ПРОГРАММИРОВАНИЕ CLOUD NATIVE

МИКРОСЕРВИСЫ, DOCKER И KUBERNETES

ИВАН ПОРТЯНКИН

Иван Портянкин

Программирование Cloud Native. Микросервисы, Docker и Kubernetes

«Издательские решения»

Портянкин И.

Программирование Cloud Native. Микросервисы, Docker и Kubernetes / И. Портянкин — «Издательские решения»,

ISBN 978-5-44-983387-7

В этой книге мы взглянем на концепцию Cloud Native:— создание приложений, «рожденных» для облака— 12 факторов облачных приложений и микросервисы— история и краткий обзор виртуализации и масштабирования — контейнеры Docker— настройка и оркестровка KubernetesПриложения Cloud Native помогут развернуть систему любой сложности в любом облаке и мгновенно приспособить ее к растущим нагрузкам.

ISBN 978-5-44-983387-7

Содержание

Программирование Cloud Native. Микросервисы, Docker и Kubernetes

Иван Портянкин

ISBN 978-5-4498-3387-7
Создано в интеллектуальной издательской системе Ridero

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 работать с миллионами пользователей и запросов 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 Kubernetes 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 облачные технологии, которые мы изучаем и пробуем в ней, являются одним из самых популярных и динамичных направлений программирования и разработки последних нескольких лет. Уровень изменений и их скорость очень высоки, и то, что было актуально и важно полгода назад, может уступить свое место новой технологии, процессу, 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, российскими Yandex и SberCloud). 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 для мгновенных экспериментов прямо из браузера рядом с документацией – 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, первый и самый важный шаг – понять суть происходящего, увидеть «лес за деревьями», узнать про краткую историю и эволюцию платформ, явлений, облаков, экосистем технологий, которые мы стараемся изучить. Нам, прежде всего, придется сначала понять, нужно ли нам вообще идти в направлении Cloud Native. 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 быстрым, интересным, компактным 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 на собственных серверах или виртуальных машинах AWS, работают с базами данных, разрабатывают стандартные приложения для операционных систем (с интерфейсами командной строки CLI, классические графические приложения desktop, 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 (monolith) 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 какова 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. 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 и ускорить процесс разработки самых сложных систем.

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 загвоздка. В книге будет некоторое количество намеренно очень простых примеров, главная цель которых состоит в том, чтобы за минимальное количество шагов и настроек 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 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 систему сервисов. Пользовательский интерфейс для любой системы это отдельная тема, чрезвычайно динамичная, вызывающая всегда горячие споры. Как и 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, которые мы затронем в этой книге. Тема чрезвычайно обширная и бесконечная, и изучения требует в отдельном разговоре. Будем считать что отличный пользовательский интерфейс у нас есть, и уделять внимания в данной книге мы ему не станем.

Примеры и их тестирование на GitHub

Все примеры и все команды, используемые нами для управления и экспериментов с Docker и Kubernetes, находятся в открытом репозитории ivanporty/cloud-docker-k8s на GitHub. Весь код примеров (включая команды, запускаемые нами из терминала) собирается и тестируется как часть непрерывной интеграции (CI) с помощью GitHub Actions – вы сможете увидеть их ежедневный запуск и результаты и сравнить со своими, если у вас что-то не получается. Ежедневная непрерывная сборка позволяет точно знать, что примеры книги работают и верны, и если новые версии Kubernetes или образов Docker что-то «ломают», я получаю уведомления об этом и быстро все обновляю, как в примерах, так и в электронных версиях книги.

Выбор 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 дороже», 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 (domain) 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 виртуальную машину GraalVM от Oracle, оптимизирующей запуск программы, и компилирующей байт-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 из 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), Microsoft Azure, российские #CloudMTS и SberCloud, или провайдеры меньших размеров, часто также предлагающих интересные цены и услуги (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 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 минимально 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 моем 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. Обновить печатную версию к сожалению не так просто, но если вы уже купили и прочитали печатную версию, вы найдете историю изменений книги на сайте и можете прочитать обновленные главы в электронном виде.

Открытый текст. Благодарности

Эта книга создана открытой (free open source) и ее текст доступен на GitHub для обновлений и изменений (репозиторий ivanporty/cloud-docker-k8s-book). Если вы увидели ошибку, неточность, или чувствуете, что какого-то материала не хватает, открывайте запрос (pull request), я его проверю, и ваши изменения тут же появятся в электронных версиях книги. Я благодарю несколько человек, которые внесли исправления и изменения в книгу (это их учетные записи на GitHub):

– AdamPirson
– lex111 (Alexey Pyltsyn)
– alg (Aleksey Gureiev)

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 Yandex, 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. Разработка 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, работающие 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), SberCloud, Yandex.Cloud.

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 это 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 «тяжелых» виртуальных машин, включающих в себя полную отдельную операционную систему. Контейнеры позволяют множеству микросервисов незаметно друг для друга работать на одном сервере, в пределах одной операционной системы.

– 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 контейнерами, внутри которых находятся микросервисы, в одном или множестве экземпляров. Основным инструментом 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* (monitoring) 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* (horizontal scaling или scaling out) – 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, мощные серверы обычно очень дороги.

Передача данных между микросервисами осуществляется по сети, по хорошо известным протоколам, поддерживаемым практически всеми известными языками и их библиотеками. Микросервисы больше не являются частью единого проекта и репозитория в системе контроля версий, 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 компонентам 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 эффективным и удобным. Это же позволяет проще настроить системы постоянного контроля качества и развертывания сделанных изменений на сервере (CI/CD, continuous integration and delivery), и сделать их работу прозрачной и высокопроизводительной, 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 операционной системы, и время ее запуска делает быстрое масштабирование и перезапуск компонентов практически невозможным. Как мы уже поняли, эту задачу берут на себя контейнеры с их легкой виртуализацией с помощью возможностей Linux (при необходимости можно запускать контейнеры и на других операционных системах). Время запуска контейнера практически неотличимо от 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 способом 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 разработчика для 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. Примеры – известные инструменты Terraform и Ansible. 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 или Yandex.Cloud, 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 (чаще всего это 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 меня-

ется. 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, обладающий достаточной мощью и 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 Yandex.Cloud, #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, Kind 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 о 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), такие как Istio и Linkerd. Смысл термина «service mesh» состоит в упорядочении и управлении сетевыми переговорами микросервисов, но слово «сеть» уже давно и прочно занято, и мы используем «сетку», чтобы уменьшить путаницу. Сервисные сетки решают многие проблемы, дают возможность наблюдать за сетевыми вызовами, строить графы, получать задержки, и многое другое. Системы сбора метрик, такие как Prometheus, позволяют интегрировать метрики вашей системы в единые центры наблюдения (к примеру, на основе Grafana). 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, которые 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 микросервисных 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 приложения (обычно с помощью Dockerfile), и он включает в себя все необходимое 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 нестандартного 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 с помощью постоянной сборки (CI, continuous integration) система помечается вер-

сией 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 версий в репозитории образов (к примеру Docker Hub), и можете строго воспроизвести любое состояние системы, не откатывая для этого 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, или с помощью системы обнаружения сервисов (service discovery). Данное требование идеально исполняется контейнерами, которые объявляют, по какому порту они будут ожидать соединений, и сервисами (service) Kubernetes, описывающими, как эти порты будут доступны в кластере. Kubernetes обеспечивает обнаружение сервисов с помощью простых имен DNS. 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 мгновенным, почти неотличимым 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 с неполноценными 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 – ElasticSearch + Logstash + Kibana, или Fluentd), работающие под управлением Kubernetes, собирают журналы с различных микросервисов, форматируют, 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.

Резюме

Завершая свой обзор мира Cloud Native, можно подчеркнуть, что в этой концепции сконцентрирован многолетний опыт технологических гигантов, прошедших путь от первых идей, воплощенных на нескольких слабых серверах, простейшими страницами на PHP и MySQL, к географически распределенным, работающим в виде десятков тысяч микросервисов системам, практически неуязвимым к простоям, пиковым нагрузкам, и трафику миллиардов пользователей со всего мира. Создаваемые вами системы могут быть более скромными, однако разрабатывая их «рожденными для облака», вы воспользуетесь опытом тысяч инженеров и успешных компаний, и разработаете приложения, удобные в эксплуатации и поддержке, надежные как Google и Yandex.

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, меньше ограничений на взаимодействие 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. Легко обмениваясь информацией (как правило, это текстовый формат HTTP/JSON, или двоичный формат gRPC), они предоставляют друг другу сфокусированные, маленькие услуги и сервисы, независимые от использованных для их реализации 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 использованию 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 повторяющегося 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 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 HTTP / 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, smart endpoint), даже если она повторяется в различных микросервисах. Сеть (и любой механизм передачи данных в целом), должна только передавать 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 заранее, 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 (полный список рекомендованных ресурсов вы найдете в конце этой главы).

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 неприятный 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 *сетками микросервисов* (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 Grafana уже встроены во многие библиотеки для создания микросервисов, и просто используя их, вы получите мощнейший центр наблюдения за 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 рассмотрели 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»

– Сэм Ньюмен (Sam Newman), «От монолита к микросервисам»

– 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 несомненный прорыв в попытках разработчиков и системных администраторов максимально использовать доступные им вычислительные ресурсы, при этом снизив сложность разработки и выпуска приложений.

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, Yandex 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 внушительный 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 появления 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.

Конец ознакомительного фрагмента.