам консоль с эмуляцией терминала - вы сразу увидите привычное приветствие интерпретатора bash и можете работать с ним так, как если бы это была реальная операционная система Ubuntu. Пока интерактивный режим активен, контейнер будет работать. Завершить работу контейнера из терминала можно набрав команду `exit`, или использовать сочетание `Ctrl-P-Q` - отсоединение (detach) интерактивного режима от работающего контейнера без его остановки.

Если вы хотите изначально оставить контейнер с Ubuntu запущенным, или он понадобится вам для продолжения экспериментов чуть позже, вы можете запустить его в отсоединенном (detach) режим, используя ключ `-d`:

```
docker run -d -it ubuntu
```

В этом случае терминал сразу не открывается, а контейнер запускается и продолжает работать в фоновом режиме уже без вашего участия, даже без работающих в нем процессов. Увидеть работающие в данный момент под управлением Docker контейнеры можно командой `ps`, заимствующей свое имя из той же Linux:

```
docker ps
…
CONTAINER ID        IMAGE               COMMAND             CREATED                 \
 STATUS              PORTS               NAMES
8abb1b4a6886        ubuntu              "/bin/bash"         Less than a second ago  \
 Up 1 second                             determined_lichterman
```

Вы видите примерный вывод команды `ps`, довольно полезный - указан образ, на основе которого был создан работающий контейнер, время его запуска и работы, а также два уникальных идентификатора - первый просто уникальный UUID, а второй некое сгенерированное забавное “имя”, которое чуть легче напечатать человеку.

Конечно, нет ничего проще подключить терминал к работающему контейнеру, если он уже работает в фоновом режиме - для этого предназначена команда `attach`. Ей необходимо передать уникальный идентификатор контейнера, к которому нужно присоединиться (один из двух):

```
docker attach determined_lichterman
```

Или

```
docker attach 8abb1b4a6886
```

Запустив эту команду, вы вновь окажетесь в терминале своего контейнера и сможете делать там любые эксперименты. Как и в первом случае, когда мы сразу же открыли терминал при запуске контейнера, команда `exit` закончит его выполнение, а клавиши `Ctrl-P-Q` просто отключат терминал, оставляя сам контейнер работающим.

Когда работающий в фоновом режиме контейнер вам станет больше не нужен, вы можете остановить его командой `stop`, как нетрудно догадаться, вновь указав его уникальный идентификатор:

```
docker stop 8abb1b4a6886
```

Или

```
docker stop determined_lichterman
```

Снова вызвав команду `ps`, вы увидите что контейнер был остановлен.

Можем себя поздравить - зная название популярных образов на Docker Hub и несколько только что опробованных команд, мы можем экспериментировать с эмуляцией популярных операционных систем Linux, базами данных и серверами, запуская их в контейнерах за считанные секунды, и экспериментируя с ними любым образом, не опасаясь мусора в своей основной операционной системе или проблем с другими приложениями - контейнеры Docker надежно изолируют все, что происходит внутри них.

Открытие мира для контейнера - веб-сервер nginx и работа с портами

Итак, проводить эксперименты в контейнерах с различными версиями операционных систем Linux, разнообразными системами, работающими на них, и получать все это в пределах секунд, великолепно. Однако основная идея контейнера - изоляция именно вашего приложения или дополнительных сервисов, необходимых для его работы, чтобы эффективно разделить ресурсы, избежать сложностей с развертыванием, и мгновенно проводить горизонтальное масштабирование.

Вновь взглянув на самые популярные образы на сайте Docker Hub, мы увидим там знаменитый веб-сервер nginx. Он часто используется для поддержки уже готовых приложений в качестве просто веб-сервера или балансировщика нагрузки. Давайте запустим последнюю версию его образа в контейнере:

```
1 docker run -d nginx
```

Запускать его лучше сразу в отсоединенном режиме (detach, `-d`), так как при запуске этого образа, в отличие от образа Ubuntu, процесс nginx сразу запускается и работает пока мы его не остановим, и запуск образа без ключа `-d` приведет к тому что мы будем наблюдать консоль с выводом сообщений nginx и выйти из нее можно только вместе с самим nginx с помощью `Ctrl-C`.

Запустив команду `docker ps`, вы увидите что контейнер с работающим сервером nginx существует и увидите его уникальное имя. Что же дальше? Что если просто хотим использовать его как свой веб-сервер, только работающий из контейнера, чтобы не запускать его на основной операционной системе и иметь всю гибкость и безопасность контейнера? Достоверно известно, что любой веб-сервер обслуживает порт HTTP 80, можно ли получить к нему доступ, если он находится в контейнере?

Открытые (exposed) порты - основной путь взаимодействия между контейнерами, которые, как мы знаем, в общем случае максимальным образом изолированы и от основной операционной системы, и друг от друга. Именно через открытые порты можно получить доступ к работающим в контейнерах приложениям (не присоединяя к ним терминалы), и обеспечить взаимодействия множества контейнеров между друг другом. Многие образы, подобные nginx, автоматически указывают, какие порты будут активны в контейнере - в nginx это конечно будет порт HTTP 80.

Указать взаимодействие между портами основной операционной системы (системы, ресурсами которой пользуется Docker, то есть хоста), и контейнера, можно все той же командой `run`. Давайте остановим уже запущенный ранее контейнер (легкое упражнение на повторение предыдущего раздела), и запустим nginx снова, на этот раз указав на какой порт главной операционной системы мы переадресуем (forward) открытый в контейнере порт 80 (возьмем порт 8888):

```
docker run -d -p 8888:80 nginx
```

После запуска такого контейнера откройте в браузере адрес `localhost:8888` или `0.0.0.0:8888` и вы увидите приветствие сервера nginx. Переадресовать открытый порт контейнера на порт основной операционной системы-хоста в общем случае можно так:

```
docker run -p [порт основной хост-системы]:[открытый порт контейнера] [имя образа]
```

Если вновь посмотреть список активных контейнеров командой `ps`, мы обнаружим что она теперь сообщает нам информацию об открытых портах и их переадресации:

```
docker ps
…
CONTAINER ID        IMAGE               COMMAND                  CREATED             \
 STATUS              PORTS                   NAMES
5e0c7f30656d        nginx               "nginx -g 'daemon of…"   5 minutes ago       \
 Up 5 minutes        0.0.0.0:8888->80/tcp   loving_rosalind
```

Когда вы вполне насладитесь возможностями сервера nginx, не забудьте остановить работающий контейнер командой `docker stop [имя]`.

Это основное предназначение и главное достижение контейнеров - вы можете запустить столько разнородных систем в отдельных контейнерах, сколько вам нужно, и на стольких серверах, как подсказывает вам здравый смысл, ваша архитектура и текущая нагрузка на приложение. Все они будут надежно изолированы друг от друга, все будут использовать свои собственные зависимости, библиотеки и пакеты, или виртуальные машины и интерпретаторы, и взаимодействовать будут через заранее оговоренные открытые порты.

Резюме

Вычислительные ресурсы дороги, и зачастую запуск множества приложений на одном сервере грозит множеством конфликтов библиотек, зависимостей, сетевых адресов и портов, а также возможными уязвимостями безопасности. Мы кратко изучили эволюцию от виртуальных машин к контейнерам, и можем сделать следующие краткие выводы:

- Контейнеры - это изоляция приложения, основанная на возможностях операционной системы Linux (иногда Windows Server), позволяющих отделить процессы друг от друга, выделить им отдельную виртуальную файловую систему и ресурсы.
- Контейнеры - это не виртуальные машины. Они работают напрямую с ядром системы Linux, на которой запущены, и не имеют внутри себя никакой собственной операционной системы или гипервизора. Благодаря этому их запуск по скорости и удобству мало чем отличается от запуска приложения напрямую на основной операционной системе.

- Инструмент Docker организует эффективный запуск, остановку и управление контейнерами Linux, и позволяет работать с ними на популярных операционных системах посредством минимальных виртуальных машин.
- Зависимости и файлы, необходимые для работы контейнера, упаковываются в образы (image), которые как правило хранятся в общедоступном репозитории Docker Hub, имеющем управление и модель команд, схожую с репозитории исходного кода GitHub.
- Популярные образы, такие как Ubuntu или Nginx, часто используются для тестирования и экспериментов без запуска тяжелых виртуальных машин. Доступ к ним организован через открытые порты, переадресуемые на порты основной операционной системы (хоста), где работает Docker.

4. Создание образов Docker. Команды Dockerfile.

В предыдущей главе мы рассмотрели основные концепции и базовое устройство контейнеров (containers). Контейнеры, поддерживаемые возможностями изоляции операционной системы Linux или, чуть реже, Windows Server, способы обеспечить легкую, скоростную виртуализацию с использованием общего ядра операционной системы, и крайне эффективно разделить и изолировать вычислительные ресурсы мощного сервера или кластера, работающего в удаленном облачном центре данных. Главный инструмент для организации и запуска стандартных контейнеров - Docker.

Все зависимости контейнеров, их файлы и библиотеки, упакованы в так называемый образ (image) контейнера. Образ с определенной меткой (tag) является неизменным (immutable), и гарантирует одинаковую работу контейнера и логики приложения или сервиса внутри него при переносе и перезапуске на любых кластерах и серверах. Образы хранятся в репозитории образов, самый популярный - это официальный репозиторий Docker Hub. Все это делает контейнеры идеальным способом переноса функциональности и зависимостей сложной распределенной системы между серверами, кластерами, и провайдерами облачных вычислительных ресурсов.

Нам, как разработчикам, прежде всего интересно, как создавать новые образы контейнеров, в которых мы будем размещать свои приложения, или микросервисы, а затем запускать контейнеры из этих образов в облаке. Управлять ими, как правило, мы станем с помощью Kubernetes. Давайте займемся этим.

Структура Dockerfile. Основные команды. Базовый образ.

Создать свои собственные образы для запуска своих команд, приложений или микросервисов с помощью Docker чрезвычайно просто. Необходимо указать перечень зависимостей, файлов, библиотек и основного, базового образа в так называемом файле `Dockerfile`. Формат так прост и популярен, что его поддержку вы найдете в любых предпочитаемых вами редакторах и IDE, иногда с использованием расширений (plugins, или extensions для VS Code).

Вот основа любого файла `Dockerfile`:

```
# Это комментарий
# Каждый файл Dockerfile должен начинаться с FROM
FROM [базовый_образ]

[Команда|Инструкция] [аргументы]
```

Любой новый образ должен на чем-то основываться - как мы помним, контейнер работает в общей операционной системе, имея доступ лишь к ядру, и даже простейшие консольные приложения требуют базовых библиотек для вывода данных на консоль и работы с терминалом. Базовый образ - это обычно или некий набор файлов, отвечающий дистрибутиву Linux, или чуть более расширенный набор библиотек, инструментов и зависимостей для компиляции и запуска приложений для выбранного языка. Стоит еще раз вспомнить, что все версии и названия Linux, используемые для создания образов - это просто файлы с инструментами и библиотеками. Ядро операционной системы будет общим, доступным через систему выполнения контейнеров Docker.

Именно базовый образ указывает команда `FROM`, правила выбора образа такие же, как и при запуске образа командой `docker run`. Если не указывать версию вместе с меткой tag явно, это будет `latest` - обычно последняя, самая свежая версия образа.

Следующая распространенная команда - `RUN`. Она запускает команду уже внутри контейнера. Этих двух команд вполне достаточно, чтобы создать первый собственный образ (image):

```
# Используем полную версию Ubuntu как базовый образ
FROM ubuntu
# Любые команды Ubuntu теперь доступны для запуска RUN
# Но, они запускаются при построении образа, не для запуска контейнера
RUN echo "привет мир!" > hello_world
```

Мы сохраним этот файл в директории `dockerfile/helloworld`, и попробуем построить на его основе образ контейнера. Сделаем это команда `docker build`:

```
$ docker build . -t helloworld
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM ubuntu
 ---> 775349758637
Step 2/2 : RUN echo "привет мир!" > hello_world
 ---> Running in 98b510ad11b3
Removing intermediate container 98b510ad11b3
 ---> eb795c9beae9
Successfully built eb795c9beae9
Successfully tagged helloworld:latest
```

Для запуска этой команды мы перешли непосредственно в директорию, где находится наш Dockerfile, указали расположение этого файла (текущая директория `.`), и самое главное, название образа нашего контейнера (`-t helloworld`). Дополнительную версию после названия образа мы для простоты не включили, и по умолчанию такая команда всегда будет строить образ с версией `helloworld:latest`.

Как мы видим из напечатанного в консоли, наш новый образ создается в два этапа - сначала скачивается и используется базовый образ Ubuntu (он конечно же будет скачан только один раз, и после этого сохраняется в кэше вашей машины для ускорения процесса), а затем запускается команда, которая создает простой файл с эпохальной фразой “привет мир!”

Давайте запустим созданный собственными руками новый образ! Начнем с интерактивного режима с терминалом (`-it`) и посмотрим, что у нас есть в нашей файловой системе:

```
$ docker run -it helloworld
root@68ec78485349:/# ll
total 76
drwxr-xr-x   1 root root 4096 Nov  7 17:53 ./
drwxr-xr-x   1 root root 4096 Nov  7 17:53 ../
-rwxr-xr-x   1 root root    0 Nov  7 17:53 .dockerenv*
drwxr-xr-x   2 root root 4096 Oct 29 21:25 bin/
drwxr-xr-x   2 root root 4096 Apr 24  2018 boot/
drwxr-xr-x   5 root root  360 Nov  7 17:53 dev/
drwxr-xr-x   1 root root 4096 Nov  7 17:53 etc/
-rw-r--r--   1 root root   21 Nov  7 17:51 hello_world
…

root@68ec78485349:/# cat hello_world
привет мир!
root@68ec78485349:/# exit
exit
```

Как мы видим, наш новый образ успешно запущен, контейнер работает, файловая система взята из базового образа Ubuntu, и созданный в процессе построения образа файл `hello_world` на месте и содержит именно то, что мы хотели.

Однако, если мы попробуем просто запустить контейнер, он тут же закончит свою работу:

```
$ docker run helloworld
$
```

Дело в том, что команда `RUN` просто исполняет указанные ей инструкции при построении образа, в нашем случае создавая файл, или запуская любые другие команды, однако после построения образа она вызываться уже не будет.

Чтобы указать команду, которая будет выполняться после запуска контейнера из образа image, используется команда `CMD` или `ENTRYPOINT`. Добавим их и создадим новый файл `Dockerfile` в папке `helloworld-loop`. Вместо создания файла в процессе построения образа, скопируем файл и скрипт для его печати командой `COPY`.

Вот наш скрипт для печати содержимого файла в цикле:

```
#!/bin/bash
while true; do date; cat hello_world; sleep $1; done
```

Скрипт будет печатать содержимое нашего файла в цикле, добавляя текущее время, в промежутке делая паузу. Размер паузы передается в параметре. Перенесем все нужные нам зависимости внутрь нового образа:

```
# Используем полную версию Ubuntu как базовый образ
FROM ubuntu

# Поменяем рабочую директорию на более удобную
WORKDIR /opt/helloworld

# Скопируем нужные нам для работы контейнера файлы в образ
# Обратите внимание - ./ отвечает директории, указанной командой WORKDIR
COPY hello_world ./
COPY print_loop.sh ./

# Команда CMD или ENTRYPOINT исполняется при запуске контейнера
# Сначала идет команда, затем список аргументов. У нас - длина паузы.
CMD ["/opt/helloworld/print_loop.sh", “2”]
```

Здесь у нас целая гроздь новых команд `Dockerfile`, все они, тем не менее, чрезвычайно просты и логичны:

- `WORKDIR` - меняет рабочую директорию в файловой системе контейнера. Для образности, представьте эту команду в виде обычной `mkdir`.
- `COPY` - копирует файл из директории, в который вы запустили команду `docker build`, в файловую систему контейнера. Обычно самая полезная и часто используемая команда для переноса исходного кода и библиотек в контейнер. Обратите внимание, что по умолчанию `COPY` переносит файлы в корень файловой системы, после команды `WORKDIR` - в эту новую директорию, а еще вы можете указать ей абсолютный путь файловой системы, куда следует поместить файлы.
- `CMD` (или `ENTRYPOINT`) - команда, которая будет выполняться после запуска контейнера. Основная форма - массив в квадратных скобках, где указывается команда и ее аргументы. Мы просто запустим свой shell-скрипт. Ему требуется параметр, мы его передаем

в том же массиве. Разница между командами `CMD` и `ENTRYPOINT` не так велика, основная разница в том, что аргументы для `CMD` чуть проще изменять при запуске контейнера. Детали легко найти в документации. Есть еще один формат этих команд - исполнение напрямую оболочкой системы shell, в этом случае следует просто указать команду целиком, без массива и кавычек. Однако использованная нами только что форма записи более гибкая и обычно предпочтительнее.

Повторим построение образа `helloworld` уже на основе нашего нового `Dockerfile`, и посмотрим, что теперь получается при запуске контейнера из этого образа:

```
$ docker build . -t helloworld
…
Step 2/5 : WORKDIR /opt/helloworld
 ---> Using cache
 ---> f35f404f3440
Step 3/5 : COPY hello_world ./
 ---> Using cache
 ---> 689899f448f4
Step 4/5 : COPY print_loop.sh ./
 ---> b04eda22b54c
Step 5/5 : CMD ["/opt/helloworld/print_loop.sh", "2"]
```

Как видно, к построению образа добавились наши новые шаги. Запустим новый контейнер:

```
$ docker run helloworld
Fri Nov  8 22:40:12 UTC 2019
привет мир!
Fri Nov  8 22:40:14 UTC 2019
привет мир!
Fri Nov  8 22:40:16 UTC 2019
...
```

Теперь в нашем образе находится по большому счету настоящее приложение - оно запускается и печатает в цикле информацию. Так как наш цикл бесконечный, остановить контейнер командой терминала `exit` не получится - тут пригодятся команды `docker ps` и `stop`, которые мы как раз применяли в прошлой главе.

Только что узнанных команд на удивление хватает для построение реальных образов контейнеров. Мы вполне можем перенести свое приложение и его ресурсы в контейнер, и запустить его при начале работы контейнера. Теперь давайте посмотрим, как создавать образы для реальных приложений и языков программирования.

Создание образов для приложений Java, Go, Node.js

Основная задача образа контейнера image - обеспечить упаковку всех необходимых зависимостей для беспроблемного переноса запускаемого в контейнере приложения или микросервиса между любыми серверами и провайдерами облака. В случае реальных, написанных нами программ это означает, что мы должны удостовериться, что все виртуальные машины Java, зависимости скомпилированного приложения, необходимые ему ресурсы правильно сохранены в образе контейнера и смогут запускаться на любой системе, совместимой с контейнерами.

Обычная проблема при создании образа - копирование бинарного файла с программой или сервисом, не совместимым со стандартами Linux, особенно для таких языков как Go или C++. К примеру, собрав приложение Go на своем ноутбуке Mac, вы не сможете перенести его в контейнер - внутри контейнера действуют стандарты Linux, и ваше приложение не запустится, несмотря на то, что среда запуска контейнеров (container runtime) Docker работает на том же самом ноутбуке.

Лучшее решение в этом случае - компилировать и собирать (build) приложение как часть построения образа image, инструкциями Dockerfile. В этом случае все происходит непосредственно внутри операционного ядра контейнера, и полученный образ будет совместим с любыми стандартными средами запуска контейнеров, в том числе в коммерческих провайдерах облака.

Что же использовать в качестве базового образа? Снова Ubuntu, или может быть, какую-то еще версию Linux, а затем скопировать туда все необходимое для компиляции и сборки языка программирования инструменты? Мы можем вздохнуть с облегчением - основная часть этой работы уже сделана. Упаковка приложений и сервисов в образы контейнеров стала настолько популярна, что все распространенные языки, их основные версии, нужные для работы с ними инструменты уже доступны на открытом репозитории Docker Hub. Надо остается подобрать нужную версию языка и систему сборки, и скопировать файлы с кодом своего приложения.

Java

Java - по прежнему король языков программирования, когда речь заходит о больших корпоративных системах и серверных приложениях (enterprise). Ничего не мешает нам запускать сервисы, написанные на Java, внутри контейнеров Docker (кстати говоря, контейнеры в некотором роде уменьшили значимость виртуальной машины JVM и важность знаменитого слогана “написано однажды, запускается везде” - ведь сами контейнеры позволяют это сделать вообще для любого языка и библиотеки).

Самая популярная библиотека для построения RESTful сервисов и серверных приложений - без сомнения Spring Boot, а система сборки - Maven. Давайте незамедлительно засучим

рукава и в течение 10 минут упакуем сервис Java и Spring Boot в образ контейнера image, а затем запустим его.

Вот наш сервис, работающий с протоколом HTTP - мы создали его с помощью удобного инструмента Spring Initializr, помогающего быстро создать заготовку приложения:

```
package com.porty.dockerfile;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

/** Простейший HTTP сервис Java с использованием Spring Boot */
@SpringBootApplication
public class HelloJavaSpringBoot {
   // запускает стандартный сервер Jetty, порт 8080
   public static void main(String[] args) {
       SpringApplication.run(HelloJavaSpringBoot.class);
   }


   @RestController
   public static final class HelloWorldController {
       // обрабатываем запрос к корневому пути /
       @GetMapping("/")
       public String helloWorld() {
           return "Привет, это Java Spring Boot из контейнера!";
       }
   }
}
```

Здесь все просто - мы используем стандартные инструменты библиотеки Spring Boot, чтобы создать приложение (`SpringApplication.run`), и обработать запросы к корневому маршруту `/`. Работать это приложение сможет на любой приличной версии Java, 8, 9, 11, 12, 13 (да, именно так, версий в Java теперь с избытком!). Располагаться этот файл для сборки проекта Maven должен в стандартной директории `src/main/java`.

Spring Initializr помог автоматически указать все необходимые нам зависимости и создал стандартный файл сборки для инструмента Maven. Многие детали опущены, но все можно найти на GitHub в примерах книги, и тем более в любом примере Spring Boot:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...

<artifactId>hello-world</artifactId>
  <packaging>jar</packaging>
<name>Hello Java and SpringBoot</name>
<version>1.0.0</version>
...

<dependencies>

  <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter</artifactId>
  </dependency>
…
```

Мы указываем, что будем собирать свое приложение в виде архива JAR, назовем его `hello-world` версии `1.0.0`. Остальное указывает, какие компоненты и библиотеки Spring Boot нам понадобятся.

Это все! Мы можем собрать и запустить этот сервис, и посмотреть, как он отвечает на запросы через порт 8080 (это порт по умолчанию). Давайте теперь соберем и упакуем сервис в образ контейнера Docker. Такой стандартный и очень простой файл сборки `Dockerfile` по большому счету подойдет для большинства приложений Java:

```
# базовый образ - OpenJDK 11 и установленный Maven
FROM maven:3.6.2-jdk-11

# Соберем и запустим приложение в этой директории
WORKDIR /app

# Для сборки проекта Maven нужны исходные тексты программы
# и непосредственно файл сборки pom.xml
COPY pom.xml ./
COPY src/ ./src/

# Компиляция, сбока и упаковка приложения в архив JAR
RUN mvn package

# Запуск приложения виртуальной машиной Java из базового образа
CMD ["java", "-jar", "/app/target/hello-world-1.0.0.jar"]
```

Вот что мы сделали для упаковки приложения Java и Maven:

- Для начала взяли базовый образ `maven`, его легко найти на репозитории Docker Hub. Есть различные версии, мы использовали версию, основанную на версии OpenJDK 11.
- Указали рабочую директорию (`app`) и скопировали туда файл сборки и код приложения из папки `src`. Прелесть системы Maven в том, что это все, что нам требуется, чтобы собрать практически любое приложение Java.
- Теперь, прямо в процессе построения образа нового контейнера, мы запустили компиляцию и упаковали приложение в архив JAR.
- Наконец, полученный архив запускается стандартной командой `java -jar` при старте контейнера. Плагин Spring Boot для Maven позаботится о том, чтобы все зависимости и библиотеки приложения были упакованы в один большой архив (fat jar).

Построим новый образ `java-hello`:

```
$ docker build . -t java-hello

Step 1/6 : FROM maven:3.6.2-jdk-11
 ---> 3b2476ab3d10
Step 2/6 : WORKDIR /app
 ---> 8ff277d1acce
Step 3/6 : COPY pom.xml ./
 ---> 96fae6707e9a
Step 4/6 : COPY src/ ./src/
 ---> 613b4042db7e
Step 5/6 : RUN mvn package
 ---> Running in 652db37dca12
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/b\
oot/spring-boot-starter-parent/2.1.4.RELEASE/spring-boot-starter-parent-2.1.4.RELEAS\
E.pom
…
[INFO] BUILD SUCCESS
…
Step 6/6 : CMD ["java", "-jar", "/app/target/hello-world-1.0.0.jar"]
```

Все шаги логичны и нам уже знакомы - но обратите внимание на то, что Maven будет заново скачивать все зависимости и библиотеки JAR из Интернета, и компилировать приложение каждый раз при построении контейнера. В этом есть плюс - это “чистая” сборка, не зависящая от кэша и состояния вашей машины. Большой минус - постоянное скачивание библиотек и долгое время сборки. Чуть позже мы увидим различные решения этой проблемы.

Давайте запустим наш контейнер - не забудьте, это серверное приложение, и надо переадресовать необходимые порты на порты локальной машины командой `-p`:

```
$ docker run -it -p 8080:8080 java-hello
…
2019-11-08 INFO 1 --- Starting HelloJavaSpringBoot v1.0.0
…
Jetty started on port(s) 8080 (http/1.1) with context path '/'
```

Контейнер успешно запущен из созданного нами образа, и компоненты Spring Boot запустили встроенный HTTP сервер, отвечающий по адресу 8080. Мы перенаправили этот порт на такой же порт своей собственной операционной системы, и теперь можем проверить, что отвечает наш сервис:

```
$ curl localhost:8080
Привет, это Java Spring Boot из контейнера!
```

Образ создан и полностью автономен и работоспособен. Для запуска нашего нового Java-сервиса не нужно больше ничего - ни установленных виртуальных машин Java определенных версий, ни настроек пути `PATH`, ни дополнительных библиотек JAR. Контейнеры сдерживают свое обещание - собранный один раз образ с сервисом или приложением теперь можем быть использован сколь угодно много раз для запуска этого сервиса на любых серверах, кластерах и других вычислительных ресурсах, не требуя никаких дополнительных настроек!

Go

Язык Go стал намного популярнее за пределами создавшей его компании Google как раз на волне популярности контейнеров и управляющих ими систем, особенно Kubernetes. Именно на Go написаны Docker и Kubernetes, а также несколько известных платформ схожей направленности, таких как OpenShift. Go - намеренно простой язык, настолько простой, что полностью игнорирует ставшие такими привычными концепции программирования как классы, объекты и исключения (exceptions). Для эффективности применяется компиляция в бинарный код и автоматическая сборка мусора, чтобы избежать печальных проблем с ручным управлением памятью в C++.

Писать на Go несложно, синтаксис похож на C, библиотеки гораздо выше уровнем, ну а сборка мусора сразу же избавляет от многих головных болей. Давайте напишем заготовку будущего RESTful микросервиса на основе сервера HTTP из стандартной библиотеки Go:

```go
package main

import (
  "fmt"
  "log"
  "net/http"
)

func main() {

  http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Привет из контейнера с сервисом Go")
  })

  log.Fatal(http.ListenAndServe(":8080", nil))

}
```

Всего несколько строк кода позволяет нам запустить HTTP сервер (`http.ListenAndServe`), мы используем обычный порт 8080, а отвечать на запросы станем с корневого пути `/`, используя метод `http.HandleFunc`. Компилятор Go соберет для нас бинарную, быструю версию сервиса для необходимой нам платформы.

Мы хотим собрать и запустить этот микросервис для работы в контейнере, и можем сделать это прямо в процессе построения образа, в файле `Dockerfile`:

```
# базовый образ - компилятор и все необходимое для Go 1.13
FROM golang:1.13

# Соберем и запустим приложение в этой директории
WORKDIR /app

# Скопируем код программы в файловую систему контейнера
COPY main.go .

# Соберем программу из исходного кода, в файл hello-go
RUN go build -o hello-go main.go

# Запустим программу при запуске контейнера
CMD ["./hello-go"]
```

Репозиторий Docker Hub содержит базовые образы, в которых есть все необходимое для сборки и запуска приложений Go определенной версии. Мы берем последнюю на момент

написания версию 13, копируем файл с кодом программы, собираем ее, и указываем, что при запуске контейнера входной точкой будет наша новая программа.

Запустим сборку образа контейнера и сразу же запустим его, не забыв переадресовать порт контейнера 8080 на свою операционную систему:

```
$ docker build . -t go-hello
…
Step 1/5 : FROM golang:1.13
1.13: Pulling from library/golang
...
Step 5/5 : CMD ["./hello-go"]
Successfully built d553c426de6c
Successfully tagged go-hello:latest
...

$ docker run -p 8080:8080 go-hello
…

$ curl localhost:8080
Привет из контейнера с сервисом Go
```

Как мы видим, собранный как часть образа контейнера бинарный микросервис Go прекрасно запускается и обслуживает порт 8080. Неважно, на какой операционной системе вы находитесь - Unix, Mac, Linux, или Windows, Docker запустит минимальную виртуальную машину Linux, запустит все шаги по сборке образа вашего контейнера внутри пространства Linux, а затем безопасно и изолированно запустит из полученного образа новый контейнер и программу из бинарного кода Linux.

Во многом благодаря изоляции и идеальной переносимости контейнеров Go стал намного популярнее - не нужно больше думать о платформах, зависимостях и необходимости сборки приложения под каждую необходимую архитектуру - достаточно один раз скомпилировать приложение и упаковать его в образ контейнера. Это же верно и для других собираемых в бинарный код языков, таких как C++ и Rust.

Node.js

Node.js - отличный способ применить свой опыт в JavaScript для разработки серверных приложений и тех же самых микросервисов. Это интерпретатор `node` и набор библиотек (модули Node.js, module), которые позволяют использовать асинхронную модель программирования, особенно подходящую для RESTful сервисов и обработки сетевых запросов.

Если вы слышали о Node.js или пробовали работать с ним, то следующий код покажется вам прекрасно знакомым:

```
const http = require('http');

const hostname = '0.0.0.0';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Привет от контейнера с сервером Node.js!');
});

server.listen(port, hostname, () => {
  console.log(`Сервер запущен по адресу http://${hostname}:${port}/`);
});
```

Мы используем модуль `http` и создадим локальный HTTP сервер (`createServer`), привязав его к порту 3000 (почему не 8080? По какой-то причине 3000 гораздо популярнее в мире Node.js!). Останется его запустить функцией `listen`.

Нетрудно догадаться, что репозиторий Docker Hub содержит все необходимое для работы с основными версиями Node.js, в виде образа контейнера. Давайте используем версию Node.js 12 в качестве базового образа:

```
# базовый образ - npm, node и все остальное для Node.js 12
FROM node:12

# Исходный код
COPY hello-world.js .

# Точка входа - запуск кода интерпретатором node
CMD ["node", "hello-world.js"]
```

Для Node.js у нас получился самый простой `Dockerfile` - мы просто копируем свой код внутрь файловой системы контейнера, а затем запускаем интерпретатор `node` при запуске контейнера, указав точку входа командой `CMD`.

Повторим уже хорошо известную последовательности действий - соберем свой новый образ, и запустим его, не забыв, что номер порта у нас теперь 3000:

```
$ docker build . -t nodejs-hello
…

$ docker run -p 3000:3000 nodejs-hello
Сервер запущен по адресу http://0.0.0.0:3000/
…

$ curl localhost:3000
Привет от контейнера с сервером Node.js!
```

Интерпретатор `node` так же успешно запущен внутри изолированного пространства контейнера. Мы сможем запустить сколь угодно много и какие угодно версии Node.js, с любыми комбинациями модулей, а контейнеры позаботятся об изоляции, и легкой переносимости между любыми серверами и облаками.

Еще одно - не забывайте, что все использованные нами в этом разделе базовые образы с инструментами и инфраструктурой языков можно применить и для быстрых экспериментов, или даже для непосредственной разработки.

Если вы не хотите устанавливать на свою рабочую машину “зоопарк” из технологий, пакетов, и инструментов командной строки, вы всегда можете запустить один из основных образов с Docker Hub в интерактивном режиме (`docker run -it`), и использовать все нужные вам инструменты, компиляторы и зависимости в изолированном от своей машины пространстве процессов и отдельной файловой системе.

Многоступенчатая сборка. Размер образа image

Итак, конструировать новые образы image для запуска контейнеров совсем несложно, благодаря простоте и прозрачности формата `Dockerfile`, и огромному выбору существующих базовых образов для работы с любыми версиями и вариантами языков распространенных языков программирования. Компилировать и собирать приложение из исходного кода оптимально прямо в процессе сборки образа, чтобы минимизировать влияние своей собственной локальной операционной системы, ее зависимостей, и настроек своих компиляторов и инструментов, и получить максимально “чистый”, не зависящий ни от чего образ, способный запустить приложение на любых серверах.

Но, есть одна неприятность. Давайте посмотрим на построенные только что нами образы, использовав команду `docker images` - она выдаст нам полный список скачанных нами из Docker Hub и построенных собственными руками образов на своей машине, и покажет нам *размер* каждого из них. Вот что мы увидим для только что собранных нами образов для популярных языков программирования:

```
REPOSITORY          TAG              SIZE
nodejs-hello                        latest                  908MB
go-hello                              latest                    810MB
java-hello                           latest                   662MB
...
```

Казалось бы, в век скоростного доступа в Интернет и довольно дешевой стоимости хранения данных, размер примерно в один гигабайт не является чем-то шокирующим. Тем не менее, это не совсем то, что *обещала* нам сама концепция контейнеров. Вспомним еще раз - контейнер использует ядро существующей операционной системы. Ему необходимы только используемые приложением дополнительные инструменты и библиотеки. Он должен запускаться практически мгновенно.

Но простейший веб-сервер размером в один гигабайт? Это чрезвычайно неэффективно, это снизит скорость запуска и масштабирования системы из множества контейнеров. В конечном итоге, хранение данных в коммерческом облаке не бесплатно, и большое количество огромных образов скажется на стоимости облачных услуг.

Причина конечно же в базовом образе. Мы строим свое приложение прямо в “чистом” контейнере, в процессе сборки образа из инструкций `Dockerfile`, и это без сомнения правильно. Но после этого наше приложение или сервис “тащит” за собой все инструменты и библиотеки, необходимые только для сборки и компиляции, но не для его работы.

Решение - многоступенчатая сборка (multi-stage build), специально созданная Docker для подобных случаев. Этот тот случай, когда лучше сразу увидеть все в действии. Посмотрим, что мы можем сделать, чтобы уменьшить чудовищный размер образа `go-hello` со скомпилированным сервисом Go (который вообще то скомпилирован в бинарный код Linux и должен занимать минимальный размер из всех языков!):

```
# первая ступень - компилятор и все необходимое для Go 1.13
FROM golang:1.13 as builder

# Соберем и запустим приложение в этой директории
WORKDIR /app

# Скопируем код программы в файловую систему контейнера
COPY main.go .

# Соберем программу из исходного кода, в файл hello-go
# Необходимо указать дополнительные флаги сборки Go
RUN CGO_ENABLED=0 GOOS=linux go build -a -o hello-go main.go

# вторая ступень - спартанская версия Linux Alpine
FROM alpine:3.10
```

```
16
17 # Используем такую же рабочую директорию
18 WORKDIR /app
19
20 # Скопируем собранный бинарный код из первой ступени
21 COPY --from=builder /app/hello-go .
22
23 # Запустим программу при запуске контейнера
24 CMD ["/app/hello-go"]
```

Многоступенчатая сборка - не что иное, как возможность использовать промежуточные, временные образы, использовать для них любые, не обязательно одинаковые базовые образы в процессе сборки, и использовать файлы из предыдущих этапов, копируя их в следующий этап.

Именно это мы и проделали в своем новом `Dockerfile`, собрав образ со своим микросервисом в два этапа. Комментарии очевидны, можно только подвести итоги:

- Каждая ступень сборки начинается с инструкции FROM, и указывает базовый образ только для этой ступени. Может быть произвольное число инструкций FROM, базовый образ в последней инструкции и будет окончательным для нового образа. В нашем примере это широко распространенная “спартанская” версия Linux Alpine, размером около 5 мегабайт! Она популярна для встроенных приложений со строгими требованиями к ресурсам. В ней есть базовый набор инструментов Linux, достаточных для базовой отладки приложения.
- Каждой ступени можно присвоить имя (в нашем примере - `builder`), или же указывать ступень по номеру (начиная с нуля).
- Команда `COPY` позволяет копировать файлы из предыдущих ступеней (по имени или номеру) с помощью флага `--from`.
- Каждая ступень может заново указывать рабочую директорию, копировать свой набор файлом, и быть совершенно самостоятельной.
- Для работы собранного сервиса Go в системе Alpine нужно собрать его специальным образом, с минимальными зависимостями, детали можно найти в документации Go.

Соберем наш образ снова, используя усовершенствованный `Dockerfile` из двух ступеней, и посмотрим его размер:

```
1 $ docker build . -t go-hello
2 ...
3 $ docker images | grep go-hello
4 go-hello                          latest   12.9MB
```

Мы уменьшили образ в 80 раз! Теперь образ действительно соответствует девизу контейнера - быстрая, легкая виртуализация без огромных пакетов, инструментов и полной операционной системы. Запустив новый образ, мы сможем убедиться, что качество сервиса нисколько не пострадало от уменьшения размера образа в десятки раз.

Если вы совсем не планируете использовать терминал и оболочку shell, можно уменьшить наш образ до минимума. Пустой базовый образ `scratch` - это минимум того, что есть в Docker. В нем нет вообще ничего - что будет означать, как мы помним, просто доступ к ядру операционной системы. Для нашего сервиса Go этого достаточно. Если мы поменяем `alpine` на `scratch`, и снова соберем образ, вот что у нас получится:

```
$ docker images | grep go-hello
go-hello                            latest   7.39MB
```

Еще на 5 мегабайт меньше, по сути это просто размер бинарного файла, собранного компилятором Go. Идеально для встроенных систем и ограничений в объемах данных, не забудьте только, что запуск в интерактивном режиме (`it`) и работа с терминалом внутри такого контейнера будут уже невозможны.

Многоступенчатая сборка Java

Давайте посмотрим, что можно сделать для оптимизации размера образа (image) с нашим микросервисом на Java и Spring Boot (`java-hello`). Сразу скажем, что таких фантастических результатов, как с Go, достигнуть не получится - нам в любом случае понадобятся виртуальная машина Java, ее библиотеки и все дополнительные JAR-файлы для работы Spring Boot. Но значительное уменьшение образа все равно возможно - снова применим двухступенчатую сборку, и вместо пакета разработки (JDK) используем пакет запуска JRE (Java Runtime Environment), значительно меньший по размеру:

```
# первая ступень - базовый образ - OpenJDK 11 и установленный Maven
FROM maven:3.6.2-jdk-11 as builder

# Соберем и запустим приложение в этой директории
WORKDIR /app

# Для сборки проекта Maven нужны исходные тексты программы
# и непосредственно файл сборки pom.xml
COPY pom.xml ./
COPY src/ ./src/

# Компиляция, сборка и упаковка приложения в архив JAR
RUN mvn package

```

```
15 # Минимальная версия JRE, версия 11, открытая версия OpenJDK
16 FROM openjdk:11-jre-slim
17
18 # Используем такую же рабочую директорию
19 WORKDIR /app
20 # Скопируем архив JAR из первой ступени
21 COPY --from=builder /app/target/hello-world-1.0.0.jar .
22
23 # Запуск приложения виртуальной машиной Java OpenJDK 11
24 CMD ["java", "-jar", "/app/hello-world-1.0.0.jar"]
```

Все инструкции нам уже хорошо знакомы по многоступенчатой сборке сервиса Go. Вторая ступень будет использовать минимальный образ JRE на базе OpenJDK, варианта Java с открытым исходным кодом, версии `11-slim` - с минимально необходимым набором модулей (modules). Вы найдете этот обновленный `Dockerfile` в той же директории `java-hello` с исходным кодом и инструкциями Maven. Соберем образ заново:

```
1 $ docker build . -f Dockerfile-multistage -t java-hello
2 …
3 $ docker images | grep java-hello
4 go-hello                          latest   218MB
```

Как видно, у нас получилось уменьшить размер образа “всего лишь” в три раза, но это огромный выигрыш. Можно уменьшить размер еще больше, найдя подходящую версию Java на базе Linux Alpine, обычно это более старая версия Java 8, впрочем, прекрасно работающая для большинства серверных приложений. Попробуйте это сделать в качестве небольшого упражнения.

Чуть другой подход можно применить для Node.js - там этап компиляции и сборки по сути отсутствует, но можно оптимизировать количество пакетов и инструментов, оставив только необходимое для запуска приложения, и значительно уменьшить окончательный размер образа своего сервиса.

Репозитории образов. Метки, версии, и latest

Все собранные нами в этой главе образы image до сих пор хранятся в нашей локальной файловой системе, и совершенно недоступны для других членов вашей команды, для запуска в облаке, или же для тестирования в системах непрерывной интеграции и развертывания CI/CD. В прошлой, обзорной, главе про Docker мы выяснили, что образы хранятся в доступных через Интернет репозиториях, с историей меток и изменений, подобно системе контроля версий кода, и сходно с главным репозиторием открытого кода GitHub, главный репозиторий для образов Docker называется Docker Hub.

Что прекрасно, репозиторий Docker Hub совершенно бесплатен (если вы согласны с тем, что ваши образы будут всем доступны). Надо лишь создать свою учетную запись. После этого нужно войти в нее со своей машины:

```
$ docker login
```

Давайте попробуем отправить образ с нашим сервисом Go (он самый маленький и быстрый) в репозиторий Docker Hub (`push`):

```
$ docker push go-hello
..
access.. denied
```

Ничего не вышло, в доступе было отказано. Дело в том, что по умолчанию, если не указывать учетную запись, все образы отправляются в стандартную библиотеку Docker Hub, к которой у нас нет доступа - туда помещаются только самые популярные, тщательно проверенные на безопасность образы. Метка для пользовательских образов должна включать в себя учетную запись в следующем формате - `{учетная_запись_Docker}/имя образа:[необязательная версия]`.

Перестраивать образ заново нет необходимости - мы можем просто указать новую метку с помощью команды `docker tag`, и отправить помеченный учетной записью образ в репозиторий Docker Hub:

```
$ docker tag go-hello {учетная_запись_Docker}/go-hello
$ docker push {учетная_запись_Docker}/go-hello
The push refers to repository [docker.io/{учетная_запись_Docker}/go-hello]
eac13675ca89: Pushed
37ce9121a810: Pushed
...
```

Теперь все прошло успешно. Обновите страницу со списком образов, хранящейся в вашей учетной записи, и вы увидите новый образ, только что отправленный в репозиторий.

Все образы, собранные нами в этой главе, в своих метках (tag) использовали только название, но никогда не указывали *версию*. Если версия не указывается, образ помечается версией `latest` - что просто означает, что именно этот образ был собран последним.

Казалось бы, в чем проблема? Вспомним еще раз, что запускаемый на основе образа контейнер обеспечивает максимальную переносимость и неизменность (immutability) системы. Так как уже созданный и собранный образ поменять нельзя, воссоздание системы на основе известных образов и версий становится тривиальной задачей - например, всегда важно воспроизвести производственное (production) окружение, чтобы понять причину сложной

ошибки, которая происходит в условиях реальной эксплуатации, но не в среде разработки или тестирования (QA environment).

Метка `latest` же чрезвычайно подвержена постоянным изменениям, в том числе случайным. Любой образ, построенный без указания определенной версии, автоматически получает версию `latest`, и предыдущая версия образа просто исчезает. Более того, если при запуске контейнера образ с версией `latest` уже есть в кэше сервера, или в кэше системы управления контейнерами, он не будет заново скачиваться, даже если образ с этой меткой в репозитории был обновлен (Kubernetes позволяет обойти это ограничение, но это надо делать явно, через дополнительные настройки системы).

Гораздо лучшая практика работы с метками - указание точных версий, и по возможности использование одной версии только для одного образа, с автоматическим увеличением номера версии при изменении функциональности приложения в образе. Особенно хорошо для этого подходит *семантические версии* (SemVer, детали можно найти в Интернете). В общем случае они начинаются с версии 0.1.0, и всегда следуют формату `X.Y.Z`, где

- `X` - главная (major) версия, она увеличивается при больших изменениях функциональности и программных интерфейсов API, как правило, несовместимых с предыдущей главной версией.
- `Y` - дополнительная (minor) версия, увеличивается при появлении новой функциональности, полностью совместимой с предыдущей главной версией.
- `Z` - версия “патча” (patch), обычно прибавляют при исправлении мелких ошибок, без каких-либо новых возможностей системы, как еще называют такие исправления, “заплатки” (отсюда и слово patch).

Использование уникальной метки для всех образов гарантирует возможность воспроизведения поведения системы, контейнера, и сохраняет список изменений в *каждой* когда-либо построенной версии сервиса, приложения, его компонента (микросервиса), и, в общем случае, всей сложной распределенной системы в целом.

В нашем случае лучше пометить образ так:

```
$ docker tag go-hello {учетная_запись_Docker}/go-hello:0.1.0
```

При дальнейшей разработке, новые образы будут увеличивать версию (согласно схеме SemVer), и всегда соответствовать упакованной в образ функциональности приложения.

В некотором смысле все, что мы сказали о метке `latest` относительно наших собственных образов, верно и для базовых образов, которые мы указываем с помощью инструкций `FROM`. К примеру, какая версия Ubuntu или Alpine будет использована в инструкции `FROM ubuntu|alpine`? Последняя, но совершенно неизвестно какая именно! Указывая точные версии вместо неопределенной версии, мы увеличиваем стабильность и предсказуемость своих образов.

Альтернативы Dockerfile. Jib.

Контейнеры стали заслуженно популярны, и мы видим, что построить образы для них несложно. Однако не все языки программирования одинаково хороши для работы с многоступенчатыми инструкциями `Dockerfile`, даже если использованы все рекомендованные практики для построение образов минимальных размеров и с минимальными зависимостями.

Яркий пример - приложения и сервисы Java, и связанные с JVM языки, такие как Scala и Kotlin. Практически все они используют системы сборки Maven, Gradle, и похожие на них (SBT), и все свои зависимости (библиотеки JAR) хранят и скачивают с центральных хранилищ, обычно Maven Central.

Написав свой многоступенчатый образ для Java, мы тем не менее полностью игнорируем кэш и локально доступные, уже скачанные библиотеки JAR. Это неизменная зависимость приложения, подписанная и надежно защищенная от изменений самим механизмом Maven Central. Мы же заново, раз за разом, полностью скачиваем все зависимости приложений через Интернет, делая процесс сборки приложения медленным и неэффективным.

Один вариант - просто скопировать весь кэш Maven или Gradle внутрь контейнера по время сборки приложения, но это опять же неэффективно - это могут быть тысячи библиотек, используемых другими приложениями.

Здесь поможет плагин Jib, специально созданный Google для оптимизации сборки образов Java-приложений. Он интегрируется с системой сборки Maven или Gradle, анализирует список зависимостей, и создает специальный кэш (в мире Docker это часть образа называется слоем layer), который не меняется и используется заново каждый раз при последующей сборке проекта. Выигрыш в эффективности и скорости сборки образа потрясающий.

Добавим Jib в наш проект Java и Spring Boot - мы использовали там сборку Maven:

```
...
    <build>
        <plugins>
            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>jib-maven-plugin</artifactId>
                <version>1.8.0</version>
                <configuration>
                    <to>
                        <image>{учетная_запись_Docker}/java-hello:0.1.0</image>
                    </to>
                </configuration>

```

```
            </plugin>
...
```

Плагин объявлен в стандартной секции `build/plugins`. В конфигурации плагина мы также указываем стандартную метку для нашего образа - учетная запись, название и версия. Дальше остается только собрать образ нашего сервиса с помощью Jib:

```
$ mvn compile jib:build
...
[INFO] Total time:  13.127 s
```

Первая сборка займет некоторое время, но каждая последующая сборка будет очень быстрой и эффективной. Размер полученного образа `java-hello` также станет еще почти в два раза меньше - по умолчанию Jib использует сжатый базовый образ `distroless`, разработанный в Google. Вот что мы получили, использовав Jib:

- Нам больше не нужно писать, оптимизировать и поддерживать `Dockerfile`! Jib выберет базовый образ, соберет проект с помощью Maven/Gradle (это уже сделано нами), и оптимизирует кэш для сборки, используя все локально доступные зависимости и библиотеки JAR.
- Более того, Jib не требует наличия самого Docker! Это может быть удобно в системах непрерывной сборки и интеграции CI/CD и автоматических скриптах, так как установку Docker не всегда удобно делать автоматически.
- Автоматическая оптимизация размера полученного образа. Собственный базовый образ можно указать в конфигурации.
- Скоростная сборка образа - при изменении кода Jib построит новый образ, в котором будут изменена часть кэша (слой layer), отвечающая только за классы Java. Как правило, образ будет готов в течение секунд, а не минут. Это особенно удобно и выгодно при разработке с Kubernetes.

Java в некотором роде выделяется среди других языков из-за уже готового кэша библиотек JAR и стандартного формата практически всех проектов на основе Maven/Gradle, что и позволяет Jib успешно оптимизировать построение контейнеров.

Другие языки не всегда имеют подобные решения, но в качестве начальной точки можно рекомендовать проект Cloud Foundry Buildpacks - набор общих решений для полуавтоматической сборки образов контейнеров без обязательного наличия Dockerfile.

Резюме

- Контейнеры и Docker хороши для экспериментов, интеграционного тестирования, и запуска известных операционных систем и баз данных, но главная их задача - упростить

и сделать более гибкой разработку новых приложений и микросервисов. Поместить свое приложение в контейнер позволяет сборка образа image с помощью инструкций `Dockerfile`.

- Каждый новый образ контейнера строится на основе базового (base) образа. Выбор базового образа влияет на размер собранного образа с приложением, и количество доступных сервисов и инструментов.
- Каждый язык программирования предлагает набор стандартных базовых образов для компиляции, сборки и запуска написанных на нем приложений. Использовать в качестве базового образа Linux, и заново скачивать и устанавливать все необходимые компиляторы и пакеты как правило расточительно и неэффективно.
- Хорошей практикой является компиляция и сборка приложения прямо внутри временного контейнера во время работы команды `docker build`. Это позволяет получить "чистую" сборку на основе исходного кода, без случайных зависимостей от локальной системы.
- Многоступенчатый (multi-stage) `Dockerfile` позволяет собрать образ контейнера в несколько ступеней, значительно уменьшив количество ненужных инструментов в окончательном образе с приложением.
- По умолчанию версией в метке (tag) для образа контейнера является `latest`. Ее легко перезаписать, потеряв старый образ. Оптимальным вариантом является использование семантических версий для каждого, даже самого маленького изменения в функциональности приложения в образе.
- Существуют эффективные альтернативы `Dockerfile` - к примеру для приложений Java можно просто использовать уже имеющиеся у программистов знания Maven/Gradle, чтобы собирать эффективные образы с помощью плагина Jib.

5. Первые шаги в Kubernetes

> “Kubernetes начинает играть роль операционной системы Linux для облаков Cloud”.
> Джим Землин (Jim Zemlin), директор фонда Linux Foundation

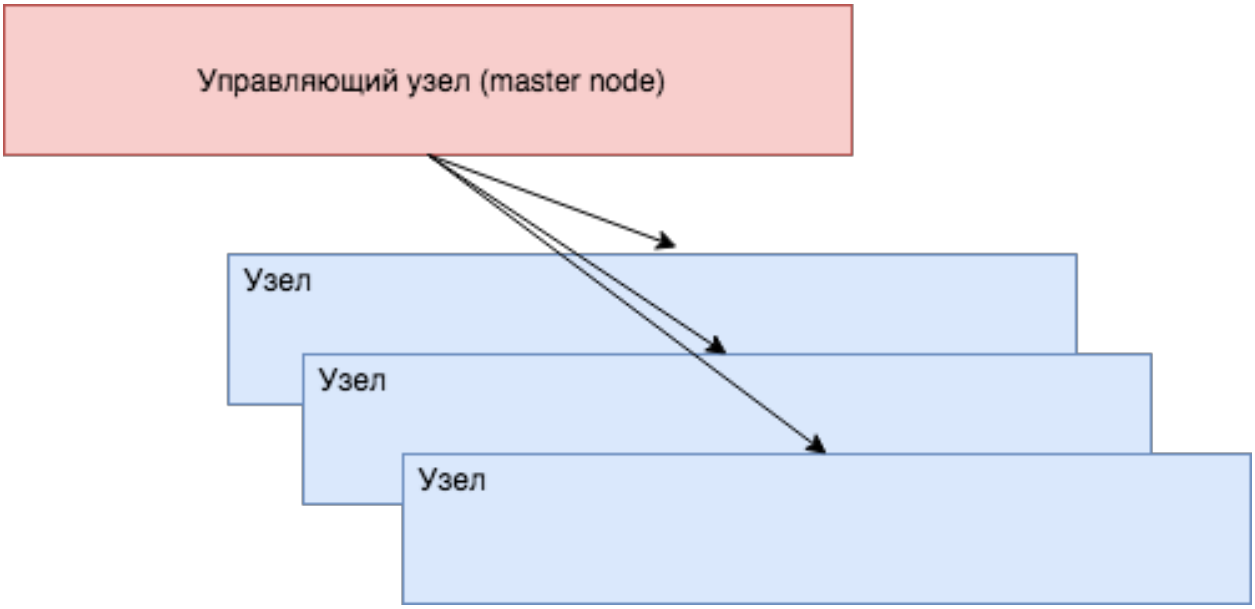
Kubernetes - на данный момент самый гибкий и ставший очень популярным ответ на важнейший вопрос глобальных приложений, способных со временем привлечь миллионы пользователей - как быстро развертывать, обновлять и масштабировать модули и сервисы моего приложения в облаке, эффективно используя всю доступную мне вычислительную мощь? Контейнеры (и их основной инструмент Docker) великолепно справляются с задачей универсальной подготовки и упаковки модулей приложения и сервисов для исполнения на любых серверных версиях Linux. Запуск разнородных приложений становится простой задачей. Остается управлять развертыванием и масштабированием. Неважно, доступен ли вам поначалу лишь один сервер или вообще ваш личный лэптоп, если ваша идея удачна, а реализация хороша, вы привлечете пользователей, и Kubernetes поможет вам развернуть, а затем масштабировать приложение для миллионов практически так же быстро, как если бы им пользовались только несколько человек.

Кстати, о произношении слова - произносится примерно как Кюбэрнэтис, ударение на третий слог. Происхождение слова греческое, κυβερνήτης, управляющий, кормчий корабля, или командир, и между прочим, очень близкое нам - мы заимствовали это слово также и для русского языка - посмотрите на слово губернатор. Заманчиво было бы назвать всю технологию “губернатор”, но это чересчур.

Итак, давайте начнем с азов, терминологии Kubernetes, которая на удивление проста и логична - что без сомнения является одной из причин успеха всей технологии. Ваше приложение будет работать в кластере (cluster) - группе однородных вычислительных ресурсов, если говорить в общих терминах. В большинстве случаев это стойка с Linux/UNIX-серверами одного типа (server rack). В других случаях это может быть набор специализированных ресурсов, например основанных на чипах GPU (графических процессоров, особенно подходящих для специализированных вычислений - как здесь не вспомнить волну популярности криптовалют или машинного обучения). В любом случае каждому из этих ресурсов надо выделить свою часть работы и запустить ее.

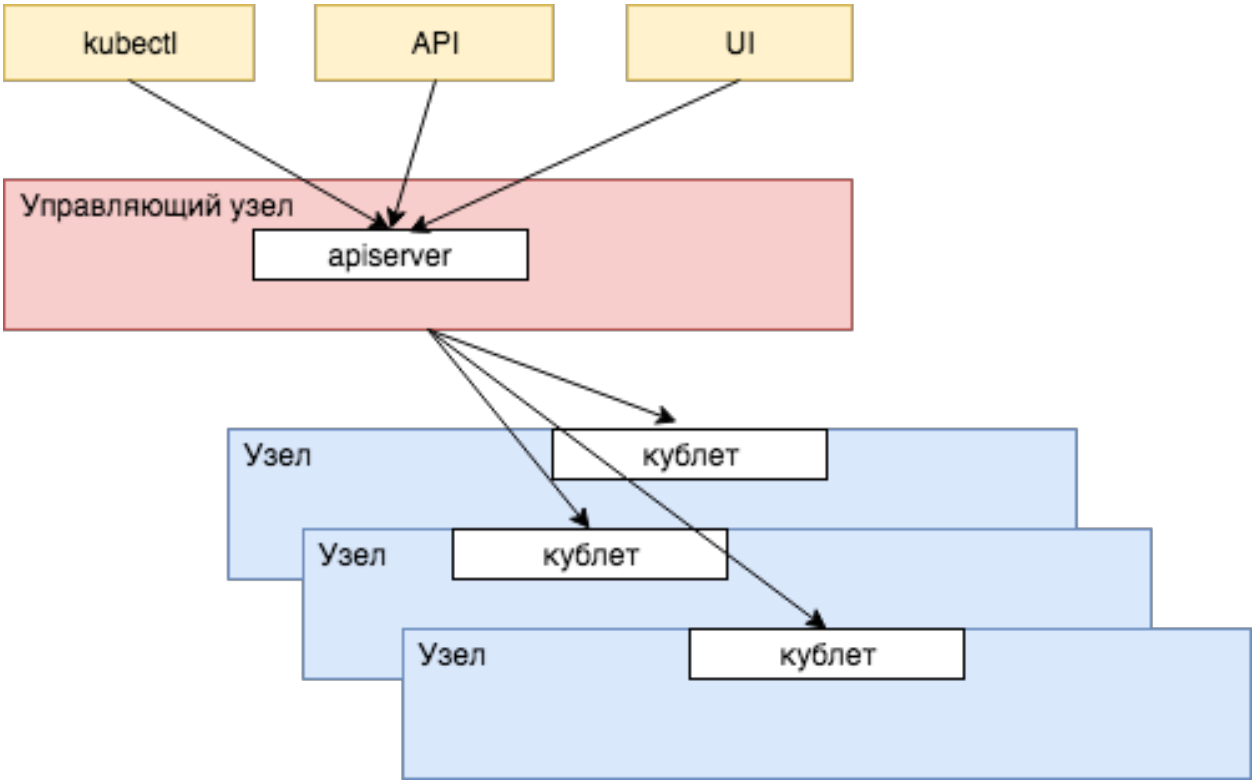
Элемент кластера, то есть сервер Linux или похожий на него ресурс, будет называться узлом (node, cluster node). Раньше этот узел часто называли подчиненным или ведомым (slave), но в Kubernetes намеренно от этого отказались, так как звучать подобное название может не слишком приятно.

Главный пункт управления всеми процессами и масштабированием находится в управляющем узле (master node).



Управляющий узел Kubernetes и будет заведовать распределением задач по доступным ему вычислительным ресурсам подчиненных узлов и держать с ними связь. Коммуникацией между управляющим и остальными узлами заведует небольшой процесс, работающий на каждом узле - кублет (kubelet, формально можно перевести как мини-управляющий. Мы привыкли подобные слова использовать без перевода - сервлет как мини-сервер, апплет как мини-приложение).

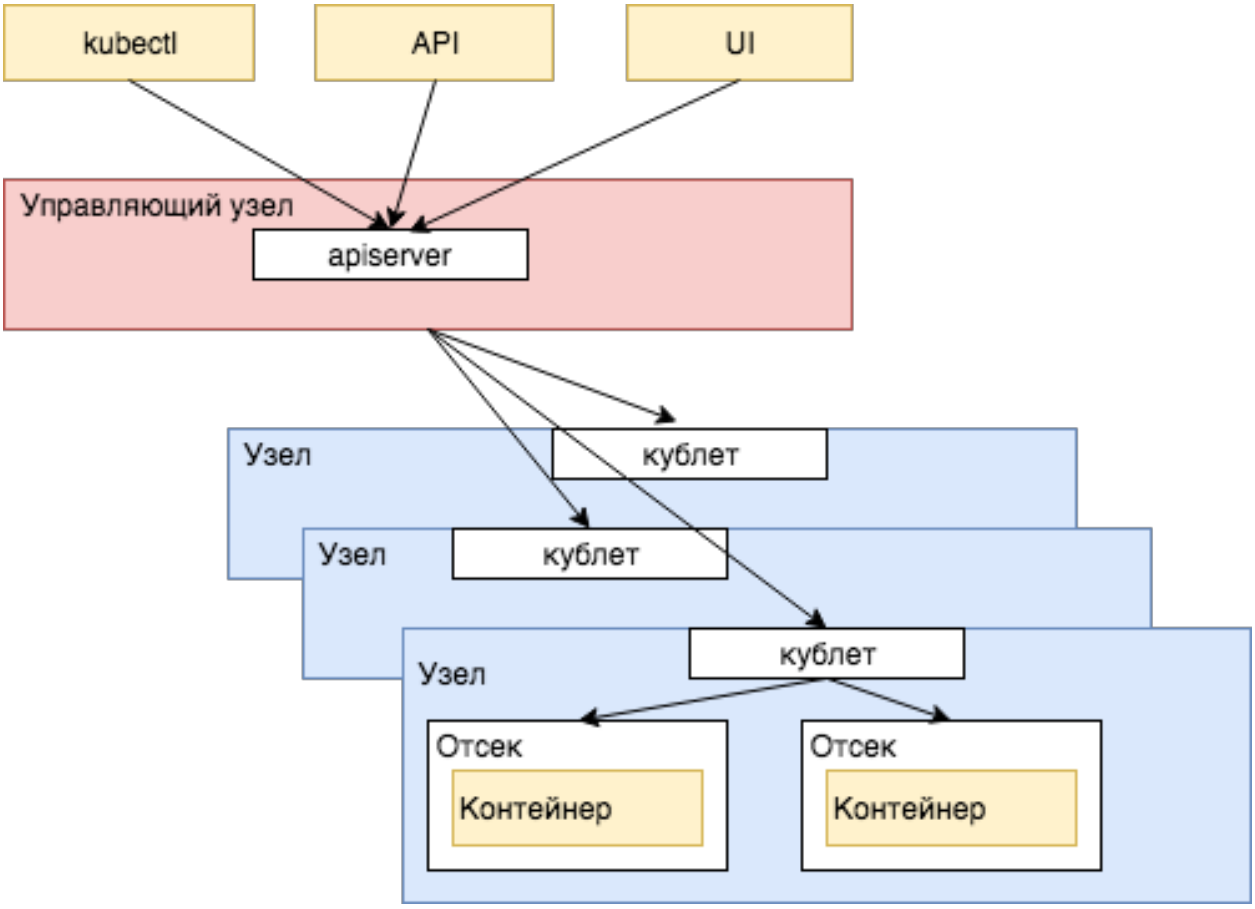
Сам управляющий узел будет получать команды и задачи уже от нас, с помощью предоставляемого им интерфейса API, работающего в процессе под довольно очевидным названием apiserver. Существует множество способов общения с управляющим узлом кластера Kubernetes - самый известный это через командную строку с помощью команды kubectl (Произносится как кюб-контрол. Часто произносят просто буквы английского алфавита, кюб-си-ти-эл), через множество вариантов пользовательских интерфейсов (UI), ну и наконец непосредственно вызывая сам сервер через его точки доступа. Команды обычно передаются в формате JSON и YAML.



Как мы теперь хорошо знаем, запускать и настраивать множество разнородных приложений на одном физическом сервере (или одной виртуальной операционной системе) непросто, и это часто приводит к путанице и ошибкам с версиями, путями, библиотеками и инструментами. Все это блестяще решают контейнеры. Нетрудно догадаться, что вместо того чтобы запускать на своих вычислительных узлах процессы или установку напрямую, в мире Kubernetes все работает исключительно в контейнерах.

Все ваши сервисы, микро или чуть больше, или непосредственно приложения, будут работать в контейнерах, размещенных в узлах кластера. Как правило контейнеры будут работать под управлением Docker, но Kubernetes поддерживает и другую систему управления под названием rkt.

Есть только одно важное отличие - контейнеры будут работать не в самом узле, а в своем небольшом пространстве, называемом отсеком (pod). В отсеке могут выполняться несколько контейнеров одновременно. Контейнеры внутри отсека имеют более тесную связь и главное, разделяют пространство своих процессов, память и тома с данными, имеют одинаковый IP адрес, а также могут связываться через локальный адрес localhost. Отсеки узла разделены друг от друга также хорошо, как и обычные контейнеры. Можно сразу сказать, что обычно на один отсек приходится один контейнер, и только в некоторых случаях к основному контейнеру с главным сервисом или приложением присоединяют вспомогательный - например для отправки журналов (logs), или настройки сети.



Первые выводы

- Kubernetes работает с кластером - группой серверов или похожих вычислительных ресурсов, с помощью которых он будет исполнять ваши сервисы или приложения.
- Главная единица сервиса или приложения, которая будет исполняться на кластере - это образ (image) контейнера, в котором содержится полное описание приложения, его зависимости, инструменты и библиотеки, открытые порты и требуемое хранилище, а также команды, необходимые для запуска. Кроме этого образа, Kubernetes ничего не знает про приложение.
- Контейнеры запускаются на узлах кластерах в отдельных пространствах, называемых отсеками.
- Один из узлов в кластере является управляющим и управляет запуском и масштабированием контейнеров на всех остальных узлах.
- На каждом узле кластера установлен агент Kubernetes, так называемый кублет. Он получает команды от управляющего узла и запускает, останавливает и проверяет состояние отсеков с контейнерами. Только узлы, на которых работает кублет, доступны для управления Kubernetes.

- Наконец, сам управляющий узел получает команды от вас - с помощью командной строки и команды kubectl, или через разнообразные варианты пользовательских интерфейсов, созданных для управления Kubernetes. Командой как правило является развертывание образа вашего контейнера, с описанием его точек доступа и желаемым уровнем и способом масштабирования - например, автоматическим в зависимости от загрузки или сразу с определенным количеством работающим экземпляров на разных узлах и отсеках.

Первое развертывание

Пока все достаточно очевидно и логично - Kubernetes это система развертывания и масштабирования, которая запускает ваши сервисы и приложения из образов, в которые они упакованы, развертывает и запускает их по всему пространству ресурсов, доступных в кластере, и далее следит за работающими экземплярами. Часто говорят, что подобная система занимается оркестровкой контейнеров (orchestration).

Особенно удобно это становится когда речь идет о приложениях, работающих в облаках и предоставляемых ими кластерах. В своем собственном физическом кластере, который вам полностью принадлежит, вы можете опробовать разные способы развертывания и управления своими сервисами и приложениями, экспериментировать с серверами и их настройками, но на общих, разделенных вычислительных ресурсах облака, к которым у вас весьма ограничен доступ, общая система управления Kubernetes особенно хороша.

Первое, что нам понадобится для работы с любыми ресурсами и API Kubernetes - интерпретатор команд Kubernetes `kubectl`. Часто он устанавливается вместе с пакетами для разработки программ самых распространенных провайдеров публичных облаков Google, Amazon или Microsoft, но если вы только начинаете, работаете с локальным кластером, или предпочитаете сразу установить `kubectl` на своем собственном компьютере самостоятельно, зайдите за инструкциями по установке на официальный сайт Kubernetes `kubernetes.io`. Установка с помощью менеджера пакетов вашей операционной системы или с помощью прямого вызова командой `curl` не должна вызвать у вас особенных затруднений.

Далее нам нужен будет непосредственно кластер Kubernetes, то есть набор вычислительных ресурсов единого типа, с установленным на ними управлением Kubernetes, что, как мы уже успели увидеть, будет включать в себя кублеты, управляющий узел, и разнообразные API.

Самый простой вариант на данный момент - предоставляемый Docker минимальный кластер Kubernetes для экспериментов и тестирования. Как правило вы уже работаете с инструментом Docker для построения и управления контейнерами на своей машине. Если эта машина не Linux (Mac или Windows), это означает, что за кулисами Docker уже запустил минимальную виртуальную машину Linux для поддержки контейнеров. Новые версии Docker могут запустить на этой же виртуальной машине кластер Kubernetes, состоящий из одного узла.

Второй распространенный и доступный вариант - попробовать установить локальный эмулятор Kubernetes под названием `minikube` на свой собственный сервер или рабочую машину. `minikube` поддерживает все популярные операционные системы. Для этого пригодится открытый проект на Github `kubernetes/minikube`, там же (а также на сайте книги `www.ipsoftware.ru`) вы найдете подробные инструкции об установке. Установка будет чуть сложнее чем для `kubectl`, придется еще выбрать драйвер виртуализации для запуска и поддержки виртуальной машины, на которой будет запущен ваш локальный узел Kubernetes.

Итак, будем считать, что с первоначальной настройкой кластера мы быстро разобрались, и теперь можно получить первую информацию о структуре нашего кластера с помощью команд `kubectl config current-context` (получит название контекста `kubectl`, то есть набора адресов, управляющего узла, и остальных данных о текущем кластере), `kubectl cluster-info` (сетевые адреса управляющего кластера) и `kubectl get nodes` (узлы кластера).

Вот что мы получим для кластера `minikube`:

```
$ kubectl config current-context
minikube
$ kubectl cluster-info
Kubernetes master is running at https://192.168.64.3:8443
KubeDNS is running at
https://192.168.64.3:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
$ kubectl get nodes
NAME       STATUS    ROLES     AGE       VERSION
minikube   Ready     master    6d        v1.10.0
```

Как легко увидеть, по умолчанию `minikube` запускает единственный узел - он будет и управляющим, и использоваться для размещения всех развертываний, что конечно, неоптимально, но вполне подходит для локального тестирования и отладки.

Если вы используете в качестве кластера виртуальную машину Docker, получится примерно следующее:

```
$ kubectl config current-context
docker-for-desktop
$ kubectl cluster-info
Kubernetes master is running at https://localhost:6443
KubeDNS is running at
https://localhost:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
$ kubectl get nodes
NAME                 STATUS    ROLES     AGE       VERSION
docker-for-desktop   Ready     master    10m       v1.10.3
```

Здесь все похоже на `minikube`, но естественно другое название для контекста `kubectl` и как видно, Docker использует адрес `localhost` вместо виртуального сетевого адреса, но суть

примерно такая же, и по умолчанию под управлением Kubernetes в кластере имеется только один узел.

Если вы создадите кластер в настоящем облаке, например Google Cloud, назовете проект для него `hello-cluster`, и для начала будете использовать три узла, получится следующая картина:

```
$ kubectl get nodes
NAME                                          STATUS    ROLES     AGE       VERSION
gke-hello-cluster-default-pool-b36cb657-hw5j   Ready     <none>    20h       v1.8.7-\
gke.1
gke-hello-cluster-default-pool-b36cb657-qwff   Ready     <none>    20h       v1.8.7-\
gke.1
gke-hello-cluster-default-pool-b36cb657-r4z2   Ready     <none>    20h       v1.8.7-\
gke.1
```

Как видно, в нашем кластере три рабочих узла, все они содержат название самого кластера (`hello-cluster`) как часть своего имени. Вы получите примерно такой же результат в своей консоли.

В качестве сервиса, который мы будет развертывать и масштабировать на нашем кластере, возьмем микросервис (скорее даже наносервис в нашем случае, но это простой пример), сообщающий текущее время.

Напишем его на Go:

```
package main

import (
        "encoding/json"
        "log"
        "net/http"
        "strconv"
        "time"
)

type Time struct {
        Time     string `json:"time,omitempty"`
        NanoTime string `json:"nanoTime,omitempty"`
}

func main() {
        log.Print("Начало работы сервиса time-service")

```

```
19          http.HandleFunc("/time", serveTime)
20          http.HandleFunc("/nanotime", serveNanoTime)
21          log.Fatal(http.ListenAndServe("0.0.0.0:8080", nil))
22  }
23
24  func serveTime(w http.ResponseWriter, r *http.Request) {
25          log.Print("Вызов функции serveTime()")
26          var serverTime Time
27          serverTime.Time = time.Now().String()
28          json.NewEncoder(w).Encode(serverTime)
29  }
30
31  func serveNanoTime(w http.ResponseWriter, r *http.Request) {
32          log.Print("Вызов функции serveNanoTime()")
33          var nanoTime Time
34          nanoTime.NanoTime = strconv.FormatInt(time.Now().UnixNano(), 10)
35          json.NewEncoder(w).Encode(nanoTime)
36  }
```

Как видно, мы создаем две точки доступа к нашему сервису - он способен сообщать время как в формате строки (формат выбран в реализации Go по умолчанию и довольно подробен), или в виде числа наносекунд стандартного времени эпохи Unix, в стандартной библиотеке Go это время возвращает функция `UnixNano()`. Для HTTP сервера Go, который будет работать в контейнере, предпочтительнее указать общий адрес для всех локальных сетевых интерфейсов `0.0.0.0` напрямую, но это малозначительная особенность, которая во многих случаях или в других средах запуска контейнеров может быть и не нужна, но и вредной не будет.

Результат сервис будет возвращать в виде JSON, поддержка этого формата реализована в Go в стандартном пакете `encoding/json`. Для простоты реализации оба варианта получения времени используют в качестве возвращаемого значения одну и ту же структуру `Time`, но любое неиспользуемое поле не будет записываться в JSON - это позволит сделать дополнительная аннотация `omitempty`.

Мы также используем стандартное решение Go для записи журналов (logs) доступа к нашему сервису - пакет `log`. Без дополнительной настройки журналы будут записываться в стандартный вывод приложения, и как мы увидим чуть позже, это именно то, что нужно для сбора журналов в Kubernetes.

Далее мы создадим файл сборки образа контейнера Dockerfile, конечно же, применив многоступенчатую сборку для уменьшения размера нашего образа:

```
# Собирает образ для контейнера с сервисом получения времени.
FROM golang:1.13 as builder

# Изменим рабочую директорию
WORKDIR /opt/time-service

# Скопируем код программы в файловую систему контейнера
COPY main.go .

# Соберем программу из исходного кода, в файл time-service
# Необходимо указать дополнительные флаги сборки Go
RUN CGO_ENABLED=0 GOOS=linux go build -a -o time-service main.go

# вторая ступень - спартанская версия Linux Alpine
FROM alpine:3.10

# номер порта, используемого контейнером
EXPOSE 8080
# Используем такую же рабочую директорию
WORKDIR /opt/time-service

# Скопируем собранный бинарный код из первой ступени
COPY --from=builder /opt/time-service/time-service .

CMD [ "/opt/time-service/time-service" ]
```

Здесь все совсем просто, благодаря отличной поддержке языка и среды Go среди стандартных образов открытого репозитория Docker. По сути мы следуем уже полученному нами рецепту из прошлой главы.

Нам нужно лишь выбрать версию Go - мы берем последний большой выпуск 1.13, используем порт 8080, и копируем наш файл в образ. Директорию мы выбираем с тем же именем, с которым этот файл лежит у нас в Github в репозитории с примерами книги, и именно это имя мы будем использовать как имя образа. Название директории также будет использовано по умолчанию компилятором Go в качестве имени скомпилированного файла. Остается скомпилировать сервис и указать Docker, что при запуске экземпляра контейнера мы запускаем наш сервис, с помощью команды `CMD`.

Остается только собрать наш образ, указав его метку для последующего хранения в открытом репозитории Docker Hub или любом другом репозитории, и наш сервис готов к развертыванию и запуску. Применим семантическую версию 0.1.0 - это самый первый вариант сервиса, еще далекий от готовности.

```
$ docker build . -t {учетная_запись_Docker}/time-service:0.1.0
```

Протестируем наш образ и сервис обычным образом на локальной машине и убедимся что все собрано и настроено верно:

```
$ docker run -d -p 8080:8080 {учетная_запись_Docker}/time-service:0.1.0
curl localhost:8080/nanotime
curl localhost:8080/time
```

Запустив `curl`, вы увидите что именно вам отвечает сервис. Если что-то не срабатывает, дело как правило в настройках сетевых портов между локальной машиной и контейнером. Не забудьте остановить запущенный контейнер (`docker stop` или просто Ctrl-C) после того, как закончите свои тесты.

Теперь наступает самое интересное - развернем наш сервис на новом кластере. Чтобы управляющая среда Kubernetes понимала, откуда ей нужно взять образ контейнера с нашим сервисом, нужно разместить его в доступном ей репозитории образов контейнеров. Проще и популярнее всего создать аккаунт в открытом репозитории Docker Hub, или загрузить его в репозиторий контейнеров, который предоставляют все популярные провайдеры облачных услуг (это будет ваше закрытое личное хранилище образов в облаке).

Поступим наиболее простым способом и загрузим образ в репозиторий Docker Hub.

```
$ docker push {учетная_запись_Docker}/time-service:0.1.0
```

Здесь часть `{учетная_запись_Docker}` нужно будет заменить на свою учетную запись, перед исполнением команды `docker push`.

Дальше нам нужно вызвать `kubectl run` и развернуть наш сервис из образа, а после развертывания получить его параметры и внешний адрес в Интернете.

```
$ kubectl run time-service --image {ваша_учетная_запись_Docker}/time-service:0.1.0
```

Как мы видим, параметры команды весьма очевидны, это большое преимущество всей инфраструктуры Kubernetes - практически все термины и команды инструментов продуманы и логичны, что делает кривую обучению намного проще. Итак, в нашем случае мы разворачиваем сервис `time-service`, это же имя будет использовано как название для всего развертывания (deployment), берем для него образ с контейнером, в котором построен и подготовлен этот сервис, и сообщаем управляющему узлу, что сервис будет пользоваться портом 8080.

Управляющий узел и его агенты (кублеты) возьмут на себя всю работу по подготовке окружения для нового сервиса - выберут один из узлов (скорее всего наименее загруженный, но это особенности внутренней реализации Kubernetes), если необходимо, запустят систему

управления контейнерами (скорее всего основанную на Docker), настроят сетевые адреса и порты, и наконец, загрузят образ и запустят экземпляр нашего контейнера. Нам, кроме короткой строки в консоли, не потребовалось сделать для этого ничего.

В итоге Kubernetes создает так называемое развертывание (deployment) для нашего сервиса. Это один из основных объектов в среде Kubernetes, описывающий, что именно за образы были запущены в отсеках, и какие параметры им были указаны. Чуть позже мы подробнее узнаем об управлении развертываниями. С нашей же точки зрения, Kubernetes взял наш микросервис (или монолитное приложение, это по сути не так важно), и развернул его на вычислительных ресурсах кластера, находящегося под его управлением.

Наш микросервис для получения времени был запущен простой командой `kubectl run`, поэтому по умолчанию он работает в единственном экземпляре, в созданном для него отсеке. Посмотреть детали работающих на данный момент отсеков в кластере можно следующим образом:

```
$ kubectl get pods -o wide
NAME                            READY     STATUS    RESTARTS   AGE       IP          \
  NODE
time-service-7c886f94bf-gwk4x   1/1       Running   0          6m        172.17.0.4 \
  minikube
```

Мы использовали расширенную форму команды (`-o wide`), чтобы увидеть на каком узле развернут наш отсек. Легко видеть, что отсек создан для нашего сервиса - его имя содержит названия сервиса (`time-service`). Чтобы убедиться что именно в этом отсеке работает наш контейнер, и получить полную информацию о нем, выполним команду `describe`, указав имя отсека:

```
$ kubectl describe pod time-service-7c886f94bf-gwk4x

Name:           time-service-7c886f94bf-gwk4x
Namespace:      default
Node:           minikube/192.168.64.3

...
    Image:          {ваша_учетная_запись_Docker}/time-service
    Port:           8080/TCP
    State:          Running
...
```

Как легко убедиться, мы видим, на каком узле работает отсек, и увидим список контейнеров и использованных для их запуска образов - в нашем случае это будет единственный контейнер с сервисом получения времени.

Итак, несколькими простыми командами мы развернули наш сервис в облачном кластере. Сервис работает в контейнере из предоставленного и собранного нами образа, и внутри этого контейнера мы управляем всеми настройками и окружением. В случае проблем с кластером, любыми его узлами, оборудованием или дисками, отказом сетей, ошибкой в самом сервисе и контейнере, управляющий узел Kubernetes немедленно запустит новые узлы, передаст их в наш кластер, создаст новые отсеки и запустит столько контейнеров с сервисами, сколько необходимо будет для удовлетворения наших требований к системе. Если вспомнить, что сервис умещается в несколько строк кода, и при разработке мы не на секунду не задумывались о любых из описанных сценариев распределенных сервисов, популярность Kubernetes становится оправданной. Более того, если вспомнить, что провайдер облака, такой как Amazon или Google, также обеспечит безопасность и постоянное обновление сервера для вашего сервиса, отказоустойчивость и поддержку команды инженеров, работа в облаке выглядит более чем привлекательной.

Сервисы Kubernetes - точка доступа к масштабируемому приложению

После этого этапа наш микросервис запущен, но никакого доступа к его портам у нас нет. Мы запустили в своем приложении минимальный веб сервер для обработки запросов по получению времени, и используем для него порт 8080 - но по умолчанию при развертывании контейнера любые открытые и используемые порты доступны лишь внутри пространства самого отсека (pod), который изолирован и от самого сервера, на котором он работает (то есть узла в терминологии Kubernetes), и даже от других отсеков на этом узле.

В мире Kubernetes доступ к портам работающих приложений открывается и управляется через сервисы (services). Сервис - это служебный объект Kubernetes с настройками, которые позволяют управляющей системе кластера понять, какие порты открываются приложением, как обеспечить к ним доступ, и в каких отсеках находятся работающие экземпляры приложения, особенно в том случае когда приложение масштабировано и работает во множественных экземплярах, каждый из которых находится в своем отсеке.

Для создания сервиса, с помощью которого мы будем получать доступ к нашему развертыванию и портам своего приложения, вызовем команду `expose`. Ей достаточно указать, какое развертывание мы собираемся открывать, и какой порт нужно будет открыть. По умолчанию сервис будет доступен только внутри кластера (что конечно же имеет смысл для взаимодействия множественных сервисов, работающих в одном кластере), но нам хотелось бы попробовать его в деле прямо сейчас. В этом нам поможет более расширенная версия сервиса Kubernetes, с названием `NodePort`, создадим мы ее следующим образом:

```
1 $ kubectl expose deployment time-service --port=8080 --type=NodePort
2 service "time-service" exposed
```

Сервис с типом “NodePort” создает прокси-доступ к нашему сервису и его открытому порту 8080 на каждом узле кластера, так что мы можем отправить к нему запрос, если у нас есть доступ к какому-либо узлу. Прежде чем сделать это, давайте посмотрим, какой порт теперь будет использоваться для нашего сервиса, посмотрев краткий список сервисов:

```
$ kubectl get services
NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP          1d
time-service   NodePort    10.110.186.179   <none>        8080:30130/TCP   3m
```

Команда `get services` получит список всех доступных в кластере сервисов, включая сам управляющий сервис Kubernetes. Мы увидим, что наш сервис был создан с таким же именем, как и само развертывание (`time-service`), и в колонке портов указан порт, через который мы теперь можем получить доступ к нашему сервису - 30130 (в вашем случае номер может быть другим, по умолчанию из диапазона, который вы можете найти в документации или исходном коде Kubernetes).

Для локальных кластеров `minikube` и `docker` у нас есть прямой доступ к нашему виртуальному кластеру, и мы сможем вызвать свой сервис через полученный прокси-порт. Для локального кластера Docker управляющий узел доступен прямо на локальном адресе localhost:

```
$ curl localhost:30130/time
{"time":"2018-09-26 16:26:10.4901133 +0000 UTC m=+227950.182386538"}
```

В случае `minikube` виртуальная машина имеет отдельный сетевой адрес, и узнать его адрес проще всего, используя встроенную команду `minikube service --url`, указав имя сервиса, доступ к которому мы хотели бы получить:

```
$ minikube service --url time-service
http://192.168.64.3:30130
$ curl http://192.168.64.3:30130/time
{"time":"2018-09-26 16:28:12.6901133 +0000 UTC m=+227950.182386538"}
```

Доступ к сервису из Интернета - балансировщик нагрузки

Если вы используете кластер одного из публичных провайдеров облака, и все ваши отсеки и развертывания находятся на удаленных серверах и виртуальных машинах, получить доступ к узлам этого кластера из Интернета не получится, если только вы не используете

дополнительную аутентификацию. Для получения доступа к сервису извне, из “большого Интернета”, например, чтобы предоставить доступ к сервису всем пользователям или внешним элементам нашей системы (чаще всего это пользовательский интерфейс UI), мы можем использовать доступный во всех провайдерах облака балансировщик нагрузки (load balancer):

```
$ kubectl expose deployment time-service --type "LoadBalancer"
```

Команда здесь также проста - мы просим открыть во внешний Интернет (expose) наш сервис, и указываем, что все запросы к нему должны будут проходить через встроенный в Kubernetes балансировщик нагрузки - мы мгновенно получаем самую распространенную схему распределения вычислительной нагрузки и потенциальных запросов к нашему сервису в кластере, не заботясь о мелочах и деталях. Балансировщик нагрузки требует публичного IP-адреса, поэтому в локальных кластерах недоступен.

К этому моменту у нас окончательно все развернуто и готово, мы можем показать параметры и внешний адрес сервиса:

```
$ kubectl get service time-service
NAME           TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
time-service   LoadBalancer   10.63.254.246   <внешний адрес>   8080:30875/TCP   18h
```

Через некоторое время в столбце `EXTERNAL-IP` появится внешний адрес нашего сервиса, и мы сможем получить к нему доступ из любой точки мира. Конечно, большая часть микросервисов стандартных приложений будет исключительно для внутреннего использования и вспомогательными, и во внешний мир мы будем выставлять конечную точку вводных данных от пользователей или других приложений, как правило веб-сервер, такой как nginx, или доступную для внешних потребителей версию доступа к интерфейсу REST, со всеми предосторожностями и соображениями безопасности, обязательными для сервисов, напрямую доступных через Интернет.

Отладка сервисов - переадресация портов

Если вы развернули сервисы вашего приложения на публичных облаках, в удаленных центрах данных, прямого доступа к узлам кластера, на которых развернуты отсеки Kubernetes и контейнеры с вашими сервисами, у вас нет, если только вы не запросите для этих узлов публичные IP-адреса, что, как правило, связано с дополнительными затратами. Тем не менее, необходимость проверить работоспособность сервисов, и отладить (debug) их функциональность в реальном кластере, а не локальной версии minikube или Docker, возникает постоянно. Использование балансировщика нагрузки для отладки своих сервисов также связано с вопросами безопасности и стоимости - доступный в Интернете IP-адрес, даже кратковременно и для отладки, неминуемо привлечет внимание, и незащищенный

доступ к сервису, обладающему доступом к чувствительным данным, особенно опасен. Балансировщики нагрузки основных провайдеров облака (Amazon, Azure и Google) к тому же как правило довольно дороги.

Как раз для такого случая в Kubernetes предусмотрена временная переадресация портов, с помощью все той же команды `kubectl`. Вместо того чтобы открывать сервис всем опасностям Интернета, вы можете временно получить доступ к любому порту любого сервиса через управляющий кластер, к которому у вас всегда есть доступ (иначе управлять кластером просто не получится). Делается это простой командой:

```
$ kubectl port-forward service/time-service 8080
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
.. вызываем сервис time-service через curl localhost:8080/time ..
Handling connection for 8080
...
```

Как видно, мы просто указываем, какой порт (оригинальный порт, открытый контейнером, в котором находится наш сервис) нужно переадресовать на нашу локальную машину. По умолчанию порт на локальной машине будет совпадать с портом контейнера. Каждый раз при доступен к порту через переадресацию команда будет печатать диагностическое сообщение. Если часто используемые порты на вашей машине уже заняты (как используемый нами 8080), можно указать, на какой порт будет осуществляться переадресация - просто укажите его еще одним параметром, перед портом контейнера, через двоеточие:

```
$ kubectl port-forward service/time-service 9999:8080
Forwarding from 127.0.0.1:9999 -> 8080
.. вызываем сервис time-service через порт 9999
```

Переадресация работает, пока активен запущенный процесс `kubectl port-forward`. Как только процесс заканчивает работу, заканчивается и переадресация.

Доступ к журналам (logs)

После начала работы приложения или сервиса основным способом узнать о его состоянии, и об истории проделанных им операций являются журналы (logs), названные так по сходству с судовыми журналами, в которых команда отслеживала свой путь и встречи. Даже если приложение выглядит простым и идеальным, и пусть даже оно полностью покрыто тестами всех возможных видов (хотя, вероятно, это фантастика), встреча с суровой реальностью, при работе систем под реальной нагрузкой, приводит к неожиданным результатам. Отследить, что произошло с приложением, позволяют подробные журналы.

Классические приложения и веб-серверы, до мира контейнеров и облака, тщательно записывали свои приключения в файлы журналов, разбивали их по размерам и датам, чтобы избежать появления гигантских файлов, не помещающихся в память; иногда сжимали в архивы. В мире контейнеров этот подход работать не будет, по крайней мере без серьезных манипуляций и настроек. Как мы помним, контейнер запускается из предварительно созданного и настроенного образа, в котором нет воспоминаний о предыдущих запусках и нет места для уже собранных ранее журналов. Файловая система контейнера исчезает вместе с завершением его работы, если только не был настроен том для данных. Мир Kubernetes подразумевает динамику и простое управление развертываниями, в том числе увеличение и уменьшение количества работающих контейнеров. Их перезапуск также происходит в случае обновления образа до новой версии, отказа или фатальной ошибки. Во всех этих, часто случающихся в Kubernetes, ситуациях, записанные в файлы журналы исчезнут без следа.

Основным решением для журналов в Kubernetes, и по большому счету в контейнерах в целом, является запись в стандартный вывод приложения (standard output). При запуске приложения из командной строки или терминала стандартный вывод обычно выводится на консоль (console). Все что записывается в стандартный вывод контейнером в своем отсеке, далее можно получить простой и логичной командой `kubectl logs`, указав или развертывание, или индивидуальный отсек, или даже контейнер в нем, в случае если их там несколько. Наш сервис разумно поддерживает журналирование для каждого вызова, используя стандартный пакет Go `log`, и по умолчанию журналы попадают именно в стандартный вывод. Проверим:

```
$ kubectl logs deployment/time-service
2018/11/11 22:12:26 Вызов функции serveTime()
2018/11/11 22:15:44 Вызов функции serveNanoTime()
```

Если вызвать функции сервиса, а затем получить его журналы (указав, что мы получаем журналы своего развертывания `deployment/time-service`), мы увидим историю вызовов. Если вы хотите посмотреть не только последние записи, а отследить журнал в динамике, используйте ключ `-f, --follow`, чтобы оставить команду `kubectl logs` работающей в терминале и немедленно печатающей все журналы по мере их поступления:

```
$ kubectl logs deployment/time-service -f
```

Получение журналов контейнеров выглядит просто, но вызывает вопросы. Как собирать и анализировать журналы в течение длительного времени? Где они хранятся, если не в файловой системе, и насколько хватает этого хранилища? Что если мы хотим получать журналы не с одного контейнера, а с нескольких одновременно?

Журналы хранятся в отсеках (pods) Kubernetes, во временных файлах JSON, именно их содержимое мы получаем, когда вызываем команду `logs`. Для удобства обнаружения ошибок, или диагностирования отказа контейнера (который обычно происходит из-за проблем приложения или микросервиса, работающего в нем), предыдущий журнал контейнера также

сохраняется - мы можем получить его указав ключ `-p, --previous`. Однако на этом преимущества стандартного способа работы с журналами заканчиваются - если с отсеком или узлом что-то произойдет, все журналы будут потеряны. Проблемой является и избыточное журналирование, к примеру, при работе приложения в режиме отладки - может закончится все выделенное на отсеки и контейнеры дисковое пространство.

Сбор и управление журналами всегда непросты, поэтому Kubernetes оставляет детали и более функциональные решения внешним сервисам. Одни из самых широко распространенных решений для сбора и анализа журналов - Fluentd и набор решений ELK (Elasticsearch, Logstash и Kibana).

Впрочем, для этапа разработки и знакомства с Kubernetes, и по большому счету, для маленьких приложений, обычной работы с журналами вполне хватает, и мы будем ей пользоваться.

Простое горизонтальное масштабирование

Мы можем перейти к пожалуй главному оружию Kubernetes - легкому масштабированию. Простейшим способом горизонтального масштабирования является увеличение количества экземпляров сервиса - в случае Kubernetes увеличение количества отсеков с запущенными контейнерами, разумно распределенными по всему кластеру. Если мы точно знаем, сколько экземпляров сервиса нас устроит, этот способ достаточно хорош. Давайте увеличим количество экземпляров до трех:

```
$ kubectl scale --replicas=3 deployment/time-service
deployment "time-service" scaled
```

Как в общем случае с Kubernetes, необходимая команда логична - мы проводим “ручное” масштабирование, указываем нам нужны три экземпляра сервиса (`--replicas=3`), и указываем наше развертывание (`deployment/time-service`). В случае с масштабированием мы обращаемся к развертыванию (deployment), а не к сервису - именно развертывание следит за количеством отсеков и их работой.

Все команды, связанные с масштабированием, приводят к изменению желаемого состояния (desired state). Как мы знаем, главная задача управляющих механизмов Kubernetes - поддерживать желаемое состояние. Сразу после этой команды управляющий узел Kubernetes проведет все операции, необходимые для достижения нового желаемого состояния - запустит систему управления контейнерами на узлах кластера, создаст новые отсеки и приведем количество экземпляров сервиса к востребованному нами. Как мы уже видели, получить описание отсеков очень легко:

```
$ kubectl get pods -o wide
NAME                            READY     STATUS    RESTARTS   AGE       IP          \
  NODE
time-service-7c886f94bf-5nvnb   1/1       Running   0          10s       172.17.0.5 \
  minikube
time-service-7c886f94bf-gwk4x   1/1       Running   0          9m        172.17.0.4 \
  minikube
time-service-7c886f94bf-kxwc4   1/1       Running   0          10s       172.17.0.6 \
  minikube
```

Результат именно тот, что мы запрашивали - теперь наш сервис работает в трех отсеках, каждый отсек запущен и выполняет контейнер с сервисом. В случае работы с локальным кластером, в этом случае `minikube`, все отсеки были запущены на одном узле, и хотя в этом все равно есть смысл (в случае ошибки или сбоя одного контейнера остальные будут работать, можно постепенно увеличивать нагрузку на мощный узел), более наглядно увидеть это на кластере из множества узлов.

Если взять наш кластер в облаке Google Cloud и проделать то же действие масштабирования, и сделать так, чтобы на каждом узле кластера работало по экземпляру сервиса - в нашем случае это три узла - мы можем получить следующее:

```
$ kubectl get pods -o wide
NAME                            READY     STATUS    RESTARTS   AGE       IP          \
 NODE
time-service-7f864d67f4-9qdvh   1/1       Running   0          17m       10.60.2.6  \
 gke-hello-cluster-default-pool-b36cb657-hw5j
time-service-7f864d67f4-fvfhn   1/1       Running   0          17m       10.60.1.7  \
 gke-hello-cluster-default-pool-b36cb657-qwff
time-service-7f864d67f4-vdnl8   1/1       Running   0          19h       10.60.0.5  \
 gke-hello-cluster-default-pool-b36cb657-r4z2
```

Результат снова тот, что мы запрашивали - наш сервис работает в трех отсеках, но в этом случае отсеки запущены и выполняют контейнеры с сервисом на отдельных узлах (то есть отдельных, изолированных и независимых серверах) кластера. Знать что-либо самому сервису для этого совершенно не нужно.

Подобное “ручное” масштабирование может быть очень полезно - к примеру вы хотите чтобы сервис гарантированно был запущен на указанном количестве отсеков, и не хотите платить за больше количество в случае увеличения нагрузки. Но в большинстве случаев пригодится еще более впечатляющее оружие от Kubernetes - автоматическое масштабирование развернутого на кластере сервиса. Оно будет доступно в случае, если в кластере Kubernetes включены сервер метрик (`metric-server`), который будет поставлять информацию о загрузке процессора и памяти - обычно он включен в коммерческих облаках. В локальном кластере `minikube` метрики можно включить так:

```
$ minikube addons enable metrics
```

Чтобы передать Kubernetes ваше желание автоматически масштабировать сервис в зависимости от нагрузки на него, достаточно очевидных параметров - минимальное и максимальное количество экземпляров сервиса (и это отличный способ контролировать ваши расходы на вычислительные ресурсы облака), и порог нагрузки на вычислительный ресурс (обычно процент загрузки процессоров CPU), после которого необходимо подготовить и запустить дополнительный экземпляр сервиса. Давайте выполним команду `kubectl autoscale`:

```
$ kubectl autoscale deployment/time-service --min=1 --max=3 --cpu-percent=80
deployment "time-service" autoscaled
```

Все параметры более чем очевидны, и вновь одной командой мы получили автоматическое горизонтальное масштабирование, причем ограниченное и по нижней, и по верхней границе. Через некоторое время, учитывая что на наш тестовый микросервис нет особой нагрузки, Kubernetes придет в оптимальное состояние и выключит лишние отсеки, оставив указанное нами минимальное состояние - один отсек с экземпляром микросервиса. Мы сможем легко в этом убедиться, вновь просмотрев список отсеков.

Удаление развертывания и сервиса

После завершения тестов, стоит удалить экспериментальные сервисы и развертывания, особенно если вам нужно платить за ресурсы кластера. Сделать это с помощью `kubectl` просто. Как и во всех остальных случаях, требуется указать команду (`delete`), а затем указать ресурс, на который команда будет распространяться. Мы уже поняли, что развертывания и сервисы Kubernetes тесно связаны между собой, но существуют в системе Kubernetes в виде отдельных ресурсов, поэтому и удалять их нужно будет по отдельности. Удаление только развертывания или только сервиса не удалит связанную с ними другую часть.

```
$ kubectl delete deployment/time-service
deployment "time-service" deleted
$ kubectl delete service/time-service
service "time-service" deleted
```

Проверьте состояние кластера с помощью команд `get` и `describe`, и вы сможете убедиться в том, что нашего микросервиса, включая выделенные для него отсеки и порты, больше нет.

Визуальное представление кластера

Хотя получать списки развертываний, отсеков и сервисов (в общем называемых ресурсами) кластера под управлением Kubernetes через командную строку легко и быстро, при возрастании их количества проще смотреть на них визуально, в виде дерева или таблицы категорий различных ресурсов. К нашим услугам огромный выбор визуальных помощников.

Самое простое - использовать стандартную панель мониторинга (dashboard), встроенную в Minikube, вызвав команду `minikube dashboard`. Этот же инструмент можно установить на любой кластер Kubernetes, обычным вызовом `kubectl`. Есть и более удобные инструменты, их легко найти в Интернете.

Простое развертывание с Kubernetes - резюме

- Kubernetes может работать на большинстве кластеров вычислительных ресурсов (серверов). Облачные провайдеры предлагают готовые кластеры произвольных мощностей и размеров для развертывания ваших микросервисов и приложений.
- Все что требуется от разработчиков микросервисов или приложений - подготовить точки доступа через сетевые порты, и упаковать приложение со всеми зависимостями в образ, на основе которого будут запускаться контейнеры.
- Управляющим системам Kubernetes необходимо передать образ с микросервисом и указать желаемое состояние этого микросервиса - в обычных случаях его имя, количество экземпляров, необходимость доступа к нему из внешнего Интернета.
- Одной строкой вы можете указать Kubernetes уровень масштабирования вашего микросервиса, простым количеством экземпляров, или логично и просто настраиваемым автоматическим масштабированием.
- Одним из основных преимуществ Kubernetes является то, что это открытый проект open source, ко всем деталям которого вы имеете полный доступ. Развернутое и настроенное приложение с легкостью переносится между публичными провайдерами облака, такими как Google, Azure или AWS, и более того, ничто не мешает вам в любой момент настроить Kubernetes на своем собственном кластере, находящемся под вашим полным контролем, к примеру, если речь зайдет о особо важных данных или их географических ограничениях, и немедленно перенести его с открытого облака.
- Kubernetes обеспечит максимизацию вычислительных мощностей и минимальные усилия по сравнению с ручным или полуавтоматическим созданием отдельных виртуальных машин в облаке и развертыванием приложений на них.

6. Объекты Kubernetes. YAML. Декларативное управление кластером

В первой главе про Kubernetes мы рассмотрели его основные составляющие, изучили термины, и провели первое развертывание и масштабирование своего минимального сервиса, просто вызывая команду `kubectl` из командной строки. Это не потребовало у нас больших усилий, а результаты получились отличные, по большому счету достаточные для того, чтобы развернуть элементарные сетевые сервисы и приложения на кластере под управлением Kubernetes.

Для одного или двух простых сервисов вызов `kubectl` из командной строки еще может выполнять свою роль. Однако для приложения, составленного из множества микросервисов, потенциально нескольких десятков, запуск каждого сервиса с помощью командной строки, дополнительные настройки с помощью множества флагов, создание и координация открытых портов для взаимодействия сервисов, снова через отдельные вызова команды `kubectl`, приведет к плохо поддерживаемой, и склонной к запутанным ошибкам системе развертывания приложения как готовой системы.

Конечно, можно было бы создать единый скрипт развертывания системы целиком, и запускать его на сервере, предназначенном исключительно для этой задачи. Отслеживать изменения в процессе запуска и развертывания проекта можно храня исходный текст этого скрипта в системе контроля версий. Но скрипты для запуска сложных действий из командной строки имеют склонность становится чрезмерное сложными, и разрастаться до порядочных размеров. К тому же языки командных оболочек, таких как `shell`, не совсем предназначены для разветвленной логики, и в итоге их сложность становится серьезным препятствием к быстрому изменению.

Kubernetes не стал бы столь популярным без модели управления кластером с помощью объектов своего программного интерфейса, следующего стандарту REST (RESTful API). Желаемое состояние (desired state) кластера Kubernetes в каждый момент описывается набором объектов, являющихся частью программного интерфейса Kubernetes. У каждого из этих объектов есть набор полей, допустимых значений и вложенных объектов, однозначно описывающих, что именно требуется от управляющих компонентов кластера для того, чтобы текущее состояние кластера пришло к желаемому состоянию, описанному объектами. Объекты Kubernetes - это RESTful ресурсы программного интерфейса Kubernetes, но как мы увидим, работать с ним напрямую нам не придется, все сделает команда `kubectl`.

Когда мы создавали свое первое развертывание, вызывая команду `kubectl run`, за кулисами для нас создали объект Kubernetes под названием `Deployment` (развертывание). После этого

мы открыли порт для доступа и взаимодействия со своим приложением в контейнере (`kubectl expose`) - то есть создали сервис, и нетрудно догадаться, эта привело к созданию объекта `Service`.

Основной способ управления кластером Kubernetes - именно описание объектов его программного интерфейса и последующая работа с ними. Вместо запуска хотя и простых и понятных команд `kubectl`, гораздо более гибким и легким для последующих изменений способом управления является прямое описание этих объектов. Для описания обычно используется язык YAML, хотя в итоге, перед отправкой на управляющий сервер Kubernetes, он преобразуется в JSON. Файлы формата YAML хранятся в системе контроля версий, зачастую рядом с исходным кодом и данными приложения, и изменение способа развертывания приложения в кластере Kubernetes легко обновлять и отслеживать.

Несколько слов об “ужасах” YAML

Работа с YAML в Kubernetes стала практически притчей во языхец. Так как мощь и гибкость Kubernetes требует создания немалого количества файлов YAML и работы с ними, разработчики часто саркастически замечают, что Kubernetes это и есть редактирование YAML, не самого удобного для человека языка разметки. Как подметил Келси Хайтауэр в своем Твиттере (Kelsey Hightower, один из самых заметных проводников и пропагандистов идей Kubernetes), “Я был на конференции, которая была полностью посвящена только YAML. Эта была конференция KubeCon (главная конференция Kubernetes)”.

Все это так, и создавать объекты Kubernetes с нуля, мучительно вспоминая все поля и правила записи YAML, действительно непросто. Однако все гениальное просто, и, как правило, эта проблема элементарно решается шаблонами и помощью редактора. Особенно хороши в этом различные расширения и плагины для сред разработки, таких как IntelliJ и VS Code. Ну а в этом главе мы попробуем обойтись без дополнительных средств, используя шаблоны команды `kubectl` (да, они есть и там!). Список удобных инструментов для работы с объектами Kubernetes вы сможете найти на сайте книги.

Объект Deployment вместо `kubectl run`

Как мы уже упомянули, команда `kubectl run`, примененная нам в предыдущей главе для развертывания и запуска контейнера в кластере Kubernetes, на самом деле создает объект Kubernetes `Deployment` и отправляет его на управляющий узел.

К нашей радости все та же самая команда поможет нам получить этот объект в виде YAML. Используем мы ее с сочетанием пары флагов `--dry-run -o yaml`, что означает не что иное, как запуск в “холостом режиме” без отправки в кластер, и вывод созданного объекта Kubernetes на консоль. Формат вывода мы запрашиваем как раз YAML.

Итак, развернем сервис `time-service` из предыдущего раздела в холостом режиме, используя для простоты образ image из моего репозитория Docker Hub (`ivanporty`). Вы можете заменить репозиторий на свой собственный.

```
$ kubectl run time-service --image=ivanporty/time-service:0.1.0 --dry-run -o yaml
```

Мы легко получаем описание объекта `Deployment`, которое команда `kubectl run` отправила бы на сервер программного интерфейса Kubernetes apiserver в случае не холостого, а реального вызова:

```
# Версия программного интерфейса Kubernetes
apiVersion: apps/v1
# Тип объекта
kind: Deployment
# Метаданные нашего объекта, вложенный объект ObjectMeta
metadata:
  creationTimestamp: null
  # список меток самого объекта Deployment
  labels:
    run: time-service
  name: time-service
# Описание собственно правил развертывания контейнера
# Вложенный объект DeploymentSpec
spec:
  # Количество запущенных отсеков pods для масштабирования
  replicas: 1
  selector:
    matchLabels:
      run: time-service
  strategy: {}
  # описание шаблона для создания новых отсеков
  template:
    metadata:
      creationTimestamp: null
      # список меток для нового отсека
      labels:
        run: time-service
    # непосредственно описание контейнера в отсеке
    spec:
      containers:
      - image: ivanporty/time-service
        name: time-service:0.1.0
```

```
33          resources: {}
34  status: {}
```

Сразу можно видеть, что информации объект Deployment предоставляет намного больше, чем мы указывали при запуске команды `kubectl run`, но последняя многим полям присваивает разумные значения по умолчанию. Для простоты комментарии с описанием самых важных полей и вложенных объектов добавлены прямо в шаблон YAML.

Для начала мы просто хотим запустить наш контейнер в кластере, поэтому для нас самым важным является способ указать, где находится образ этого контейнера. Этим заведует шаблон, по которому создаются все отсеки и контейнеры внутри них - `.tempate.spec`. Еще мы видим загадочный набор меток (labels), повторяющийся три раза - это уникальные “бирки” на объектах Kubernetes, позволяющие найти нужные объекты в сложном большом кластере. Пока же давайте считать, что для простоты они все должны совпадать.

Используя этот шаблон, создадим свой первый объект Kubernetes, оставив лишь нужные нам поля. Сохраним его в отдельном файле YAML и назовем `k8s-time-service-deploy.yaml`. Вместо метки `run`, которая используется командой `kubectl run`, используем свою собственную метку, чтобы обозначить свое развертывание. Пусть это будет просто описание приложения `app[lication]`, довольно популярная метка для простых развертываний.

Вот что у нас получится за объект:

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     labels:
5       app: time-service
6     name: time-service
7   spec:
8     replicas: 1
9     selector:
10      matchLabels:
11        app: time-service
12    template:
13      metadata:
14        labels:
15          app: time-service
16      spec:
17        containers:
18        - image: ivanporty/time-service:0.1.0
19          name: time-service
```

Остается только создать (`create`) свое развертывание с помощью объекта Kubernetes. Передать команде `kubectl` описание объекта из файла YAML нужно с помощью флага `-f`.

(Внимание: Если у вас еще остался запущенным сервис и развертывание `time-service` из предыдущего раздела, не забудьте их удалить с помощью команд `kubectl delete`!)

```
$ kubectl create -f k8s-time-service-deploy.yaml
```

Убедиться в том, что развертывание создано, можно с помощью уже хорошо знакомых команд:

```
$ kubectl get deployments
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
time-service   1         1         1            1           9m
```

Чем сложнее и более распределена по небольшим микросервисам будет ваша система и приложение в целом, тем больше преимуществ предоставит управление кластером с помощью объектов, а не вызовов команды `kubect xyz ...`:

- Объекты хранятся в системе контроля версий рядом с исходным кодом системы. Вы видите, какие изменения были сделаны с состоянием кластера просто по изменениям YAML и комментариям в Git.
- Элементарно добавить новые метки, изменить количество экземпляров, передать контейнерам переменные окружения, все в одном описании Deployment, и опять же, легко следить за этим в системе контроля версий.
- Любой оператор кластера, включая разработчиков, сможет мгновенно восстановить состояние кластера в определенный момент, просто воссоздав объекты Kubernetes из системы контроля версий. Сделать это вызовами команды `kubectl` будет очень сложно и путанно.

Объект Service вместо `kubectl expose`

При знакомстве с Kubernetes мы использовали команду `kubectl expose` для открытия сетевого порта и доступа к работающему контейнеру, и этой же командой в холостом режиме сможем получить отличный шаблон для объекта сервиса Kubernetes `Service`:

```
$ kubectl expose deployment time-service --port=8080 --type=NodePort --dry-run -o ya\
ml
```

Вот что у нас получится в результате. Структура объекта будет очень похожа на развертывание `Deployment` и для краткости здесь удалены лишние поля, а метки сразу же заменены на `app` вместо `run`, аналогично тем, что мы использовали только что для развертывания:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: time-service
  name: time-service
spec:
  # список портов. Дополнительно можно указать протокол
  ports:
  - port: 8080
  # по этим меткам идет поиск отсеков, куда отправляются запросы
  selector:
    app: time-service
  # тип сервиса. В облаке можно использовать LoadBalancer
  type: NodePort
```

Базовое описание полей сервиса есть прямо в файле YAML, ну а полностью мы рассмотрим все детали и возможности сервисов Kubernetes в отдельной главе.

Назовем файл по аналогии с развертыванием, намекнув в его названии что он содержит описание сервиса: `k8s-time-svc.yaml`. Создадим новый сервис (если у вас запущен предыдущий вариант сервиса, лучше будет его удалить):

```
$ kubectl create -f k8s-time-svc.yaml
```

И уже знакомыми командами узнаем, что сервис успешно создан и увидим порт внутри кластера, по которому к нему можно обратиться.

```
$ kubectl get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP          17d
time-service   NodePort    10.98.92.168   <none>        8080:30721/TCP   5s
```

В прошлом разделе мы посмотрели, как в локальных кластерах Minikube или Docker обратиться к сервисам типа NodePort, это будет хорошее повторение материала.

Если вы пользуетесь публичным полноценным кластером, таким как Google Cloud Kubernetes Engine, вы сможете получить доступ к сервису из Интернета, поменяв в описании объекта тип сервиса на `LoadBalancer`, а затем получив его открытый IP-адрес из описания сервиса, полученного такой же командой `kubectl get services`.

Императивное управление кластером Kubernetes - create и delete

После описания объектов Kubernetes в формате YAML остается их создать (`kubectl create -f`), а потом удалить (`kubectl delete -f`). Подобная работа с объектами в программировании называется императивной (imperative), или процедурной - мы напрямую создаем объекты и управляем их жизненным циклом в “приказном порядке”, при необходимости обновляем, в конце концов удаляем, и затем создаем заново с обновленными настройками. Именно так мы пока и поступали с созданными нами развертыванием и сервисом для своего приложения `time-service`. Система лишь исполняет наши инструкции и в управлении непосредственно объектами не участвует - она лишь использует их для определения, каким должно быть “желаемое состояние” (desired state) нашего кластера.

Интересно (и удобно), что инструмент `kubectl` поддерживает создание и удаление объектов не только из одного файла (а в одном файле могут содержаться множество объектов разного типа, например развертывание и сервисы вместе), а множества файлов YAML, находящихся в директории. Учитывая, что мы поместили наши декларации Kubernetes (два файла с объектами развертывания и сервиса) в директорию `k8s`, создать и удалить все необходимое для микросервиса `time-service` можно и так:

```
$ kubectl delete -f k8s/
deployment.apps "time-service" deleted
service "time-service" deleted

$ kubectl create -f k8s/
deployment.apps "time-service" created
service "time-service" created
```

Если после предыдущих примеров развертывание и сервис уже существуют, сначала удалите их, а затем попробуйте эти команды. Для множества микросервисов вероятно появление и множества деклараций с объектами кластера Kubernetes, и такая команда становится весьма кстати, если вы развертываете все сервисы разом.

Что же дальше? Управлять созданием и удалением объектов кластера и менять таким образом его желаемое состояние понятно и прямолинейно. Однако дополнительные преимущества Kubernetes использовать будет непросто - например, что, если вы хотите создать автоматически масштабируемое приложение? Или хотите обновить контейнеры только на части отсеков (pods), или меняете версию контейнера только для одного отсека? Удалять и создавать под-множества объектов отдельными командами будет неудобно, также как очень тяжело отследить историю внесенных изменений. Для больших кластеров, сложных приложений и сервисов, почти всегда применяется *декларативное* управление.

Декларативное управление кластером. Команда apply.

Декларативное описание объектов Kubernetes не требует от нас прямого вмешательства в дела кластера и оставляет управление объектами, включая их создание и удаление, в руках системы управления кластером. Девиз декларативного описания - неважно в каком порядке, в течение какого времени и как часто система получает описание объектов кластера. Управление кластера обязуется преобразовать простое, понятное и легко отслеживаемое описание объекта в *желаемое состояние кластера* (desired state).

Если мы передали системе объект с развертыванием `Deployment`, и такого развертывания в кластере еще нет, он будет создан, как если бы была вызвана команда `create`. Если мы передали обновленный сервис `Service`, в котором, например, добавили еще один порт, система динамически обновит соответствующий объект (или же удалит и создаст заново), и приведет кластер в его желаемое состояние.

Получить описание объектов и сделать на их основе необходимые действия позволяет команда `kubectl apply`. Учитывая, что основы объектов развертывания и сервисов мы уже знаем, и создали файлы YAML с их описанием, передача объектов в управление кластера теперь происходит в одну строчку

```
$ kubectl apply -f k8s/
deployment.apps "time-service" created
service "time-service" created
```

Как мы видим, при отсутствии объектов в системе они будут созданы. Если бы они там уже существовали, изменений в системе не было бы.

Предположим, что мы получили первых пользователей для микросервиса и одного экземпляра сервиса нам мало, а использовать автоматическое масштабирование мы пока не хотим. Отредактируем объект развертывания и укажем, что нам требуется два экземпляра (`replicas`) отсека и контейнера в нем при развертывании:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: time-service
  name: time-service
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
11        app: time-service
12    template:
13      metadata:
14        labels:
15          app: time-service
16      spec:
17        containers:
18        - image: ivanporty/time-service
19          name: time-service
```

Декларативное описание позволит не волноваться о том, удалено ли старое развертывание и проверять, в скольких экземпляров оно находится сейчас (а может быть, оно вообще еще не создано). Передадим объекты в управление кластера и получим свое желаемое состояние без дополнительных усилий:

```
$ kubectl apply -f k8s/
deployment.apps "time-service" configured
service "time-service" unchanged
```

Как мы видим, изменений в сервисе не было, а вот конфигурация развертывания была изменена - причем команда `apply` описывает, что именно было сделано с объектами - создание, удаление или пере-конфигурация.

Проверим, в каком состоянии и в скольких экземплярах работает развертывание:

```
$ kubectl get deploy
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
time-service   2         2         2            2           2m
```

Состояние кластера верно. Декларативное описание развертываний и сервисов с помощью объектов Kubernetes помогло нам достичь своего желаемого состояния без дополнительных проверок командой `get`, создания и удаления объектов, масштабирования `scale`, и множества других “ручных” команд.

Если мы управляем кластером исключительно декларативным способом, то в случае отсутствия ошибок всегда можем быть уверены в том, что его состояние совпадает с тем, что описано нашими объектами Kubernetes. Управление сложными развертываниями, их обновлениями, и масштабированием становится намного проще и понятнее.

Резюме

- Простой запуск контейнеров, открытие портов, и масштабирование в Kubernetes возможны просто с использованием команды `kubectl`. На этой основе можно создать

простые скрипты для управления кластером. Однако это императивное управление, зависящее от предыдущего и текущего состояния кластера, и поэтому оно быстро становится сложным и запутанным.

- Программный интерфейс Kubernetes API, доступный через apiserver, полностью следует концепции REST. Все концепции и возможности Kubernetes описываются как RESTful ресурсы, называемыми *объектами Kubernetes*. Обычно их описывают в формате YAML и отслеживают в системе контроля версий.
- Развернуть и запустить контейнер в кластере Kubernetes и открыть его порты позволяют объекты Deployment и Service. Их можно напрямую (императивно) создавать и удалять командами `create` и `delete`.
- Самым удобным и максимально уменьшающим ошибки способом управления состоянием кластера является *декларативный*. Мы передаем управляющей системе кластера набор объектов командой `kubectl apply`, и говорим что хотим видеть кластер соответствующим переданным объектам (желаемое состояние). Управляющий цикл (control loop) Kubernetes постоянно проверяет состояние объектов и приводит кластер к желаемому состоянию.

7. Взаимодействие сервисов Kubernetes

Одним из впечатляющих преимуществ разработки приложения в виде набора микросервисов является возможность работать над ними индивидуально, используя разные библиотеки, языки, платформы, более того, различный процесс и подход к разработке. Далее микросервисы отдельно масштабируются, обновляются и отлаживаются - все это делает жизнь разработчиков более гибкой, а разработку быстрой. Однако возникает вопрос взаимодействия микросервисов - вместо вызова обычных методов и функций каждое взаимодействие становится сетевым вызовом, как правило, следующим формату HTTP и правилам вызова REST.

Вопрос эффективности сетевых взаимодействий между компонентами одной большой системы довольно обширен. Туда входят протоколы данных, оптимизация и сжатие данных, попытка найти золотую середину между громоздкими данными в форматах JSON/Protobuf, слепляющими несколько логических вызовов в один большой пакет, и нагрузкой на сеть, возникающую при чрезмерно частых и мелких сетевых вызовах. Определить нужные параметры и выбрать оптимальный протокол для своих сервисов и систем сможете только вы сами, но взаимодействие сервисов в мире Kubernetes уже работает для вас без дополнительных усилий.

Основной задачей системы управления кластером является обнаружение сервисов (service discovery). Полагаться на прямое использование обычного IP-адреса нельзя - кластер Kubernetes слишком динамичен, при обновлении версий, ошибках, повышении нагрузки и масштабировании отсеки (pod) с контейнерами останавливаются и удаляются, потом запускаются заново, а их сетевые адреса все время меняются. В Интернете системой обнаружения является сервис поиска имен DNS, сервера которой поддерживают базу данных имен и адресов для них, и позволяют пользователям быстро переходить на нужный IP-адрес по простому имени, даже если адреса постоянно меняются.

DNS используется и в кластере Kubernetes. Каждому сервису, созданному с помощью объекта Service, присваивается доменное имя, совпадающее с именем самого сервиса. Это доменное имя доступно только внутри самого кластера, на каждом его узле, для любого запущенного там контейнера. Для взаимодействия сервисы находят необходимые им для работы сервисы-партнеры по заранее известному имени в DNS. Как мы помним, у каждого сервиса Kubernetes должно быть свое уникальное имя.

Давайте проверим.

Обнаружение сервисов через DNS

Для начала мы посмотрим, как именно работает система обнаружения имен DNS внутри нашего кластера Kubernetes. Для этого нам понадобится получить к ней доступ - все внутренние системы Kubernetes, кроме управляющих, например, сервера приема команд API (API server), недоступны извне кластера. Это же относится и к внутренней системе DNS. Можно было бы написать простой скрипт `shell`, упаковать его в контейнер, запустить на кластере и проверить его журнальные записи (как мы знаем, состоящие из записей стандартного вывода), но это чрезмерно долго и сложно.

Для проверки параметров сети и имен DNS нам понадобится обычный терминал с самыми распространенными командами Linux, и здесь как нельзя кстати пригодится специальный вариант команды `kubectl run`. Используя ключ `--tty`, мы просим инструмент `kubectl` присоединить к запущенному контейнеру эмулятор терминала, а ключ `-i` (interactive) открывает этот терминал сразу же после запуска контейнера, и в том же сеансе, который используется для самой команды `kubectl`. Остается выбрать образ Linux, который мы запустим - для этих целей как нельзя лучше пригодится самый минимальный образ `busybox`, основанный на крайне минималистичной версии Linux, применяемой для встраиваемых устройств - размер этого образа не превышает нескольких мегабайт, а ресурсов требует этот контейнер совсем мало!

```
$ kubectl run -i --tty busybox --image=busybox --restart=Never -- sh
/ #
```

Запустив наш контейнер `busybox`, мы тут же получим доступ к интерактивному терминалу, работающему внутри нашего кластера. Здесь нам доступны все системы и все адреса кластера, и теперь мы можем использовать базовые сетевые команды Linux для проверки наших сервисов в DNS.

```
$ nslookup time-service
Server:                10.15.240.10
Address:        10.15.240.10:53

Name:        time-service.default.svc.cluster.local
Address: 10.15.251.13
```

Как мы уже упомянули, каждому сервису, созданному с помощью объекта Service, присваивается доменное имя, совпадающее с именем самого сервиса. Именно это мы видим в выводе команды `nslookup`, передав ей в качестве параметра имя нашего сервиса `time-service` (пока он единственный в нашем кластере).

Сервис `time-service` доступен не только по простому имени сервиса из объекта Service, но мы видим и более сложное составное имя `time-service.default.svc.cluster.local`. Это

общий формат имени сервиса в случае использования дополнительных пространств имен (namespace) в кластере Kubernetes. Мы пока не использовали их и узнаем о них подробнее чуть позже, но по сути это способ разделить кластер на индивидуальные зоны, где возможно использовать одни и те же имена объектов Kubernetes, что-то вроде пакетов в Java или пространств имен в C#. Если вы не указывает пространство имен явно, все сервисы и остальные объекты Kubernetes создаются в пространстве по умолчанию `default`. Если вы захотите вызвать сервис из другого пространства имен, придется указать именно такой формат или чуть короче, просто указав пространство имен через точку:

```
[имя сервиса].[пространство имен = default].[сервис = svc].[общий суффикс для класте\
ра = cluster.local]

[имя сервиса].[пространство имен = default]
```

Суффикс кластера обычно не меняют, для примера в кластере Google Kubernetes Engine это `cluster.local`, но при желании можно поменять и его - как правило это связано с взаимодействием имен из нескольких кластеров одновременно, и встречается не так часто.

Мы видим, что наш сервис действительно зарегистрирован в системе DNS нашего кластера Kubernetes, но сработает ли сам сервис?

```
$ wget time-service:8080/nanotime -O -
Connecting to 10.15.251.13:8080 (10.15.251.13:8080)
{"nanoTime":"1547850153973197546"}

$ wget time-service.default.svc.cluster.local:8080/nanotime -O -
Connecting to time-service.default.svc.cluster.local:8080 (10.15.251.13:8080)
{"nanoTime":"1547850322608275624"}

$ wget time-service.default:8080/nanotime -O -
Connecting to time-service.default:8080 (10.97.105.149:8080)
{"nanoTime":"1550678757460527000"}
```

Сервис работает, и путь к простому взаимодействию микросервисов в кластере Kubernetes открыт - им необходимо знать лишь имена сервисов-партнеров и открытые ими порты, или, в случае различных пространств имен, имя сервиса, пространство имени, и потенциально суффикс имен кластера. Необходимости искать и снабжать сервисы точными адресами IP и управлять этими адресами нет.

Обнаружение сервисов с помощью переменных окружения

До стабильной поддержки обнаружения сервисов с помощью службы Kubernetes DNS при появлении нового сервиса управляющий узел Kubernetes добавлял в каждый работающий в кластере отсек набор переменных окружения, состоящих из имени сервиса и поддерживаемых им портов. Эти переменные содержали IP-адрес, по которому сервис можно было обнаружить. Мы можем проверить, что это по прежнему работает и для нашего сервиса `time-service`. Используем запущенный контейнер с Linux `busybox` и посмотрим, какие переменные, начинающиеся с имени нашего сервиса, доступны в отсеке:

```
$ printenv | grep TIME_SERVICE
TIME_SERVICE_PORT=tcp://10.97.105.149:8080
TIME_SERVICE_SERVICE_PORT=8080
TIME_SERVICE_PORT_8080_TCP=tcp://10.97.105.149:8080
TIME_SERVICE_SERVICE_HOST=10.97.105.149
TIME_SERVICE_PORT_8080_TCP_ADDR=10.97.105.149
TIME_SERVICE_PORT_8080_TCP_PORT=8080
TIME_SERVICE_PORT_8080_TCP_PROTO=tcp
```

Можно использовать эти переменные напрямую из терминала для подключения к сервису:

```
$ wget $TIME_SERVICE_SERVICE_HOST:$TIME_SERVICE_SERVICE_PORT/time -O -
Connecting to 10.97.105.149:8080 (10.97.105.149:8080)
{"time":"2019-02-20 16:33:02.865982 +0000 UTC m=+139444.036060438"}
```

В основном использовались две переменные окружения - `[ИМЯ СЕРВИСА]_[SERVICE_HOST]` и `[ИМЯ СЕРВИСА]_[SERVICE_PORT]`, определяющие адрес сервиса и его порт, остальные переменные добавляются для совместимости с Docker. По большому счету, использовать переменные окружения не только менее удобно, чем имена DNS, но и не всегда разумно - переменные окружения добавляются во все отсеки немедленно после появления нового объекта Service, даже если он еще не готов. В конце главы мы узнаем, как определяется готовность сервиса в Kubernetes и реализуем такую проверку готовности для своего нового сервиса.

Сервисы и метки

Мы увидели, что сервисы Kubernetes позволяют вам найти микросервис или приложение по его имени, получить его сетевой адрес внутри кластера и открыть соединение. Как именно система обнаружения находит наш контейнер (работающий внутри отсека pod)? Используются метки (label), которые мы задаем в объекте Service и затем передаем в управляющую систему Kubernetes. Метки задаются в секции metadata и доступны для любого объекта Kubernetes. Вот что мы указали в прошлом разделе для своего микросервиса `time-servce`:

```
...
metadata:
  labels:
    app: time-service
```

Мы использовали всего одну метку (`app`), и таким образом, все отсеки, имеющие совпадение по этим меткам, будут выбираться для доступа через созданный сервис. Когда мы создавали наше развертывание Deployment, мы указали, что для всех созданных в этом управлении развертываний будет применяться те же самые метки (шаблон для всех создаваемых отсеков с контейнерами - `template.spec`):

```
...
  template:
    metadata:
      labels:
        app: time-service
```

Использование меток позволяет однозначно находить все экземпляры, подходящие для работы с сервисом, независимо от того, в каком количестве они созданы, какие версии контейнеров в них запущены, и как они обновляются и управляются. Для отладки можно воспользоваться ручным поиском отсеков по меткам с помощью ключа `--selector`, указав для него список меток и их значений:

```
$ kubectl get pods --selector=app=time-service
NAME                            READY   STATUS    RESTARTS   AGE
time-service-77d9656579-xrg52   1/1     Running   0          11d
```

Мы ожидаемо получили единственный экземпляр своего микросервиса (и получим больше, если применим масштабирование) - это единственные отсеки с таким набором меток. Именно этот отсек будет вызываться через систему обнаружения сервисов Kubernetes.

Основные типы сервисов в Kubernetes

Сервис в Kubernetes - это точка доступа к приложению через сетевой порт. Вариантов такого доступа может быть несколько, и особенно важно в динамической, масштабируемой среде запуска Kubernetes обеспечить поддержку работы с приложением, работающим во множестве экземпляров. В этом случае обратиться напрямую к его IP-адресу и порту будет неправильно - мы просто будем игнорировать остальные экземпляры. В простейшем случае должна быть возможность выбрать один из экземпляров приложения случайным образом или циклически. Поддержка динамических экземпляров в Kubernetes использует особую сетевую архитектуру и три основных вида сервисов (поле `type` в описании объекта Service):

- `ClusterIP`. Это тип сервиса по умолчанию. Сервис будет доступен через IP-адрес в кластере, внутри сети кластера, и не будет доступен снаружи кластера и в Интернете. Остальные микросервисы и части приложения смогут найти этот адрес через обнаружение имен Kubernetes DNS. Доступ к экземплярам (контейнерам) этого микросервиса обеспечивает виртуальный IP-адрес и прокси-компонент kube-proxy, особенности которого мы сейчас рассмотрим. В большинстве случаев микросервисы (как составные части приложения) являются вспомогательными и не должны быть доступны извне - это подходящий для них тип сервиса.
- `NodePort`. Порт на каждом узле кластера. Для сервиса отводится уникальный номер порта (*внимание - он не совпадает с номером порта самого приложения!*). Этот порт открывается на всех узлах кластера. Номер порта выбирается из специально отведенного диапазона. Обычно этот тип сервиса хорош для нестандартной настройки балансировки нагрузки - сделать это через единый номер порта и все узлы кластера проще. Также к этому типу сервиса легче получить доступ для отладки - достаточно сделать переадресацию портов с любого узла кластера или обратиться напрямую к узлу, к которому у вас есть доступ.
- `Load Balancer`. Балансировщик нагрузки, встроенный в кластер. Обычно он доступен только в публичных облаках, таких как Google Cloud или AWS (и недоступен в эмуляторах Docker и minikube). По сути сервис создается с типом NodePort, а затем к нему подключается балансировщик нагрузки от провайдера облака. Обычно это делается для единой внешней точки входа в приложение из Интернета. Обратите внимание, что стоимость такого балансировщика может быть весьма немалой! Мы уже использовали этот тип сервиса в самом начале знакомства с Kubernetes для своих первых тестов.

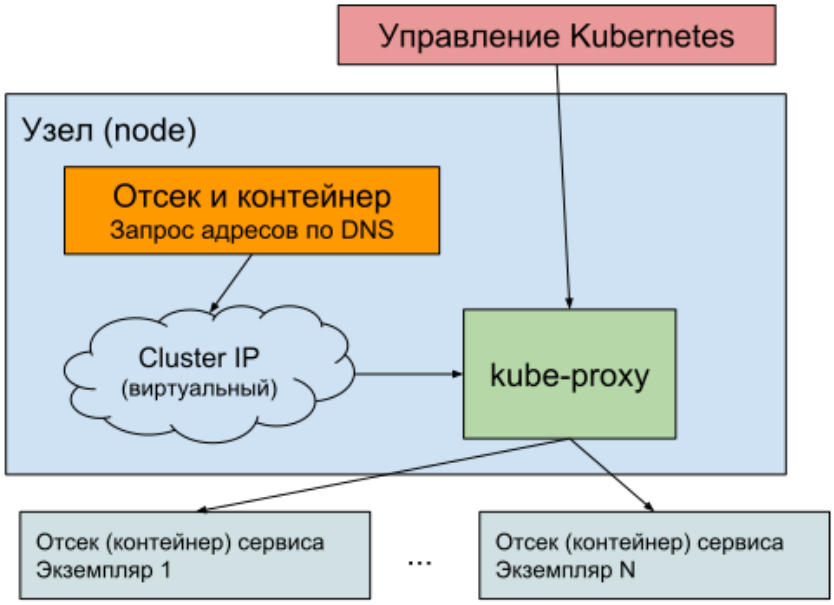
Виртуальные IP-адреса. Kube-proxy

Все мы примерно представляем, как работает система доменных имен DNS - на каждое имя тем или иным способом, обычно с помощью записей типа A, возвращается список точных IP-адресов, и по этому адресу находится или сам сервер, или (чаще) точка входа в кластер, обычно это сервер балансировщика нагрузки. В мире Kubernetes сервисы работают внутри отсеков (pods), при создании отсека при масштабировании, обновлениях, новых версиях, всем им присваиваются новые IP-адреса внутри кластера. Это может происходить часто и быстро, и кэш DNS просто не успеет обновить список адресов, по которым надо обратиться. Решение в Kubernetes - виртуальные IP-адреса.

Виртуальные IP-адреса не указывают на конкретный сервер или виртуальный отсек. Вместо этого они служат точкой переадресации на список существующих в данный момент отсеков, соответствующих сервису. Как мы видели, список этих сервисов определяется выбором по меткам. Напрямую обращаться к ним не получится. Для обеспечения их работы в каждом узле (node) кластера Kubernetes работает вспомогательный прокси-компонент kube-proxy. Через этот прокси проходит весь сетевой трафик узла ото всех работающих в нем отсеков (и соответственно, находящихся в них ваших контейнеров с сервисами).

При обращении к виртуальному IP-адресу kube-proxy находит список соответствующих этому адресу отсеков (по меткам сервиса), и выбирает один из них. Выбор одного из множества отсеков, если сервис работает во множестве экземпляров, зависит от настроек самого кластера - как правило это или случайный порядок, или классический циклический выбор, она же “карусель” (round robin).

На схеме это выглядит примерно так:



Теперь картина создания и работы сервиса становится полной - при создании нового объекта Service в кластере создается новый виртуальный IP-адрес, и прокси-компоненты kube-proxy на всех узлах добавляют его в свои списки адресов. После того, как сервис готов к работе, обращение по этому виртуальному адресу переадресуется к одному из отсеков, которые относятся к сервису (поиск по меткам). При перезапуске отсеков, увеличении и уменьшении их количества при масштабировании, их обновлении, состояние kube-proxy постоянно обновляется, позволяя работающим в кластере сервисам прозрачно работать с виртуальным адресом, совершенно не волнуясь о всех происходящих за его кулисами переменах. Это великолепная возможность реализовать взаимодействие большой группы распределенных сервисов в очень динамичном, постоянно меняющемся состоянии.

Развертывание нескольких сервисов

Итак, мы поняли, что можем находить сервисы просто по их имени, используя встроенную в Kubernetes поддержку DNS. Мы рассматриваем не только возможности Kubernetes, но больше

непосредственно разработку новых облачных сервисов и их работу в кластере. Давайте напишем новый сервис, который будет работать на том же самом кластере, где в предыдущих главах мы уже тестировали и запускали простейший микросервис для определения текущего времени. Код этого сервиса будет обращаться к сервису времени, получать текущее время (представим гипотетически, что отдельный сервис времени позволит нам решить задачу точной синхронизации), и определять, выходной ли сегодня день. Как мы прекрасно помним, в мире много часовых поясов (time zone), в некоторых из них дни разные, поэтому такой сервис должен в качестве параметра знать часовой пояс, для которого определяется выходной день. Возможно, такой сервис нам пригодится позже для создания календаря и других услуг.

Создание второго микросервиса, который должен будет использовать библиотеку с поддержкой часовых поясов и инструменты для работы с календарями и датами, как нельзя лучше поможет нам увидеть, как легко соединить совершенно разные технологии и подходы к разработке в одном кластере Kubernetes. Для определения часового пояса и выходных дней используем Java, точнее встроенный в основной набор библиотек SDK пакет java.time, для взаимодействия в стиле HTTP/REST применим простую и компактную библиотеку Spark, которая будет работать со встроенным сервером Jetty, а работу с форматом JSON организуем с помощью библиотеки Google GSON.

В результате технологии двух наших сервисов будут отличаться как небо и земля, но как мы увидим, устроить их взаимодействие и развертывание с мощью Kubernetes будет просто.

Итак, напишем сервис выходного дня:

```
1  package com.porty.k8s;
2
3  import com.google.gson.Gson;
4  import org.slf4j.Logger;
5  import org.slf4j.LoggerFactory;
6
7  import java.io.InputStreamReader;
8  import java.net.URI;
9  import java.time.DayOfWeek;
10 import java.time.Instant;
11 import java.time.ZoneId;
12 import java.util.EnumSet;
13 import java.util.concurrent.atomic.AtomicBoolean;
14
15 import static spark.Spark.*;
16
17 /**
18 * Простой сервис на основе Spark/Java, запрашивающий время у микросервиса
19 * time-service и возвращающий результат - выходной ли день в указанной
20 * временной зоне, и какой именно это выходной.
```

```
21  */
22  public class WeekendService {
23      private static Logger logger =
24              LoggerFactory.getLogger(WeekendService.class);
25
26      public static void main(String[] args) {
27          port(5678);
28
29          Gson gson = new Gson();
30
31          // страна и город для выяснения выходного дня кодируется прямо в пути
32          // запроса HTTP
33          // пример: /weekend/Europe/Moscow
34          get("/weekend/:country/:city", (req, res) -> {
35              ZoneId timeZoneId =
36                      ZoneId.of(req.params("country") + "/" + req.params("city"));
37              logger.info("Запрошен статус выходного дня для зоны {}",
38                      timeZoneId);
39
40              // запрашиваем время от вспомогательного сервиса и преобразуем
41              // его в объект с данными
42              TimeServiceResponse timeServiceResponse = gson.fromJson(
43                      new InputStreamReader(
44                              URI.create("http://time-service:8080/nanotime").
45                                      toURL().openStream()),
46                      TimeServiceResponse.class);
47              // используем пакет java.time для получения данных о текущем дне
48              Instant millisTime = Instant.ofEpochMilli(
49                      Long.parseLong(timeServiceResponse.getNanoTime()) / 1000000);
50              DayOfWeek dayOfWeek = millisTime.atZone(timeZoneId).getDayOfWeek();
51              boolean isWeekend = EnumSet.of(DayOfWeek.SATURDAY, DayOfWeek.SUNDAY)
52                      .contains(dayOfWeek);
53
54              return new TimeZoneReply(isWeekend, dayOfWeek.name());
55          }, gson::toJson);
56      }
57
58      // стандартный класс с данными для преобразования результата сервиса в JSON
59      static class TimeZoneReply {
60          private boolean weekend;
61          private String day;
62
63          TimeZoneReply(boolean weekend, String day) {
```

```
64              this.weekend = weekend;
65              this.day = day;
66          }
67
68          public boolean isWeekend() {
69              return weekend;
70          }
71
72          public String getDay() {
73              return day;
74          }
75      }
76
77      // стандартный класс с данными для получения данных JSON от сервиса
78      // time-service
79      static class TimeServiceResponse {
80          private String time;
81          private String nanoTime;
82
83          public String getTime() {
84              return time;
85          }
86
87          public void setTime(String time) {
88              this.time = time;
89          }
90
91          public String getNanoTime() {
92              return nanoTime;
93          }
94
95          public void setNanoTime(String nanoTime) {
96              this.nanoTime = nanoTime;
97          }
98      }
99  }
```

Код краток и достаточно информативен сам по себе - мы импортируем статические методы библиотеки Spark, и задаем редко используемый номер порта методом `port`, чтобы избежать конфликтов и путаницы с широко распространенными номерами портов при тестировании. Основной метод нашего нового сервиса - обработка HTTP-запроса GET по маршруту `/weekend`. Для простоты обращения к сервису необходимые нам параметры для указания часового пояса задаются прямо в маршруте HTTP, а не дополнительными параметрами,

так что для того, чтобы выяснить выходной ли день в Москве, нужно будет вызвать адрес `/weekend/Europe/Moscow`.

Для получения времени используется уже созданный и развернутый нами в кластере Kubernetes сервис `time-service`, и заранее известный нам номер порта, по которому к нему можно обратиться. Мы используем класс URL и открываем с его помощью поток данных для чтения InputStream, ну а тот, в свою очередь, преобразуется в объект с данными Java `TimeServiceResponse` с помощью GSON. Для преобразования необходимо совпадения полей и их типов между данными Json и классом Java.

Далее мы преобразуем время с точностью до наносекунд, которое нам предоставляет сервис `time-service`, в миллисекунды, и используя элементарные преобразования библиотеки Java Time, получаем день недели в указанном часовом поясе, легко выводя, выходной ли это день. Далее данные упаковываются в объект `TimeZoneReply` и преобразуются в Json с помощью GSON. Все детали используемых методов и библиотек легко можно найти в их документации в Интернете.

Наконец, чтобы иметь представление, что происходит в уже работающем сервисе, мы применяем самую распространенную библиотеку для записи журналов (logs) в Java - SLF4J. Без дополнительной настройки журналы будут использовать формат по умолчанию (как правило с указанием времени, имени потока thread, и названия объекта, создавшего запись), что обычно вполне достаточно, и выводить записи в стандартный вывод. Как мы знаем из введения в Kubernetes, это основной способ работы с журналами в контейнерах. Более того, инфраструктура Spark также использует SLF4J для записи журналов.

Сборка сервиса и образ контейнера

Концепция облачного приложения или его части в виде отдельного микросервиса диктует упаковку в контейнер, что мы и сделаем с помощью уже хорошо нам знакомого инструмента Docker и его декларации для описания образа контейнера Dockerfile. Для начала используем стандартный для Java способ сборки проекта и описания его зависимостей и библиотек с помощью инструмента Gradle (можно использовать и Maven, но для таких небольших проектов описание сборки проекта с помощью Gradle чуть более кратко и выразительно, чем используемый в Maven XML):

```
1  plugins {
2     id 'java'
3  }
4
5  group 'com.porty.k8s'
6  version '0.1.0-SNAPSHOT'
7
8  sourceCompatibility = 1.8
9
```

```
repositories {
   mavenCentral()
}

dependencies {
   compile "com.sparkjava:spark-core:2.7.2"
   compile "org.slf4j:slf4j-simple:1.7+"
   compile "com.google.code.gson:gson:2.8+"
}

// особый способ упаковки сервиса java в контейнер -
// все библиотеки складываем в "толстый архив" (fat JAR)
jar {
   manifest { attributes "Main-Class": "com.porty.k8s.WeekendService" }
   from {
       configurations.compile.collect { it.isDirectory() ? it : zipTree(it) }
   }
}
```

Файл сборки объявляет название и версию проекта (для первой версии, согласно популярному способу семантического версионирования `semver.org`, обычно используют 0.1.0), зависимости от библиотек Spark, GSON и журналирования SLF4J, и самое интересное, собирает все необходимые зависимости в “толстый” архив JAR, и объявляет его класс с методом `main()`. Это распространенная практика для упрощения создания контейнера для программ Java, чтобы уменьшить копирование обычно немаленького количества требуемых для работы дополнительных библиотек JAR. Собрать наш сервис в “толстый” архив теперь элементарно:

```
$ ./gradlew build
```

В результате мы получим архив со всеми зависимостями в директории проекта `build\libs`.

Создадим образ контейнера, применив технику многоступенчатой сборки, рассмотренную нами в разделе о сборке образов image:

```
# Первая ступень на основе стандартного образа Gradle
FROM gradle:jdk11 as builder

# Директория для файлов проекта
WORKDIR weekend-service/

# Копируем файл сборки и исходный код
COPY build.gradle ./
COPY src/ ./src/

# Сборка
RUN gradle build

# Вторая ступень для окончательной сборки
# Минимальная версия JRE, версия 11, открытая версия OpenJDK
FROM openjdk:11-jre-slim

# Такая же директория
WORKDIR weekend-service/
# Скопируем архив JAR из первой ступени
COPY --from=builder home/gradle/weekend-service/build/libs/weekend-service-0.1.0-SNA\
PSHOT.jar .

# Объявим свой порт
EXPOSE 5678

CMD ["java", "-jar", "/weekend-service/weekend-service-0.1.0-SNAPSHOT.jar"]
```

Детали сборки нашего образа указаны в комментариях в файле `Dockerfile`, и нам уже хорошо знакомы. Для получения чистой сборки мы собираем приложение полностью заново внутри процесса сборки самого образа контейнера. Первая ступень основана на стандартном образе Gradle и JDK 11, мы копируем только файл сборки и исходный код, и тем самым гарантируем новую сборку, не зависящую от состояния библиотек и зависимостей на локальной машине. Полученный архив копируется во вторую ступень, оптимизированную версию (`slim`) JRE версии 11 (на момент написания версии Java 11 для Linux Alpine еще не было, обычно это самый компактный по размеру вариант). При запуске контейнера мы просто вызовем свой метод основной класс и его метод `main()`, записанный в манифесте архива JAR.

Если ваш файл сборки и исходный текст не меняется, то сборка образа будет медленной только в первый раз - в последующие разы сработает кэш. Но, без сомнения, в коде будет много изменений, и для написанного нами `Dockerfile` это означает скачивание всех зависимостей из репозитория Maven Central вновь и вновь при каждой сборке, и только потом идет компиляция кода - это будет занимать немалое время при каждой попытке собрать обновленное приложение.

Существуют решения, ускоряющие этот процесс, например, хранить зависимости приложения так, чтобы они кэшировались отдельно от кода (отдельный слой layer в образе). Лучше всего поможет нам инструмент Jib, созданный в компании Google. Мы его уже кратко упоминали, рассматривая альтернативные методы сборки образов. Чтобы избежать неэффективной и медленной сборки Java-приложений с постоянным скачиванием зависимостей из Интернета, Jib будет использовать локальный кэш Gradle или Maven, и значительно оптимизирует процесс.

Для работы с Jib нужно лишь добавить один плагин:

```
plugins {
    id 'com.google.cloud.tools.jib' version '1.8.0'
}
```

Построим образ, используя все ту же версию 0.1.0 в качестве метки, и после этого отправим его в центральный репозиторий Docker Hub, чтобы образ стал доступен для удаленных кластеров (это необязательно при использовании локального кластера, такого как minikube или Docker):

```
$ docker build . -t {ваша_учетная_запись_Docker}/weekend-service:0.1.0
…
$ docker push {ваша_учетная_запись_Docker}/weekend-service:0.1.0
```

Если вы используете Jib, то построить и отправить образ в репозиторий позволяет следующая простая команда. Повторные сборки образа при изменениях в коде будут в разы быстрее, а размер образа при этом значительно уменьшится, даже по сравнению с нашей двухступенчатой сборкой это будет выигрыш около 50%!

```
$ gradle jib --image={ваша_учетная_запись_Docker}/weekend-service:0.1.0
```

Локальная проверка

Прежде чем погрузиться в создание развертывания Deployment и сервиса Service для кластера Kubernetes, всегда будет отличной мыслью проверить полученный образ контейнера на работоспособность, запустив его на своей локальной машине с помощью того же инструмента Docker и команды `run`:

```
$ docker run -p 5678:5678 {ваша_учетная_запись_Docker}/weekend-service:0.1.0

[Thread-0] INFO org.eclipse.jetty.util.log - Logging initialized @233ms to org.eclip\
se.jetty.util.log.Slf4jLog
[Thread-0] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - == Spark has ignite\
d ...
[Thread-0] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - >> Listening on 0.0\
.0.0:5678
[Thread-0] INFO org.eclipse.jetty.server.Server - jetty-9.4.z-SNAPSHOT, build timest\
amp: 2017-11-21T21:27:37Z, git hash: 82b8fb23f757335bb3329d540ce37a2a2615f0a8
[Thread-0] INFO org.eclipse.jetty.server.session - DefaultSessionIdManager workerNam\
e=node0
[Thread-0] INFO org.eclipse.jetty.server.session - No SessionScavenger set, using de\
faults
[Thread-0] INFO org.eclipse.jetty.server.session - Scavenging every 660000ms
[Thread-0] INFO org.eclipse.jetty.server.AbstractConnector - Started ServerConnector\
@1f813481{HTTP/1.1,[http/1.1]}{0.0.0.0:5678}
[Thread-0] INFO org.eclipse.jetty.server.Server - Started @347ms
```

Запустив контейнер с помощью команды `docker run`, необходимо не забыть перенаправить порт контейнера, который обслуживает веб-сервер, на порт нашей локальной машины, после чего мы увидим процесс запуска веб-сервера библиотеки Spark с помощью журнальных записей в стандартном выводе на консоль. Там же мы увидим, по какому адресу можно найти этот сервер `Started ServerConnector@1f813481 {HTTP/1.1,[http/1.1]} {0.0.0.0:5678}`. Попробуем получить ответ сервера:

```
$ curl 0.0.0.0:5678
<html><body><h2>404 Not found</h2></body></html>
```

Ответ ожидаемый, и приятно видеть что используемая нами инфраструктура Spark позаботилась о форматировании и правильном коде ответа HTTP - в нашем сервисе запрос к корневому адресу `/` не обслуживается. Подтверждение запроса мы увидим в журнале сервиса:

```
[qtp1759478938-18] INFO spark.http.matching.MatcherFilter - The requested route [/] \
has not been mapped in Spark for Accept: [*/*]
```

Попробуем теперь получить день недели, и узнать, выходной ли это, для часового пояса Москвы, в стандартном описании часового пояса:

```
$ curl 0.0.0.0:5678/weekend/Europe/Moscow
<html><body><h2>500 Internal Server Error</h2></body></html>
```

На этот раз ответ не столь приятен - внутренняя ошибка сервера. В идеальном сервисе мы бы позаботились о детальном ответе в формате JSON, но для прототипа и просто примера этого вполне достаточно. Вновь посмотрев на стандартный вывод нашего контейнера с сервисом, мы ожидаемо увидим причину ошибки - требуемый для работы сервис `time-service` не доступен через адрес DNS и свой порт (`UnknownHostException`):

```
[qtp1759478938-19] INFO com.porty.k8s.WeekendService - Запрошен статус выходного дня\
 для зоны Europe/Moscow
[qtp1759478938-19] ERROR spark.http.matching.GeneralError -
java.net.UnknownHostException: time-service
        at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:184)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
        at java.net.Socket.connect(Socket.java:589)
        at java.net.Socket.connect(Socket.java:538)
        at sun.net.NetworkClient.doConnect(NetworkClient.java:180)
        at sun.net.www.http.HttpClient.openServer(HttpClient.java:463)
           ...
```

Чтобы получить доступ к дополнительным микросервисам через DNS, нам уже не обойтись простым запуском второго контейнера `time-service`. Чуть позже мы обсудим, какие подходы существуют в тестировании микросервисов на локальной машине без запуска новых сервисов, развертываний, и кластеров. Сейчас давайте запустим свой сервис на кластере Kubernetes.

Создание объектов Kubernetes и развертывание сервиса

Вновь применяя уже знакомый нам по предыдущей главе подход `kubectl --dry-run` для получения шаблонов YAML для объектов Kubernetes, создадим заготовки с описанием объекта Deployment для развертывания Kubernetes, и поменяем имя образа и его версию:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
   app: weekend-service
 name: weekend-service
spec:
 selector:
   matchLabels:
     app: weekend-service
 template:
   metadata:
     labels:
       app: weekend-service
   spec:
     containers:
       - image: ivanporty/weekend-service:0.1.0
         name: weekend-service
```

Мы применили практически идентичный с нашим первым микросервисом для получения времени подход, используя метку (label) `app` с именем сервиса, и указав образ контейнера и его версию в секции `template`, которая и будет определять, какие контейнеры будут запускаться при создании новых отсеков развертывания.

Практически аналогичным будет и объект сервиса Service:

```
apiVersion: v1
kind: Service
metadata:
 labels:
   app: weekend-service
 name: weekend-service
spec:
 ports:
   - port: 5678
 selector:
   app: weekend-service
 type: NodePort
```

Нам остается удостовериться, что сервис выбирает правильные отсеки по меткам label, указать открытый порт, и сделать этот порт доступным на всех отсеках кластера, используя тип сервиса `NodePort`.

Файлы с описанием объектов развертывания и сервиса мы разместили в папке `k8s`, и теперь вновь можем применить практически магическую команду `apply`, чтобы привести состояние своего кластера в соответствие с состоянием объекта в наших описания YAML:

```
$ kubectl apply -f k8s/
deployment.apps/weekend-service created
service/weekend-service created
```

При первом запуске развертывание и сервис будут созданы, а при последующим запусках уже модифицированы (при необходимости конечно), и все детали и разницу в состояниях возьмет на себя управляющий узел Kubernetes. Мы уже должны иметь работающий сервис получения времени на своем кластере из предыдущих тестов (если нет, разверните микросервис `time-service` снова), и проверив состояние кластера, мы увидим, что в нем развернуты и доступны наши два сервиса:

```
$ kubectl get deploy
NAME  ...
time-service         ...
weekend-service    …

$ kubectl get service
NAME       ...
time-service      NodePort    ...       8080:31250/TCP
weekend-service   NodePort  ...        5678:31056/TCP
```

Остается проверить, что сервис и развертывание работают, и на этот раз вспомогательный микросервис доступен с помощью встроенной в Kubernetes системы DNS. Напрямую получить доступ к сервису, если это не локальный кластер Docker и Minikube, у нас не получится, но как мы знаем из первых опытов с Kubernetes, мы всегда сможем использовать услугу переадресации порта `port-forward` инструмента `kubectl`, запустив и оставив ее работать в отдельном терминале. Дальше мы сможем запросить свой новый сервис и узнать о дне в запрошенном часовом поясе:

```
$ kubectl port-forward deploy/weekend-service 5678
Forwarding from 127.0.0.1:5678 -> 5678
Forwarding from [::1]:5678 -> 5678
…

$ curl 0.0.0.0:5678/weekend/Europe/Moscow
{"weekend":false,"day":"FRIDAY"}
$ curl 0.0.0.0:5678/weekend/Asia/Tokyo
{"weekend":true,"day":"SATURDAY"}
```

Как мы видим, теперь наш новый сервис работает, и запрашивает вспомогательный микросервис для получения текущего времени, а затем преобразует его во время переданного ему часового пояса, возвращая ответ в формате JSON. К сожалению, в момент теста выходной день в Москве еще не настал, но это была пятница - тоже неплохо. Зато к этому моменту уже настала суббота в Токио - наш сервис подтверждает, что дни календаря относительны.

Запросив журнал (log) для развертывания, мы увидим записи успешно работающего вебсервера и обработку запросов:

```
$ kubectl logs deploy/weekend-service
…
[Thread-0] INFO org.eclipse.jetty.server.AbstractConnector - Started ServerConnector\
@2a2cb2dc{HTTP/1.1,[http/1.1]}{0.0.0.0:5678}
[Thread-0] INFO org.eclipse.jetty.server.Server - Started @756ms
…
[qtp1759478938-19] INFO com.porty.k8s.WeekendService - Запрошен статус выходного дня\
 для зоны Europe/Moscow
[qtp1759478938-19] INFO com.porty.k8s.WeekendService - Запрошен статус выходного дня\
 для зоны Asia/Tokyo
```

Локальное тестирование взаимодействующих сервисов

Запустив свой второй микросервис, взаимодействующий с помощью сетевых вызовов с сервисом получения времени `time-service`, мы столкнулись с проблемой локального запуска распределенной многокомпонентной системы. Проверить даже простейшую функциональность на локальной машине без запуска всех необходимых компонентов кажется невозможным. Более того, требуются и услуги Kubernetes, в нашем случае обнаружение сервисов через DNS. Мы запустили все сервисы на своем кластере Kubernetes, и, к счастью, они заработали, но что если возникла проблема, требующая отладки? Мало что сравнится с легкостью и эффективностью отладки в редакторе IDE, где можно проверить все состояние программы в удобной точке ее исполнения.

Существует несколько вариантов решения проблемы отладки микросервисов, работающих в кластере Kubernetes. Вот основные из них:

- Локальный запуск взаимодействующих сервисов. Один вариант - запуск всех микросервисов внутри виртуальной машины Docker, и организация их взаимодействия с помощью оркестратора Docker Compose. Есть множество минусов - необходимо поддерживать дополнительную конфигурацию Compose, недоступны сервисы Kubernetes, а также доступ к библиотекам и функциям API облака, в котором работает приложение. Второй вариант - использование локального эмулятора `minikube`. Несомненный плюс -

идентичная настоящему кластеру конфигурация Kubernetes, однако остальные минусы остаются - лишь один узел (node), возможный недостаток вычислительных ресурсов на вашей локальной машине, и опять недоступные дополнительные облачные сервисы и базы данных.

- Перенаправление сетевого трафика от локального контейнера в кластер Kubernetes и обратно. Эту возможность предоставляет отличный инструмент Telepresence (telepresence.io). Тонкая работа с настройками сети и прокси-компоненты на обеих сторонах (вашей локальной машине в кластере) создают иллюзию работы локального контейнера в удаленном кластере Kubernetes. У вас должно быть достаточно прав доступа к кластеру Kubernetes, чтобы установить дополнительные компоненты, и затем вы сможете не только запустить микросервис локально, не теряя доступа к удаленным сервисам-партнерам, но и отладить ваш код обычным образом из вашего редактора, как если бы все сервисы были доступны на вашей машине. Очень мощный инструмент, заслуженно ставший популярным и часто представленный на конференциях и в статьях. Детали настройки довольно обширны, вы найдете все необходимое на сайте Telepresence.
- Удаленная отладка микросервиса, уже развернутого в кластере Kubernetes, как правило предназначенного для тестирования (QA) или отработки новых изменений (staging). Практически все языки передают состояние для режима отладки через сетевой порт, используя собственный протокол. Для Java к примеру, это будет протокол JDWP (Java debug wire protocol). Зная, как работает сервис Kubernetes, нетрудно понять, что понадобится создать версию сервиса для отладки, открыв в этом сервисе порт, через который будет передаваться отладочная информация. Конечно же, потребуется и отдельный вариант образа для контейнера микросервиса, в котором он будет запущен в режиме отладки (обычно это несколько ключей командной строки или специальный режим сборки). Приготовления для того, чтобы просто поставить точку останова в программе, немаленькие, и есть множество решений, автоматизирующие подобную отладку, от создания специального образа до модификации сервиса, например Skaffold или Google Cloud Code. Некоторые другие инструменты можно так же найти на сайте книги www.ipsoftware.ru.

Проверка готовности сервиса к работе

Мы постоянно упоминаем о том, что кластер Kubernetes и работающие в нем контейнеры (в отсеках) с модулями и микросервисами приложения - очень динамичная система. Обновления, перезапуск остановившихся контейнеров, новые версии модулей все время меняют состояние приложения. Обращение по виртуальному IP-адресу может привести нас к совершенно разным экземплярам и даже версиям сервисов, и состояние сервиса может быть еще не готовым, если он был только что запущен.

В подобных распределенных системах хорошей проверенной практикой считается поддержка каждым сервисом сигнала его готовности к работе (readiness check). Как правило, готовность сервиса к работе может значительно отличаться от момента непосредственного

запуска его процесса. В примере с нашим микросервисом выходного дня это особенно легко будет увидеть, случись нам использовать его под нагрузкой и в процессе непрерывного обновления - это приложение Java, которое, как хорошо известно, требует некоторого времени для выделения памяти, загрузки классов, и “прогрева” виртуальной машины. Все эти мгновения процесс уже будет запущен, однако сервис будет недоступен. Это может привести к значительному замедлению распределенной системы из множества сервисов, и к возможному сбою многоступенчатой операции.

Если же мы реализуем хороший сигнал готовности к работе, наш контейнер (и его отсек) никогда не будут вызываться через сервисы Kubernetes и их виртуальные IP-адреса, до тех пор пока сигнал не станет положительным. После этого экземпляр сервиса становится доступен.

Kubernetes поддерживает несколько стандартных сигналов готовности сервиса - самый распространенный и как правило удобный - обращение по определенному маршруту по протоколу HTTP. Описание проверки готовности к работе добавляется к объекту развертывания Deployment, в под-объект `template/spec/containers` - в этом есть смысл, ведь именно описание шаблона для всех будущих контейнеров (template) определяет, как они будут создаваться, а объект Deployment управляет набором отсеков с контейнерами приложения или его модуля/сервиса.

Давайте добавим проверку готовности в развертывание сервиса, только что нами созданного:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
   app: weekend-service
 name: weekend-service
spec:
 selector:
   matchLabels:
     app: weekend-service
 template:
   metadata:
     labels:
       app: weekend-service
   spec:
     containers:
       name: weekend-service
       # секция проверки готовности
       - readinessProbe:
           # проверка готовности с помощью запроса HTTP
           httpGet:
             path: /ready
```

```
23              port: 5678
24            periodSeconds: 2
25            initialDelaySeconds: 5
26            failureThreshold: 1
27            successThreshold: 1
28        - image: ivanporty/weekend-service:0.2.0
```

Итак, проверка готовности описана в разделе `readinessProbe`. Сразу же идет тип проверки - у нас это будет запрос по адресу (`/ready`) и порту HTTP (`httpGet`). Можно отрегулировать проверку параметрами:

- `periodSeconds`. Время в секундах между проведением очередной проверки. Проверка проводится регулярно и постоянно, чтобы быстро выявить не готовые контейнеры.
- `initialDelaySeconds`. Задержка перед первым вызовом проверки, если мы точно знаем что сервис сразу готов не будет. Это полезно в нашем случае с Java.
- `failureThreshold` и `successThreshold`. Количество попыток для объявления сервиса не готовым, и наоборот, готовым. Мы поставили везде 1 - одной удачной или неудачной попытки хватит для смены состояния сервиса. Можно менять эти параметры в зависимости от природы вашего микросервиса.

Проверка готовности будет определяться возвращаемыми кодами HTTP - успешные коды (от 200 и меньше 400) будут означать готовность сервиса к работе, любые другие - неготовность.

Обратите также внимание на то, что мы увеличили номер версии в своей декларации развертывания YAML - теперь это версия 0.2.0. Как мы помним, одним из основных столпов концепции приложений, разрабатываемых для облака (cloud native), является их неизменяемость. Мы уже создали версию 0.1.0, упаковали ее в контейнер, и, самое важное, отправили эту версию на централизованный репозиторий. После этого контейнер развернули в кластере. Теперь мы можем быть уверены в том, что образ контейнера версии 0.1.0 всегда будет содержать в себе именно эту функциональность и зависимости, вне зависимости от того, какие изменения мы произвели после этого, и сколько времени прошло после выпуска первой версии. Именно это позволяет нам точно воспроизводить состояние системы и возможность всегда запустить ее, не опасаясь никаких потерь в процессе ее эволюции. Любые последующие изменения должны быть упакованы в новые образы с новыми версиями.

Однако это теория, а на практике нам остается включить поддержку нового маршрута (`/ready`) в свой сервис и реализовать логику, которая определит точный момент готовности сервиса к обработке запросов. На наше счастье, в библиотеке Spark есть для этого простой метод. Вот что у нас получится:

```
...
private static AtomicBoolean ready = new AtomicBoolean(false);

public static void main(String[] args) {
   port(5678);

  …

        // поддержка сигнала о готовности сервиса к работе. Используем
        // встроенный в библиотеку Spark
        // метод для ожидания готовности сервера к работе, ожидая его в
        // отдельном потоке Thread.
        get("/ready", (req, res) -> {
            if (ready.get()) {
                logger.info("Запрос проверки готовности, сервер готов.");
                return "Готов!";
            } else {
                logger.warn("Запрос проверки готовности, сервер не готов.");
                throw new IllegalStateException("Не готов!");
            }
        });
        // ожидаем готовности в параллельном потоке.
        new Thread(() -> {
            awaitInitialization();
            ready.set(true);
        }).start();
```

Мы добавили новый маршрут (`/ready`) и сделали его зависимым от значения флага `ready`, который будет выставлен в отдельном потоке, как только синхронный метод `awaitInitialization()` вернет потоку управление, что и будет означать готовность встроенного сервера Spark к приему запросов. В противном случае мы выбросим исключение, что автоматически приведет к ответу с кодом HTTP 500 (Internal Error). Система проверки готовности Kubernetes в этом случае выведет наш контейнер из списка готовых к приему запросов.

Повторим наш цикл разработки - построим новую версию образа со своим микросервисом, отправим ее в центральный репозиторий образов, запросим обновление состояния кластера Kubernetes с помощью команды `apply`:

```
$ docker build . -t {ваша_учетная_запись_Docker}/weekend-service:0.2.0
…
$ docker push {ваша_учетная_запись_Docker}/weekend-service:0.2.0
…
$ kubectl apply -f k8s/
deployment.apps "weekend-service" configured
service "weekend-service" unchanged
```

Чем больше будет в вашей распределенной системе модулей и микросервисов, чем чаще они будут обновляться и масштабироваться, тем более жизненно необходимой будет качественная проверка готовности сервиса к работе. Еще одна проверка в Kubernetes - проверка жизнеспособности (liveness) контейнера, отличается тем, что проверяется возможность контейнера продолжать работу в принципе, или же необходимость его полностью перезапустить. Мы узнаем о ней в подробном описании развертываний (deployment).

Резюме

- Сервисы в Kubernetes предоставляют доступ к экземплярам вашего работающего в кластере микросервиса или приложения через сетевой порт. Найти адрес сервиса удобнее всего через обнаружение имен DNS.
- Сервис Kubernetes использует специальные настройки внутренней сети кластера, прокси-компонент kube-proxy, и виртуальные IP-адреса, чтобы обеспечить прозрачный доступ к динамичному, имеющему склонность к постоянным обновления и перезапускам развертыванию Deployment.
- Внутренняя реализация сервисов позволяет прозрачно работать с любым количеством экземпляров ваших микросервисов и включает в себя простые алгоритмы балансировки нагрузки. Для поиска экземпляров используются метки labels.
- Кроме обычного адреса внутри кластера, вы также можете выбрать тип сервиса NodePort для собственного алгоритма балансировки нагрузки, или использовать встроенный балансировщик облачного кластера с помощью типа LoadBalancer.
- Апробированной и рекомендованной практикой распределенных динамических систем является проверка готовности (readiness check) каждого работающего в кластере сервиса. Kubernetes предоставляет нам встроенную проверку готовности нескольких видов.

8. Метки и аннотации Kubernetes. “Канарейки”. Service Mesh.

В прошлом разделе мы рассмотрели первые объекты Kubernetes, развертывания Deployment и сервисы Service. В них мы указали простейшие *метки* (labels), которые в нашем начальном случае совпадали для двух созданных объектов, и позволяли каким-то образом соединить развертывания и сервисы в единую работающую систему.

Метки - это произвольный набор текстовых ключей и их значений, который позволяет производить поиск любого объекта среди всех объектов одного типа в кластере Kubernetes, или объединять объекты по какому-либо признаку. Можно представить их в виде бирок на чемоданах в аэропорту - именно они позволяют грузовой службе различить практически одинаковые чемоданы друг от друга, и выбрать те, что направляются в один аэропорт.

Основная задача меток в кластере - обеспечить эффективную организацию объектов по категориями (которые в общем случае зависят от нужд вашего конкретного приложения) и обеспечить работу селекторов (selector), выбирающих набор объектов, например, для сервисов Kubernetes.

Обычно метки применяются для следующих целей:

- “Канареечные” выпуски обновлений (canary releases). Близко к первому случаю, красивое название связано с историей о том, как шахтеры брали с собой в забой чрезвычайно чувствительных к опасным газам канареек, чтобы узнать об опасности заранее. В случае же нового сервиса или обновления небольшая часть пользователей или сервисов-клиентов переключается на новую версию, чтобы в реальных условиях проверить работоспособность новых изменений. Очень эффективная техника, которую легко реализовать с помощью меток.
- Безопасное переключение между версиями микросервиса в одном кластере. Такое переключение еще называется “сине-голубым” развертыванием (blue-green deployment). В случае нашего примера `time-service`, мы можем захотеть обновить его, и добавить новую точку доступа и новый формат времени, и перейти на новую версию мы хотим сразу же, для всех клиентов и всех экземпляров микросервиса. С помощью меток мы можем создать новое развертывание с меткой `version: green`, и как только оно будет готово (все отсеки pods развернуты и запущены), переключить сервис `Service` на новые отсеки (pods) и контейнеры в них, имеющие метку `green`.
- Разбиение вычислительных потоков или связанных с ними потоков данных (partitions). Если алгоритм данных подразумевает разбиение данных и параллельную обработку, вместо управления этим процессом в коде, можно использовать одинаковые сервисы, а разбиение данных и их перенаправление устроить с помощью меток и сервисов.

Ваши конкретные нужды могут отличаться от этих основных случаев, но основная задача меток в кластере - гибко разделить одинаковые объекты на категории и управлять ими и их потоками данных, не усложняя код и конфигурацию микросервисов и приложений.

Одно из основных, самых фундаментальных применений меток - сервисы `Service`, с которым мы уже знакомы. Используя указанный при его создании набор меток, сервис `Service` выбирает набор отсеков и направляет к одному из них (обычно случайно выбранному) очередной запрос от клиента.

И еще - имя метки не должно превышать 63 символов, ее значение - 253. В общем случае лучше не использовать нестандартных символов. Детали легко найти в документации.

Метки на практике. “Канареечное” развертывание

Давайте попробуем применить метки на практике. Запустим в своем кластере сразу две версии `time-service`, одну классическую, созданную нами в первых главах, а вторую обновленную. Представим однако, что у нас сотни или тысячи существующих пользователей, чье доверие, как хорошо известно, очень тяжело завоевать и проще простого потерять. Цена ошибки велика, а найти все ошибки в сложной распределенной системе из микросервисов весьма непросто и дорого.

Идея “канареечного” развертывания заключается в том, чтобы в приложении одновременно сосуществовали две версии одной и той же функциональности. Как правило, проверенная, предыдущая версия преобладает, а новая версия обслуживает намного меньшее количество запросов. Классическим запуском приложений на выделенных серверах сделать подобное непросто, а управлять еще сложнее. Но как оказывается, самые простые метки Kubernetes помогут нам развернуть первое элементарное “канареечное” развертывание.

Представим, что мы обновили сервис `time-service` следующим образом:

```
…

func main() {
  log.Print("Начало работы сервиса time-service")

  http.HandleFunc("/time", serveTime)
  http.HandleFunc("/nanotime", serveNanoTime)
  log.Fatal(http.ListenAndServe("0.0.0.0:8080", nil))
}

func serveTime(w http.ResponseWriter, r *http.Request) {
  log.Print("Вызов функции serveTime()")
  var serverTime Time
  serverTime.Time = time.Now().Format("02 Jan 2006")
```

```
15    json.NewEncoder(w).Encode(serverTime)
16  }
17
18  ...
```

По сути ничего не поменялось, только более привычный нам формат даты в методе `Format()`, которую мы отправляем при запросе ресурса `/time`, вместо чрезмерно неудобного результата функции `String()`. Все шаги по построение приложения и упаковке его в образ контейнера будут неизменны, мы изменим лишь версию, увеличив ее на следующую минимальную 0.2.0:

```
1  $ docker build . -t {ваша_учетная_запись_Docker}/time-service:0.2.0
2  $ docker push {ваша_учетная_запись_Docker}/time-service:0.2.0
```

К этому моменту в нашем репозитории Docker Hub хранятся две версии `time-service`. Давайте развернем в нашем кластере вторую версию, только укажем для наши объектов дополнительную метку, чтобы отличить отсеки от предыдущей, стабильной версии. Для “канареечных” версий часто используют метку `release: canary`.

Начнем с развертывания `Deployment`. Откроем описание развертывания в YAML и создадим новый вариант версию, назвав его `k8s-time-deploy-canary.yaml` и сохранив во вложенной директории `k8s/canary`:

```
1  # Версия программного интерфейса Kubernetes
2  apiVersion: apps/v1
3  # Тип объекта
4  kind: Deployment
5  # Метаданные нашего объекта, вложенный объект ObjectMeta
6  metadata:
7     # список меток канареечного развертывания
8    labels:
9      app: time-service
10     release: canary
11   name: time-service-canary
12 # Описание собственно правил развертывания контейнера
13 # Вложенный объект DeploymentSpec
14 spec:
15   # Количество запущенных отсеков pods для масштабирования
16   replicas: 1
17   selector:
18     matchLabels:
19       app: time-service
20       release: canary
```

```
21     # описание шаблона для создания новых отсеков
22   template:
23     metadata:
24       # список меток для канареечных отсеков pods
25       labels:
26         app: time-service
27         release: canary
28     # непосредственно описание контейнера в отсеке
29     spec:
30       containers:
31       - image: ivanporty/time-service:0.2.0
32         name: time-service
```

Единственное отличие нашего нового развертывания - новая метка в шаблоне `spec.template` и новая версия образа для запускаемых контейнеров - это означает, что всем создаваемым в этом развертывании отсекам pods будет присваиваться дополнительная метка `release: canary`.

Давайте развернем оба развертывания одновременно - конечно же, декларативно, просто указав директорию со всеми нашими объектами в YAML:

```
1 $ kubectl apply -f k8s/
2 deployment.apps/time-service created
3 service/time-service created
4 $ kubectl apply -f k8s/canary/
5 deployment.apps/time-service-canary created
```

Мы видим, что теперь у нас два развертывания - стабильное, с проверенной версией образа контейнера и сервиса, и новое, “канареечное”, для “притирки” и сбора информации о том, как обновленный микросервис работает в реальных условиях. Однако сервис Kubernetes у нас по прежнему один, и он выбирает все отсеки с отметками `app: time-service`, то есть любые отсеки, неважно, есть ли у них метка `release: canary`, или нет.

```
1 …
2   # по этим меткам идет поиск отсеков, куда отправляются запросы
3   selector:
4     app: time-service
```

Между прочим, это не что иное, как простое “канареечное” развертывание (canary deployment)! Доступ через сервис будет выбирать любые отсеки и контейнеры с метками `app: time-service` в случайном порядке, и мы получим возможность сравнить, как работает новая функциональность, полностью не отказываясь от старой проверенной версии. С другой стороны, мы имеем полный контроль над тем, сколько “канареечных” контейнеров работает в нашем

кластере, и в любой момент можем изменить их количество (просто поменяв число экземпляров replicas), или, в случае обнаруженной ошибки, немедленно удалить “канареечное” развертывание и откатить систему в стабильное состояние.

Проверить как это работает весьма просто - давайте запрашивать `time-service` в простом цикле, и вот что мы увидим:

```
$ while true; do curl localhost:[порт_NodePort]/time; sleep 1; done
{"time":"2019-10-28 13:56:59.6738709 +0000 UTC m=+111.618738301"}
{"time":"2019-10-28 13:57:00.7030272 +0000 UTC m=+112.647894601"}
{"time":"28 Oct 2019"}
{"time":"28 Oct 2019"}
{"time":"28 Oct 2019"}
{"time":"2019-10-28 13:57:04.82033 +0000 UTC m=+116.765197501"}
{"time":"2019-10-28 13:57:05.8119032 +0000 UTC m=+117.791169801"}
{"time":"28 Oct 2019"}
```

Как мы видим, сервис случайным образом отвечает или стабильной существующей версией, или новым форматом данных “канареечного” выпуска. Кстати, доступ к сервису мы получили по номеру порта `NodePort` - напрямую, если это локальный кластер Docker, или, в случае облачного кластера, с помощью перенаправленного порта (`kubectl port-forward`).

Настраивая экземпляры `replicas`, мы можем уменьшить или увеличить долю “канареечной” версии в работе системы. Еще один вариант применить “канареечное” развертывание - объект для входящего Интернет-трафика `Ingress`, используя их, возможно назначить отдельные развертывания на выделенные маршруты HTTP. Вы сможете найти детали в документации этого объекта.

Сине-голубое развертывание

Что еще можно легко сделать с метками? Представим себе, что мы хотим очень аккуратно перевести свой микросервис на новую версию, предварительно проверив, что все отсеки и контейнеры готовы к работе, и только потом переключить всю систему на новую версию. Запустим в кластере совершенно отдельную, новую версию `time-service`, и для простоты назовем ее “зеленой”. Все микросервисы для этой версии будем помечать довольно часто использующейся меткой `version: green`.

Создадим развертывание `Deployment` в файле `k8s/blue-green/k8s-time-deploy-green.yaml`, примерно так же, как с “канареечным” развертыванием, только используем другой набор меток и назовем это развертывание `time-service-green` (мы помним, все объекты одного типа должны иметь уникальное название):

```
# Версия программного интерфейса Kubernetes
apiVersion: apps/v1
# Тип объекта
kind: Deployment
# Метаданные нашего объекта, вложенный объект ObjectMeta
metadata:
   # список меток зеленого развертывания Deployment
  labels:
    app: time-service
    version: green
  name: time-service-green
# Описание собственно правил развертывания контейнера
# Вложенный объект DeploymentSpec
spec:
  # Количество запущенных отсеков pods для масштабирования
  replicas: 1
  selector:
    matchLabels:
      app: time-service
      version: green
   # описание шаблона для создания новых отсеков
  template:
    metadata:
      # список меток зеленых отсеков
      labels:
        app: time-service
        version: green
    # новая версия 0.2.0
    spec:
      containers:
      - image: ivanporty/time-service:0.2.0
        name: time-service
```

Новые отсеки будут иметь метку `version: green`. Если мы развернем новую версию микросервиса, но не поменяем сервис Kubernetes, то вновь получим “канареечное” развертывание, так как отсеки и старого, и нового развертывания будут подходить по критерию поиска (`app: time-service`). Нам надо полностью разделить стабильную версию сервиса от новой (“зеленой”). Назовем стабильную версию “синей” и добавим “синие” метки в наше старое развертывание и сервис. Все изменения поместим для простоты в отдельные директории (`k8s/blue-green`):

```
# Версия программного интерфейса Kubernetes
apiVersion: apps/v1
# Тип объекта
kind: Deployment
# Метаданные нашего объекта, вложенный объект ObjectMeta
metadata:
   # список меток синего развертывания Deployment
  labels:
    app: time-service
    version: blue
  name: time-service
# Описание собственно правил развертывания контейнера
# Вложенный объект DeploymentSpec
spec:
  # Количество запущенных отсеков pods для масштабирования
  replicas: 1
  selector:
    matchLabels:
      app: time-service
      version: blue
   # описание шаблона для создания новых отсеков
  template:
    metadata:
      # список меток синих отсеков
      labels:
        app: time-service
        version: blue
    # старая версия 0.1.0
    spec:
      containers:
      - image: ivanporty/time-service:0.1.0
        name: time-service
```

Теперь наша стабильная версия развертывания имеет метки `version: blue`, поменяем наш сервис, чтобы он искал только “синие” отсеки:

```
# Версия программного интерфейса Kubernetes
apiVersion: v1
# Тип объекта
kind: Service
metadata:
  labels:
    app: time-service
    version: blue
  name: time-service
spec:
  # список портов. Дополнительно можно указать протокол
  ports:
  - port: 8080
  # по этим меткам идет поиск синих отсеков
  selector:
    app: time-service
    version: blue
  # тип сервиса. В облаке можно использовать LoadBalancer
  type: NodePort
```

Обновим наш кластер, и сделаем стабильную, старую версию доступной только через “синие” метки (Внимание: при изменении меток развертывания данное развертывание сначала надо удалить, а только потом запустить заново!):

```
$ kubectl apply -f k8s/blue-green/blue/
deployment.apps/time-service created
service/time-service created
```

Теперь мы можем запустить обновленную “зеленую” версию развертывания, не опасаясь того, что к ней будут попадать реальные запросы:

```
$ kubectl apply -f k8s/blue-green/green/k8s-time-deploy-green.yaml
deployment.apps/time-service-green created
```

Наш сервис теперь не будет обращать внимания на параллельно работающую “зеленую” версию, так как настроен на “синюю” версию:

```
$ while true; do curl localhost:[порт_NodePort]/time; sleep 1; done
{"time":"2019-10-30 16:24:03.4171022 +0000 UTC m=+165.410751801"}
{"time":"2019-10-30 16:24:04.4479167 +0000 UTC m=+166.441567201"}
...
```

Итак, у нас в кластере запущены полностью раздельные две версии микросервиса `time-service`. Мы в любой момент можем переключить сервис с “синих” на “зеленые” метки. Напишем YAML для “зеленого” варианта сервиса (он будет иметь то же имя, просто другой селектор, так что клиенты могут продолжать использовать этот сервис без каких-либо изменений, по тому же DNS-имени):

```
# Версия программного интерфейса Kubernetes
apiVersion: v1
# Тип объекта
kind: Service
# Метаданные нашего объекта, вложенный объект ObjectMeta
metadata:
  labels:
    app: time-service
    version: green
  name: time-service
spec:
  # список портов. Дополнительно можно указать протокол
  ports:
  - port: 8080
  # список меток зеленых отсеков для отправки запросов
  selector:
    app: time-service
    version: green
  # тип сервиса. В облаке можно использовать LoadBalancer
  type: NodePort
```

Торжественно (и немного волнуясь) переведем систему на “зеленую” версию:

```
$ kubectl apply -f k8s/blue-green/green/k8s-time-svc-green.yaml
service/time-service configured
$ while true; do curl localhost:31297/time; sleep 1; done
{"time":"30 Oct 2019"}
{"time":"30 Oct 2019"}
...
```

Одной строкой мы мгновенно переключили `time-service` на новую версию, а клиенты совершенно незаметно для себя стали использовать ее через то же самое имя DNS.

Еще не время удалять “синюю” версию! В случае непредвиденных проблем и краха “зеленой” версии мы можем мгновенно откатиться на старую версию - ведь “синии” отсеки все еще работают в кластере:

```
$ kubectl apply -f k8s/blue-green/blue/k8s-time-svc-blue.yaml
service/time-service configured
$ while true; do curl localhost:31297/time; sleep 1; done
{"time":"2019-10-30 16:31:02.1427651 +0000 UTC m=+584.591143901"}
{"time":"2019-10-30 16:31:03.1720833 +0000 UTC m=+585.620463701"}
...
```

Именно это и называется “сине-зеленым” развертыванием (blue-green deployment). У нас в кластере одновременно развернуты две версии одного и того же микросервиса, которые совершенно не знают о существовании друг друга. Сервис же один, и он может указывать или на “синий”, или на “зеленый” сервис с помощью простейших меток. Каждый следующий цикл обновления может чередовать цвет - в следующее обновление в нашем примере уже “зеленая” версия будет стабильной, а “синяя” придет ей на замену.

Обратите внимание, насколько легко было организовать динамическое управление версиями и трафиком в кластере с помощью простейшей метки! В это вся суть Kubernetes. Без оркестратора контейнеров организовать подобное вручную было бы непросто и требовало бы работы опытного администратора. Это еще не все - чуть позже мы увидим, как реализовать еще более удобное и простое непрерывное обновление с помощью того же объекта `Deployment`.

Шаблоны YAML - Kustomize

И еще, как легко заметить, количество файлов YAML с различными минимальными отличиями стало быстро увеличиваться для каждой новой стратегии, реализуемой в кластере, будь это “канареечное” или “сине-голубое” развертывание. Зачастую ядро системы, базовые ее настройки совершенно одинаковы, как в нашем примере, а отличие заключается лишь в номере версии образа контейнера и паре дополнительных меток labels.

Поддерживать большое количество одинаковых фрагментов YAML непросто и приводит к ошибкам, которые трудно отладить и понять. Здесь помогают различные инструменты для работы с шаблонами и многослойным построением итогового файла YAML объекта Kubernetes. Один из лучших вариантов - Kustomize. Его использование позволит в нашем случае оставить один главный объект `Service` и `Deployment` в формате YAML, а дополнительные мелкие настройки версий и меток перенести в дополнительные накладываемые на базовый файл фрагменты (overlays). Детали вы найдете в Интернете.

Ручная отладка отсеков Pods и контейнеров с помощью меток

Метки можно задавать и удалять не только в декларациях объектов YAML, но и вручную, с помощью команды `kubectl label`. Казалось бы, это путь в никуда, ручное императивное управление объектами, как мы помним, через какое-то время приведет к системе, состояние которой было достигнуто потерянными и забытыми командами. Но в некоторых случаях это может быть очень полезно. Посмотрим на список отсеков pods развертывания `time-service`, у которых есть метки `app`, основная метка, по которой их выбирает сервис:

```
$ kubectl get pods --selector app=time-service
NAME                                  READY   STATUS    RESTARTS   AGE
time-service-564b4d479f-tjgx7         1/1     Running   0          22h
time-service-canary-54ccf7c869-s8snk  1/1     Running   0          3d
time-service-green-5c4497bbf9-m59sx   1/1     Running   0          22h
```

Как мы видим, можно указать селектор для команды `kubectl` вручную. Мы видим три отсека для текущего развертывания, “канареечного” и “зеленого”. Представим, что наше “канареечное” развертывание работает в нескольких экземплярах, и что-то пошло не так. Нам хотелось бы взять текущий экземпляр сервиса и контейнера `time-service`, но все они обслуживают реальные запросы. Нестандартное решение - убрать основную метку селектора отсека! Развертывание `Deployment` “потеряет” этот экземпляр микросервиса и запустит новый отсек, а сервис перестанет направлять к нему запросы.

```
$ kubectl label pod time-service-canary-54ccf7c869-s8snk debug=true app-
pod/time-service-canary-54ccf7c869-s8snk labeled
```

Команда `kubectl label` позволяет одновременно добавлять и удалять метки. Здесь мы удалили нашу основную метку для селекторов `app` (добавив минус к ее имени), и добавили новую `debug`.

Посмотрим, как управляющий цикл исправит потерю отсека:

```
$ kubectl get pods --selector app=time-service
NAME                                  READY   STATUS    RESTARTS   AGE
time-service-564b4d479f-tjgx7         1/1     Running   0          22h
time-service-canary-54ccf7c869-n6zgb  1/1     Running   0          31s
time-service-green-5c4497bbf9-m59sx   1/1     Running   0          22h
```

Как видим, для “канареечного” развертывания был запущен новый отсек, и система вернулась в желаемое состояние (desired state).

А мы получили желаемый экземпляр прежде рабочего микросервиса в свое распоряжение, и можем изучить его состояние и понять, в чем может быть проблема:

```
$ kubectl get pods --selector debug=true,release=canary
NAME                                  READY   STATUS    RESTARTS   AGE
time-service-canary-54ccf7c869-s8snk  1/1     Running   0          3d
```

Эта нехитрая, но действенная техника может быть очень полезной для “горячей” отладки прямо на работающем в производственном режиме кластере, без потери функциональности и желаемого состояния кластера.

Сетка микросервисов - Service Mesh

Мы уже смогли насладиться тем, как несколько меток позволили нам элементарно реализовать сложные (без помощи Kubernetes) техники “канареечного” и “сине-голубого” развертывания. Однако все они работают на уровне перенаправления запросов к индивидуальным контейнерам, работающим в отсеках (pods). Чтобы изменить пропорцию запросов, отправляемых к “канареечной” версии, надо запускать новые отсеки, а это ресурсы, прежде всего процессора и памяти, они не бесконечны, а случае коммерческого облака, еще и весьма дороги.

Если ваш микросервис не такой уж и “микро”, например большой, полноценный сервер, реализовать такие развертывания будет дорого, а иногда и невозможно по причине нехватки ресурсов кластера. В этом случае оптимально было бы управлять запросами и сетевым трафиком на уровне данных, непосредственно идущих через сеть. К примеру, имея только два отсека, обычный и “канареечный”, некий сетевой компонент мог бы гибко направлять запросы между ними, в зависимости от указанной пропорции.

Именно это и делают сетки микросервисов (service mesh). Устоявшегося перевода этого термина нет, но назвать их сетью было бы неверно, так как сеть это общее понятие передачи данных (network). Задача же service mesh - объединить микросервисы в кластере Kubernetes в единое управляемое, наблюдаемое пространство, делать это упорядоченно - то есть распределить их в некую правильную, практически геометрическую, сетку.

Основные задачи сеток service mesh:

- Гибкое управление сетевыми потоками данных и трафиком между микросервисами. Пример - то самое “канареечное” развертывание, но на этот раз уже на уровне сетевых потоков.
- Тестирование А/Б. Вариант “канареечного” тестирования, только еще более тонкий. Обычно применяется для функциональности, видимой для конечного пользователя, например корзины покупок. С помощью триггера (обычно заголовок HTTP, флаг, или имя/тип пользователя), часть трафика направляется на новые компоненты системы.
- Наблюдение и мониторинг кластера. Одно из важнейших преимуществ использования микросервисной сетки. Так как все сетевые потоки всех микросервисов находятся под управлением прокси-компонентов, можно отследить их взаимодействия в системе,

подсчитать задержки ответов, и сигнализировать при необычных отклонениях от ожидаемых, средних величин обслуживания (например, service level objective, SLO). Обычно результаты наблюдения выводятся на известные системы с открытым кодом Prometheus и Grafana.

Самые известные микросервисные сетки - Istio и Linkerd. Есть некоторый порог входа и обучения, установка и первоначальная настройка не так проста, требуется перенастройка всех контейнеров для работы с прокси-компонентами и установка немалого количества нестандартных сетевых компонентов, но потом вы получаете весомые преимущества. Некоторые провайдеры, например Google Kubernetes Engine, дают вам возможность сразу запускать кластер с работающей в нем сеткой Istio.

Аннотации

Как мы увидели в этой главе, метки играют важнейшую роль в управлении кластером Kubernetes, и поиске объектов в этой динамичной, сложной распределенной системе. Использовать метки стоит только со смыслом, для поиска, и для важнейших параметров ваших развертываний в Kubernetes.

В большой команде разработчиков, в больших системах из многих компонентов зачастую необходимо предоставить больше информации - кому принадлежит микросервис или часть системы, к кому обратится в случае проблемы, где находится исходный код или документация. В этом случае можно использовать аннотации (annotations) объектов Kubernetes. Это, по сути, те же самые метки, с теми же ограничениями на формат и размер, но не участвующие в поиске, и таким образом они могут содержать любую информацию, любого формата, и использовать одинаковые имена и значения, без опасений повлиять на работу селекторов (например, для сервисов).

В качестве простого примера мы можем добавить аннотацию `owner` с именем автора или администратора определенного компонента. Например, для нашего развертывания `time-service`:

```
...
# Тип объекта
kind: Deployment
# Метаданные нашего объекта, вложенный объект ObjectMeta
metadata:
   # список меток самого объекта Deployment
  labels:
    app: time-service
  # аннотации объекта
  annotations:
    owner: ivan.porty@ipsoftware.ru
  name: time-service
...
```

Аннотации часто используются в готовых решениях Kubernetes, таких как Google Kubernetes Engine или Amazon EKS, чтобы добавить к системным объектам дополнительную информацию.

Резюме

- Основная задача меток (labels) Kubernetes - разбиение объектов, таких как отсеки (pods), развертывания (deployments) и сервисы (services), на группы и категории, и легкий поиск по значениям меток. Метки - просто именованные произвольные строки. Особенно важны метки для работы сервисов. Именно по указанным меткам сервисы выбирают набор отсеков для направления к ним запросов.
- Несмотря на свою простоту, метки в совокупности с сервисами `Service` позволяют простым способом организовать сложную маршрутизацию трафика вашей системы - мы рассмотрели “канареечные” и “сине-голубые” развертывания.
- В некоторых случаях полезно менять набор меток вручную, командой `kubectl`, вместо декларативного объявления - например, для вывода части отсеков из развертывания для отладки.
- Хорошей практикой является использование упорядоченного набора меток (выбранного и оговоренного заранее вашей командой) для всех микросервисов, работающих в кластере под управлением Kubernetes. В дополнение к меткам, можно применять аннотации.
- Несмотря на свою мощь, метки ограничены работой на уровне объектов Kubernetes. Микросервисные сетки (service mesh) позволяют управлять трафиком и запросами более тонко, на уровне сетевых запросов и их параметров.

9. Непрерывное обновление в Kubernetes. Deployment

В предыдущей главе мы выяснили, что декларативная модель Kubernetes, и селекторы, работающие с обычными метками, настолько удачно взаимодействуют, что позволяют нам несколькими движениями руки и парой меток быстро и удобно реализовать такие мощные и непростые в других системах действия, как “канареечные” и “сине-голубые” развертывания.

Но Kubernetes не стал бы так популярен и не получил бы своего практически культового статуса, если бы в каждой операции не предоставлял еще более простые, а главное, декларативные, способы реализовать бесперебойную работу распределенной системы из множества компонентов и микросервисов. Объект для развертывания контейнеров `Deployment`, который мы до сих пор использовали просто для запуска контейнеров определенной версии и конфигурации, способен превратить нашу систему микросервисов в практически неуязвимую к простоям (*robust*), постоянно доступную (*HA, highly available*) и автоматически масштабируемую по требованию (*autoscaling*).

Непрерывное обновление (rolling update)

Давайте используем наш сервис по получению времени `time-service` еще раз, в этот раз для обновления его версии и функциональности без применения ручной работы с метками. У нас после работы в прошлых главах есть две версии - `0.1.0` и `0.2.0`. Перед запуском обновления, если у вас остались предыдущие варианты сервиса из прошлых глав, в том числе “канареечные” или “сине-зеленые”, нужно будет их удалить и запустить один, первый вариант сервиса, версии 0.1.0:

```
…
    # непосредственно описание контейнера в отсеке
    spec:
      containers:
      - image: ivanporty/time-service:0.1.0
        name: time-service
```

```
$ kubectl apply -f k8s/
deployment.apps/time-service created
service/time-service created
```

С таким применением развертывания мы уже хорошо знакомы - простое описание в YAML, и Kubernetes разворачивает для нас отсеки pods и запускает контейнеры в них, и поддерживает это желаемое состояние.

Но вот появляется новая, без сомнения лучшая, версия сервиса `0.2.0`, мы хотим перевести всю систему на новую версию, конечно же без остановки ее работы и перерыва в обслуживании драгоценных пользователей. Нет ничего проще - просто обновим версию сервиса, и передадим новое описание развертывания Deployment управляющей системе Kubernetes. Для наглядности скопируем описание YAML в папку `update`:

```
…
    # обновленная версия контейнера в отсеке
    spec:
      containers:
      - image: ivanporty/time-service:0.2.0
        name: time-service
```

```
$ kubectl apply -f k8s/update/
deployment.apps/time-service configured
```

Как мы видим, Kubernetes ответил нам, что развертывание `time-service` было сконфигурировано (configured), а не удалено и создано заново - обновление системы встроено в объекты `Deployment`, и останавливать и заново запускать отсеки и контейнеры не придется. Легко это проверить, как мы уже не однажды делали, с помощью обычного цикла `curl` к точке доступа сервиса прямо в терминале:

```
$ while true; do curl localhost:31890/time; sleep .5; done
{"time":"2019-11-06 15:09:14.6094974 +0000 UTC m=+116.323069101"}
{"time":"2019-11-06 15:09:15.1394157 +0000 UTC m=+116.852988101"}
{"time":"2019-11-06 15:09:15.6651447 +0000 UTC m=+117.378718301"}
{"time":"06 Nov 2019"}
{"time":"06 Nov 2019"}
```

Наш сервис, благодаря тому, что он по сути “нано”, а не “микро”, и написан на Go, запустился практически мгновенно, и как только управляющая система Kubernetes получила сигнал от отсека (pod), что процесс контейнера успешно запущен, она остановила предыдущую версию, а сервис `Service` остался неизменным - он по прежнему выбирает любые отсеки с метками `app: time-service`, и стал без всяких изменений со стороны пользователей сервиса передавать запросы к новой версии. Это и есть базовое непрерывное развертывание. Все что нам требуется - обновленная версия сервиса в новом образе контейнера с уникальной версией, и вставка этой версии в описание развертывания YAML!

История обновлений. Откат к стабильным версиям.

Обновления для развертывания `Deployment` не только делают это по возможности непрерывно, еще они, как и следует надежной системе, легко позволяют нам отследить историю изменений и обновлений, и в случае проблемы с новыми версиями (проблемы с запуском, несовместимость с другими микросервисами), откатить микросервис на предыдущую, стабильную версию.

Получить историю обновлений можно командой `rollout history`:

```
$ kubectl rollout history deploy time-service
deployment.extensions/time-service
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

Как и ожидалось, у нас две “редакции” (revision), или версии нашего развертывания. Пока мы еще хорошо знаем, что это обновление версий образа контейнера, и какие это были версии, но как мы видим, Kubernetes не знает, что именно было причиной (change cause). Хорошей практикой является использование аннотации `kubernetes.io/change-cause` - именно ее значение и показано в списке редакций обновления. Если мы добавим её в описание YAML развертывания с новой версией (в директории `update`):

```
# Тип объекта
…
kind: Deployment
# Метаданные нашего объекта, вложенный объект ObjectMeta
metadata:
   # список меток самого объекта Deployment
  labels:
    app: time-service
  # аннотации объекта
  annotations:
    owner: ivan.porty@ipsoftware.ru
    kubernetes.io/change-cause: Updated version to 0.2.0
  name: time-service
```

Теперь список редакций нашего развертывания станет выглядеть намного приличнее:

```
$ kubectl rollout history deploy time-service
deployment.extensions/time-service
REVISION  CHANGE-CAUSE
1         <none>
2         Updated version to 0.2.0
```

История обновлений хороша сама по себе, особенно в совокупности с декларативным управлением кластером, когда мы можем просто использовать систему контроля версий (Git) чтобы увидеть, когда, кто, и зачем обновлял и изменял систему и развертывание.

Но представим, что случилась ошибка. Как это часто бывает, один разработчик не понял другого, и третий программист решил, что сервис `time-service` выпущен в стабильной версии (`1.0.0`), и было бы просто глупо оставаться на старой версии. Обновление в Kubernetes - это быстро, просто и логично, файл с развертыванием был обновлен и передан в кластер:

```
…
  # аннотации объекта
  annotations:
    ...
    kubernetes.io/change-cause: Updated version to released 1.0!
…
    # обновленная версия контейнера в отсеке
    spec:
      containers:
      - image: ivanporty/time-service:1.0.0
        name: time-service
```

```
$ kubectl apply -f k8s/update/
deployment.apps/time-service configured
```

Если мы снова проверим работоспособность сервиса, то он продолжит работать - Kubernetes быстро поймет, что образа версии `1.0.0` просто нет, и не станет останавливать старый отсек с рабочей версией. Однако новая редакция развертывания будет теперь существовать в кластере, отчаянно пытаясь запустить новый отсек и контейнер из несуществующего образа (ошибка ImagePullBackOff), и достичь желаемого состояния:

```
$ kubectl get pods
NAME                            READY   STATUS             RESTARTS   AGE
time-service-749f577cbc-mng5m   0/1     ImagePullBackOff   0          3m55s
time-service-7ff577b7bd-kq7vb   1/1     Running            0          4m32s
```

Мы можем увидеть новую редакцию развертывания в истории изменений:

```
$ kubectl rollout history deploy time-service
deployment.extensions/time-service
REVISION  CHANGE-CAUSE
1         <none>
2         Updated version to 0.2.0
3         Updated version to released 1.0!
```

Ресурсы, особенно при использовании коммерческих провайдеров облака, дороги, и держать в кластере “сломанную” версию, несмотря на то что Kubernetes поддерживает общую работоспособность микросервиса, просто расточительно. Как поступить? Удалить нерабочий отсек pod? Развертывание `Deployment` снова и снова будет пытаться запустить его, так как мы передали это как желаемое состояние системы. Удаление развертывания будет совсем неудачной идеей - мы остановим непрерывную работу системы, пока будет запускаться новое развертывание.

Возможно, проблема с новой версией временная - например, нет доступа к репозиторию Docker Hub, и мы захотим вернуться к этой редакции позже. Развертывание `Deployment` способно помочь - мы можем вернуться к любой редакции, не удаляя никаких изменений! Давай посмотрим:

```
$ kubectl rollout undo deploy time-service
deployment.extensions/time-service rolled back

$ kubectl rollout history deploy time-service
deployment.extensions/time-service
REVISION  CHANGE-CAUSE
1         <none>
3         Updated version to released 1.0!
4         Updated version to 0.2.0

$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
time-service-7ff577b7bd-kq7vb   1/1     Running   0          21m
```

Команда `rollout undo` откатывает изменения в развертывании к предыдущей версии. Кстати, если это необходимо, можно указать точный номер редакции, к которой необходимо откатиться (флаг `--to-revision=...`).

После отката к предыдущей редакции “сломанная” редакция 3 никуда не исчезла! Если мы исправим проблему, то всегда можем к ней вернуться. Что еще интереснее, редакция 2 стала теперь новым номером 4. Текущая редакция развертывания всегда является последней, и ее номер всегда увеличивается по порядку. В этом есть смысл - вы всегда знаете, сколько редакций было у обновления, и какие из них были заново использованы при откатах.

Ну и наконец мы видим, что неработающий отсек pod исчез - предыдущей редакции развертывания он просто не нужен. Если же мы вернемся к редакции 3, он снова появится и попробует запустить контейнер с образом `time-service:1.0.0` - Kubernetes будет приводить кластер к желаемому состоянию (desired state).

Помощник развертывания - ReplicaSet

Kubernetes известен тем, что старается распределить обязанности по отдельным объектам, и сконцентрировать управление ими в одном месте - к примеру, мы управляем развертыванием с помощью объекта `Deployment`, однако непосредственно запуск контейнера и управление его ресурсами выполняет отсек `Pod`. Аналогично происходит с редакциями и масштабированием отсеков для развертываний - для каждой редакции развертывания создается свой объект `ReplicaSet` (набор экземпляров, или реплик, отсеков с контейнерами). В нашем случае с тремя редакциями развертывания мы получим три набора `ReplicaSet`:

```
$ kubectl get replicasets
NAME                      DESIRED   CURRENT   READY   AGE
time-service-5f59fbf479   0         0         0       42m
time-service-749f577cbc   0         0         0       41m
time-service-7ff577b7bd   1         1         1       41m
```

Легко увидеть, какой их этих наборов является активным и управляет активными отсеками (pod). Именно суффикс объекта `ReplicaSet` используется для активным отсеков, иногда это полезно знать, в случае большого количества экземпляров. В общем случае вручную управлять наборами отсеков `ReplicaSet` нет особенного смысла - развертывания `Deployment` намного мощнее и удобнее, однако знать о них стоит - на каждую редакцию вашего развертывания будет создаваться новый набор отсеков, всегда готовый включиться в случае отката версий.

В общем и целом наборы `ReplicaSet` недороги и не требуют много ресурсов, однако если у вас много развертываний и микросервисов, и вы нечасто используете откат к предыдущим версиями (что в общем и целом возможно и просто с использованием предыдущей версии описания YAML из системы контроля версий, если используется декларативное управление), то можно ограничить их количество (и количество возможных откатов). По умолчанию история развертываний ограничена 10 редакциями, давайте уменьшим их до 5:

```
...
# Описание собственно правил развертывания контейнера
# Вложенный объект DeploymentSpec
spec:
  # Количество запущенных отсеков pods для масштабирования
  replicas: 1
  # максимальное количество редакций (revisions)
  revisionHistoryLimit: 5
...
```

Стратегия непрерывного обновления

Все что мы использовали и только увидели, обновляя наше развертывание `time-service`, является результатом работы так называемой стратегии обновления (strategy), и по умолчанию используется непрерывное обновление (rolling update). Если мы укажем значение стратегии явно в нашем описании YAML, то получим следующее:

```
…
# Вложенный объект DeploymentSpec
spec:
  # Количество запущенных отсеков pods для масштабирования
  replicas: 1
  # максимальное количество редакций (revisions)
  revisionHistoryLimit: 5
  # стратегия обновления
  strategy:
    type: RollingUpdate
...
```

Стратегий наше развертывание поддерживает две:

- `Recreate` - развертывание заново. Удаляет все существующие отсеки (pods), и только потом создает новые. Как легко видеть, в случае проблем с новым развертыванием или контейнерами, микросервис сразу же перестает быть доступным. Эту стратегию конечно не стоит использовать в эксплуатации (production), но она может быть полезна, чтобы полностью стереть состояние предыдущих отсеков и контейнеров, например в кластере для разработки и отладки.
- `RollingUpdate` - знакомая нам стратегия непрерывного обновления по умолчанию. Запускает новые отсеки pods по одному, проверяет что они успешно запущены, и только после этого заканчивает работу отсеков с предыдущими версиями. Как мы уже видели, очень хороша для истории обновлений, легких откатов к прошлым версиям, и бесперебойной работы вашего сервиса.

Стратегия `RollingUpdate` имеет дополнительные настройки, крайне полезные в зависимости от типа микросервиса или приложения, и изменений в его логике и функциональности.

- `maxSurge` - насколько можно превысить желаемое (desired) количество отсеков pods. Если мы хотим, что наш сервис работал в 2 экземплярах, то значение
  - 1 - разрешает развертыванию создавать максимум три экземпляра, как правило два старой версии, один новой, потом два новой, один старой, и так далее.
  - 50% - делает все то же самое, но в случае автоматически масштабируемого развертывания учитывает, сколько отсеков работает на данный момент, вместо использования точного числа.
- `maxUnavailable` - дополняющая настройка к первой, имеющая такие же возможные абсолютные и процентные значения. Она указывает, насколько можно уменьшить количество желаемых отсеков, уменьшая доступность сервиса с целью экономии времени и ресурсов.

Проще всего все увидеть на примере с несколькими экземплярами микросервиса, например тремя. Давайте удалим все наши предыдущие развертывания (`kubectl delete -f time-service/k8s`), и попробуем следующее развертывание для первой версии `0.1.0`:

```
…
spec:
  # Количество запущенных отсеков pods для масштабирования
  replicas: 3
  # максимальное количество редакций (revisions)
  revisionHistoryLimit: 5
  # стратегия обновления
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 2
  selector:
    matchLabels:
      app: time-service
   # описание шаблона для создания новых отсеков
  template:
    metadata:
      # список меток для нового отсека
      labels:
        app: time-service
    # обновленная версия контейнера в отсеке
    spec:
```

```
      containers:
      - image: ivanporty/time-service:0.1.0
        name: time-service
```

Как видно, у нас будет три экземпляра нашего микросервиса, а непрерывное обновление может максимально превысить количество экземпляров на 1 (в общем 4), а уменьшить максимум на два (минимум 2 работающих отсека). Теперь передадим управляющему серверу Kubernetes такое же обновление, но с новой версией `0.2.0`, и посмотрим что произойдет с нашими отсеками (используем флаг `-w[atch]`, чтобы увидеть все изменения в непрерывном потоке по мере обновления отсеков):

```
$ kubectl apply -f k8s/update/rolling-0.1.0
…
$ kubectl apply -f k8s/update/rolling-0.2.0
...
$ kubectl get pods -w
 - отсеки первого развертывания
NAME                            READY   STATUS    RESTARTS   AGE
time-service-5f59fbf479-9v4vw   1/1     Running   0          25s
time-service-5f59fbf479-chxvp   1/1     Running   0          43s
time-service-5f59fbf479-sg995   1/1     Running   0          25s
- момент начала второго обновления - создан 1 новый отсек,
  и остановлена работы двух существующих отсеков.
time-service-7ff577b7bd-78zv4   0/1     Pending   0          0s
time-service-5f59fbf479-9v4vw   1/1     Terminating   0          6m47s
time-service-5f59fbf479-sg995   1/1     Terminating   0          6m47s
- создается еще два новых отсека, так как два остановлено
time-service-7ff577b7bd-6td9f   0/1     Pending       0          0s
time-service-7ff577b7bd-lg8db   0/1     Pending       0          0s
- запуск контейнеров и процесса в трех новых отсеках
   остается только один работающий отсек старой версии!
time-service-7ff577b7bd-78zv4   0/1     ContainerCreating   0          0s
time-service-7ff577b7bd-6td9f   0/1     ContainerCreating   0          0s
time-service-7ff577b7bd-lg8db   0/1     ContainerCreating   0          0s
- первый отсек с новой версией запущен
time-service-7ff577b7bd-6td9f   1/1     Running             0          2s
- в этот момент у нас 4 отсека, 1 старой версии, 1 новой
  и два в процессе запуска контейнеров. Так как новая версия
  запущена, остановлен последний отсек со старой версией
time-service-5f59fbf479-chxvp   1/1     Terminating         0          7m7s
- три отсека с новой версией - желаемое состояние!
time-service-7ff577b7bd-78zv4   1/1     Running             0          3s
time-service-7ff577b7bd-lg8db   1/1     Running             0          3s
```

Если мы посмотрим внимательно, то увидим, что непрерывное обновление развертывания никогда не превышает 4 экземпляров, указанных нами в качестве параметров для стратегии обновления. Однако, как мы видим, обновление очень быстро останавливает отсеки, как только отсеки с новой версией сообщают управляющему `kubelet` о том, что процесс запущен. В зависимости от того, как быстро запускается ваш микросервис, необходимо или сделать паузу перед остановкой старых отсеков, или создать так называемую проверку жизнеспособности контейнера (liveness probe), которая даст обновлению знать о том, что процесс в отсеке готов к работе.

Если нас больше заботит качество сервиса, а количество ресурсов достаточно велико, мы можем подстраховаться таким образом, что будет работать даже для автоматического масштабирования:

```
  # стратегия обновления
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 100%
      maxUnavailable: 0
```

В этом случае мы говорим, что готовы удвоить количество отсеков (`100%`), и не разрешаем уменьшать количество работающих отсеков (`maxUnavailable: 0`).

Автоматическое масштабирование

Как вы помните, еще в первой, обзорной главе про Kubernetes мы использовали автоматическое масштабирование нашего развертывания, впечатляющее оружие для автоматического управления кластером, способное сделать его действительно постоянно доступным и приспособленным к растущим нагрузкам, особенно если они случаются случайно, резко растут, а затем снижаются (например, просмотр видео через стриминг-сервис по вечерам, или выход новой версии популярной игры) - в таких случаях ручное масштабирование не настолько эффективно, и правильно рассчитать нагрузку заранее довольно сложно. Горизонтальное автоматическое масштабирование (horizontal pod autoscaling) считывает метрики ваших отсеков, их использование процессора и памяти, и при возрастании нагрузки увеличивает количество экземпляров сервиса под нагрузкой.

Автоматическое масштабирование не всегда доступно в любом кластере Kubernetes по умолчанию, хотя в коммерческих облаках, таких как Google Kubernetes Engine или Amazon EKS, оно почти всегда будет доступно. Если вы экспериментируете с Kubernetes на локальном кластере или запустили собственный кластер, нужно будет развернуть дополнительный сервис `metrics-server`, собирающий метрики кластера и предоставляющий их в едином виде для всех пользователей. Детали можно найти на сайте данного сервера в GitHub, ну а для локального кластера `minikube` метрики можно включить через расширение (addon) следующей командой:

```
$ minikube addons enable metrics
```

Проверить, что в кластере доступны метрики загрузки процессора и памяти, позволяет хорошо знакомая по Unix команда `top`, только теперь в составе `kubectl`. Она показывает текущий статус ресурсов для узлов кластера, например так:

```
$ kubectl top node
NAME       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube   301m         15%    1344Mi          69%
```

Ее же можно использовать и для просмотра загрузки отдельных отсеков (pods):

```
$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
time-service-68978c4db5-tn5nw       0m           5Mi
```

Теперь, зная что метрики загрузки кластера и работающих в отсеках приложений нам доступны, мы можем настроить горизонтальное автоматическое масштабирование. Как мы помним, в мире Kubernetes все представляет собой объект, и описание правил автоматического масштабирования - не исключение. Новый объект - это новый файл с неизвестным нам форматом YAML, но мы помним, что мы всегда можем прибегнуть к помощи команды `kubectl` с флагом `--dry-run`, чтобы получить заготовку описания нашего объекта в формате YAML или JSON. Второй вариант - использовать редактор со встроенной поддержкой схемы объектов Kubernetes, который подскажет вам правильные поля и их значения. Попробуем знакомый нам вызов:

```
$ kubectl autoscale deployment/time-service --min=1 --max=3 --cpu-percent=80 --dry-r\
un -o yaml
```

Вот что мы получим в результате - объект `HorizontalPodAutoscaler`, созданный как точный аналог результата вызванной нами команды, только на этот раз в предпочтительном декларативном варианте, который всегда будет проще отследить через систему контроля версий и использовать, чтобы попросить Kubernetes привести систему в желаемое состояние:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  creationTimestamp: null
  name: time-service
spec:
  maxReplicas: 3
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: time-service
  targetCPUUtilizationPercentage: 80
```

Описание нашего объекта содержит именно то, что мы и хотели получить вызовом команды `kubectl autoscale`. Можно задать нижний и верхний порог количества экземпляров, и описать, для каких ресурсов (`scaleTargetRef`), и как именно будет вычисляться необходимое количество экземпляров (`targetCPUUtilizationPercentage`) - в нашем случае мы указываем, что следим за использованием процессора для развертывания `time-service`, и если использование процессора одним из запущенных экземпляров превышает 80% от доступного максимума, необходимо добавить еще один экземпляр. Сервис Service для развертывания `time-service` незамедлительно включит новый экземпляр в свой список отсеков (после того, как будет запущен), и начнет отправлять ему запросы, снизив среднюю нагрузку. Обратное будет сделано при падении загрузки менее 80% - по прошествии некоторого времени ненужные экземпляры будут остановлены.

Остается только передать правила масштабирования управляющей системе Kubernetes:

```
$ kubectl apply -f k8s/autoscale/
horizontalpodautoscaler.autoscaling/time-service created
```

Аналогичным образом можно настроить масштабирование при превышении использования памяти. Как проверить, что автоматическое масштабирование работает? На мощном кластере, и таком миниатюрном примере, как наш `time-service`, превысить нагрузку непросто. Это будет хорошим упражнением - например, можно указать низкий порог нагрузки (10% загрузки процессора), или придумать тестовый метод, дающий высокую нагрузку - например, считать некую сложную математическую формулу.

Резюме

- Развертывания `Deployment` в Kubernetes обладают встроенной мощной поддержкой обновления версии ваших микросервисов без перерывов в обслуживании своих клиентов. В простейшем случае необходимо просто обновить версию или метку образа контейнера с микросервисом, и все будет сделано для нас автоматически.

- Развертывания поддерживают историю обновлений и легкий откат к предыдущим версиям в случае проблем с новой версией. За кулисами историю обновлений обеспечивают объекты `ReplicaSet`.
- Стратегия непрерывного обновления настраивается с помощью дополнительных параметров - мощное оружие для разнообразных типов обновлений, микросервисов и типов нагрузки.
- Масштабировать развертывания можно как вручную, просто указывая количество экземпляров работающего микросервиса или приложения, так и автоматически, указывая, при каких параметрах загрузки процессора и памяти необходимо добавить дополнительные экземпляры.

Завершение

Книга подошла к концу, а по большому счету мы только начали понимать концепцию Cloud Native и ее реализацию с помощью контейнеров и Kubernetes. Это и есть самое важное - понимая, что именно лежит в основе концепции, почему стали популярны микросервисы, что точно представляет собой контейнер, и какие принципы лежат в работе Kubernetes, можно уже с гораздо большей скоростью двигаться дальше, просто используя справочную информацию.

К примеру, мы вообще не затронули тему работу с данными - все наши простые примеры и обсуждения были основаны на микросервисах без состояния (stateless). Тем не менее любое приложение и система всегда обладает данными, которые нужно хранить. Дело в том, что управлять томами с данными, сетевыми дисками, масштабировать базы данных довольно тяжело, и в общем и целом это не является разработкой и программированием, а относится к инфраструктуре. Гораздо проще и надежнее в этом случае воспользоваться системой хранения данных, предлагаемых облаком. Это могут быть управляемые облаком классические базы данных (Amazon RDS, Google Cloud SQL), хранилища неструктурированных данных (Amazon S3), и много других решений, предлагаемых облаками. Это значительно упростит разработку и масштабирование вашей системы, это же рекомендуется 12 факторами (фактор 6).

Возвращаясь к 12 факторам облачного приложения, обсуждая их и работу с конфигурацией приложения, мы несколько раз упоминали, как Kubernetes упрощает конфигурацию с помощью секретов и карт конфигураций (config maps). Это важная часть, но работает она очень просто, и одного простого фрагмента YAML из любого подходящего примера на GitHub или сайта Kubernetes будет достаточно, чтобы начать ими пользоваться.

Мы также не обсудили работу с несколькими кластерами, контексты `kubectl` (context), и более эффективное использование ресурсов кластера с помощью разделенных пространств имен (namespaces). Можно снова сказать, что это важные части Kubernetes, однако они относятся больше к организации работы, а не к базовым концепциям, и их также будет проще найти в справочных страницах в Интернете.

Если вы еще помните начало книги, мы пообещали, что намеренно сделаем ее компактной и небольшой, и не станем перепечатывать документацию, примеры, существующие хорошие обучающие статьи, блоги, и все остальное. Если после прочтения этой книги вас по настоящему заинтересовал мир приложений Cloud Native, созданных для облака, как раз настало время идти дальше, воспользоваться местами очень хорошей онлайн-документацией, которая к тому же чуть не несколько раз в день обновляется и остается актуальной, найти подходящее вам по стилю обучающее видео, и продолжать свое путешествие вглубь страны Cloud Native.

Ресурсов и материалов огромное количество, не вся документация бывает высокого качества, но в целом два основных ресурса по Kubernetes и Docker предлагают отличную справочную информацию, весьма актуальную после того, как стали понятны основы концепции Cloud Native и контейнеров:

- `kubernetes.io` - центр информации, посвященный непосредственно Kubernetes. Не вся информация подается очевидно, некоторые статьи ссылаются друг на друга, но я надеюсь, после прочтения данной книги вам станет чуть легче ориентироваться в океане информации по Kubernetes.
- `docker.io` / `docs.docker.com` - основной сайт компании и технологии Docker и документация по технологии контейнеров и продуктам Docker. Надо сказать, что документация просто великолепная, навигация простая и очевидная, а раздел подробной справочной информации по командной строке `docker` и формату `Dockerfile` достойны того, чтобы быть постоянно открытыми в вашем браузере. Делает работу с контейнерами намного проще и эффективнее.
- `www.cncf.io` - главный сайт фонда Cloud Native Foundation. Это более обзорный сайт, но он дает неплохое понимание экосистемы, существующей вокруг концепции Cloud Native, например, посмотрите раздел проектов (projects). Можно многое узнать о способах работы с контейнерами, их безопасности, сборе журналов и метрик, и многом другом.
- Канал CNCF на YouTube содержит большую часть выступлений с конференций KubeCon, а также совместных конференций, посвященных различным смежных технологиям, таким как сервисные сетки service mesh. Некоторые видео и обучающие тренинги довольно удачно описывают многие концепции, если вам нравится данный формат.

В книге мы в основном, для простоты понимания общей картины, использовали самые базовые инструменты, такие как управление кластером с помощью команды `kubectl`, локальные кластеры `minikube` и `docker`, и только вскользь упоминали другие, расширенные инструменты. Чем больше вы станете программировать в стиле Cloud Native, чем очевиднее станут причины для использования сервисных сеток, шаблонов для конфигураций YAML, и много другого. Можно рекомендовать следующие инструменты, иногда значительно облегчающие работу:

- Skaffold, среда ускоренной постройки контейнеров с одновременным развертыванием в кластере Kubernetes. Создана с участием моей команды в Google, убирает постоянную необходимость заново помечать образы контейнеров новыми версиями и вызывать `kubectl`, вместо этого автоматически развертывая ваше приложение при любых изменениях в исходном коде. Если вы предпочитаете работать в редакторе IDE, посмотрите также на Google Cloud Code.
- Kubernetes Dashboard, стандартная визуальная панель мониторинга вашего кластера. Устанавливается в ваш кластер обычной командой `kubectl apply`, и значительно облегчает процесс знакомства с кластером и его ресурсами. Все детали на репозитории GitHub.

- Сервисные сетки (service mesh) Istio и Linkerd. Довольно высокий порог изначальной настройки и обучения, особенно Istio, но после установки и знакомства вы получаете огромные возможности по управлению трафиком и взаимодействию ваших микросервисов, и панели мониторинга за всеми вызовами, задержками, ошибками и всем остальным. Трудно оценить на маленькой "игрушечной" системе, лучше применять на реальной микросервисной архитектуре.

На этом наша краткая обзорная книга заканчивается. Надеюсь, что ваше дальнейшее знакомство с облачными приложениями будет продуктивным, программирование в стиле Cloud Native доставит массу удовольствия, а дизайн приложений в виде микросервисов станет намного проще. В результате должна получиться гибкая, мощная, устойчивая к отказам, всегда доступная система, легко масштабируемая под запросы миллионов пользователей по всему миру. Удачи!