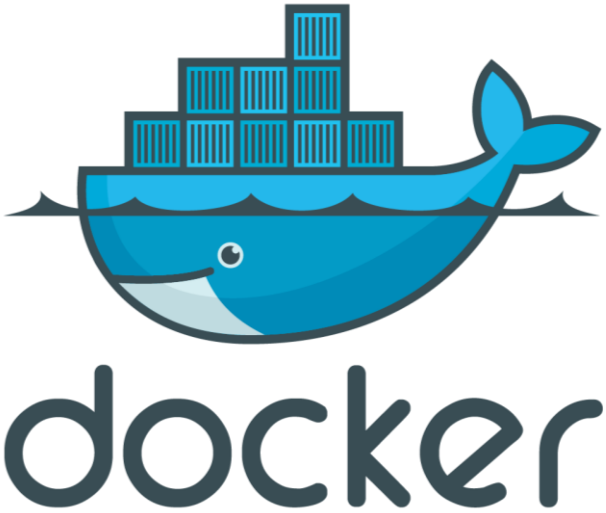


ОСНОВНЫЕ ТЕРМИНЫ DOCKER

Образ контейнера: пакет со всеми зависимостями и сведениями, необходимыми для создания контейнера. Образ включает в себя все зависимости (например, платформы), а также конфигурацию развертывания и выполнения для среды выполнения контейнера. Как правило, образ создается на основе нескольких базовых образов, наслоенных друг на друга в файловой системе контейнера. После создания образ остается неизменным.

Dockerfile: текстовый файл, содержащий инструкции по сборке образа Docker. Он похож на пакетный сценарий, где первая строка указывает базовый образ, с которого начинается работа, а следующие инструкции устанавливают необходимые программы, копируют файлы и т. п. для создания необходимой рабочей среды.

Сборка: действие по созданию образа контейнера на основе сведений и контекста, предоставленных файлом Dockerfile, а также дополнительных файлов в папке, где создается образ. Сборка образов выполняется с помощью следующей команды Docker:

```
BashКопировать
docker build
```

Контейнер: экземпляр образа Docker. Контейнер отвечает за выполнение одного приложения, процесса или службы. Он состоит из содержимого образа Docker, среды выполнения и стандартного набора инструкций. При масштабировании службы вы создаете несколько экземпляров контейнера из одного образа. Или пакетное задание может создать несколько контейнеров из одного образа, передавая разные параметры каждому экземпляру.

Тома: предложите файловую систему с возможностью записи, которую может использовать этот контейнер. Поскольку образы доступны только для чтения, а большинству программ требуется возможность записи в файловую систему, тома добавляют слой с поддержкой записи поверх образа контейнера, который программы смогут использовать как файловую систему с возможностью записи. Программа не знает, что она обращается к многоуровневой файловой системе. Для нее это обычный вызов файловой системы. Тома размещаются в системе узла под управлением Docker.

Тег: метка для образа, которая позволяет различать разные образы или версии одного образа (в зависимости от номера версии или целевой среды).

Многоэтапная сборка: этот компонент входит в Docker, начиная с версии 17.05, и позволяет сократить итоговый размер образов. Например, большой базовый образ, содержащий пакет SDK, можно использовать для компиляции и публикации, а затем можно использовать небольшой базовый образ среды выполнения для размещения приложения.

Репозиторий: коллекция связанных образов Docker, помеченная тегом, указывающим на версию образа. Некоторые репозитории содержат несколько вариантов одного образа, например образ с пакетом средств разработки (больший объем), образ только со средой выполнения (меньший объем) и т. д. Эти варианты можно пометить тегами. Один репозиторий может содержать варианты платформ, например образ Linux и образ Windows.

Реестр: служба, предоставляющая доступ к репозиториям. Реестр по умолчанию для большинства общедоступных образов — Центр Docker (принадлежащий Docker как организации). Реестр обычно содержит репозитории нескольких команд. Компании часто используют частные реестры для хранения своих образов и управления ими. Еще один пример — реестр контейнеров Azure.

Мультиархитектурный образ: Для нескольких архитектур это функция, которая упрощает выбор соответствующего образа в соответствии с платформой, в которой работает Docker. Например, когда Dockerfile запрашивает базовый образ **FROM mcr.microsoft.com/dotnet/sdk:6.0** из реестра, фактически возвращается **6.0-nanoserver-20H2**, **6.0-nanoserver-1809** или **6.0-bullseye-slim** (в зависимости от ОС и версии среды, в которой работает Docker).

Центр Docker: общедоступный реестр для загрузки образов и работы с ними. Центр Docker обеспечивает размещение образов Docker и интеграцию с GitHub и Bitbucket, предоставляет общедоступные или частные реестры, триггеры сборки и веб-перехватчики.

Реестр контейнеров Azure: общедоступный ресурс для работы с образами Docker и их компонентами в Azure. Он предоставляет реестр, близкий к вашим развертываниям в Azure, так что вы можете контролировать доступ и использовать группы и разрешения в Azure Active Directory.

Доверенный реестр Docker (DTR) : служба реестра Docker (из Docker), которую можно установить локально, чтобы она находилась в центре данных и сети организации. Его удобно использовать для частных образов, которыми необходимо управлять внутри предприятия. Доверенный реестр Docker входит в Центр данных Docker.

Docker Desktop: инструменты разработки в среде Windows и macOS для локальной сборки, выполнения и тестирования контейнеров. Docker Desktop для Windows предоставляет среды разработки для контейнеров Linux и Windows. Узел Linux Docker на Windows базируется на виртуальной машине Hyper-V. Узел для контейнеров Windows базируется непосредственно на Windows. Docker Desktop для Mac основан на платформе Apple Hypervisor и гипервизоре xhyve, который предоставляет виртуальную машину узла Docker для Linux в macOS. Docker Desktop для Windows и Mac заменяет решение Docker Toolbox, которое было основано на Oracle VirtualBox.

Compose: программа командной строки и формат файлов YAML с метаданными для определения и выполнения многоконтейнерных приложений. Вы определяете одно приложение на основе нескольких изображений с одним или несколькими *YML-файлами* , которые могут переопределять значения в зависимости от среды. Создав определения, вы можете развернуть все многоконтейнерное приложение с помощью одной команды (docker-compose up), которая создает один контейнер на образ на узле Docker.

Кластер: коллекция узлов Docker, представленная в виде единого виртуального узла Docker, чтобы можно было масштабировать приложение в нескольких экземплярах служб, распределенных по нескольким узлам кластера. Кластеры Docker можно создавать с помощью Kubernetes, Azure Service Fabric, Docker Swarm и (или) Mesosphere DC/OS.

Оркестратор: инструмент, упрощающий управление кластерами и узлами Docker. С помощью оркестраторов можно управлять образами, контейнерами и узлами через интерфейс командной строки (CLI) или графический интерфейс. Вы можете управлять соединениями контейнеров, конфигурациями, балансировкой нагрузки, обнаружением служб, высоким уровнем доступности, конфигурацией узлов Docker и многим другим. Оркестратор используется для выполнения, распределения, масштабирования и восстановления рабочих нагрузок в коллекции узлов. В качестве оркестраторов

обычно используются те же продукты, которые обеспечивают кластерную инфраструктуру, например Kubernetes и Azure Service Fabric (на рынке доступны и другие предложения).

При использовании Docker вы создаете приложение или службу и упаковываете их и их зависимости в образ контейнера. Образ — это статическое представление приложения или службы, а также их конфигурации и зависимостей.

Для запуска приложения или службы создается экземпляр образа приложения, чтобы создать контейнер, который будет запущен на узле Docker. Контейнеры изначально проверяются в среде разработки или на ПК.

Вам следует хранить образы в реестре, который выступает в качестве библиотеки образов. Он необходим при развертывании на оркестраторы в рабочей среде. Docker поддерживает общедоступный реестр с помощью Docker Hub. Другие поставщики предлагают реестры для различных коллекций образов, включая Реестр контейнеров Azure. Кроме того, организации могут развернуть локальные частные реестры для своих образов Docker.

Реестр напоминает книжную полку. В нем хранятся образы, которые можно извлекать для создания контейнеров, необходимых для запуска служб или веб-приложений. Частные реестры Docker могут храниться в локальной среде или в общедоступном облаке. **Docker Hub** — это общедоступный реестр, поддерживаемый Docker.

Размещение образов в реестре позволяет хранить статические и неизменяемые фрагменты приложений, включая все их зависимости на уровне платформы. Затем эти образы можно добавить в систему управления версиями и развернуть в нескольких средах, создав, таким образом, согласованную единицу развертывания.

Частные реестры образов, размещенные локально или в облаке, рекомендуется использовать, если:

- Ваши образы не могут быть открыты для общего доступа по соображениям конфиденциальности.
- Вам необходимо обеспечить минимальную сетевую задержку между образами и выбранной средой развертывания. Например, если рабочей средой является Azure, вы, вероятно, захотите разместить образы в Реестре контейнеров Azure, чтобы сетевая задержка была минимальной. Точно так же, если рабочая среда является локальной, вы можете развернуть локальный доверенный реестр Docker в той же локальной сети.

На рисунке 1 показано, как **образы и реестры** в Docker связаны с другими компонентами. На нем также показано несколько вариантов реестра от поставщиков.

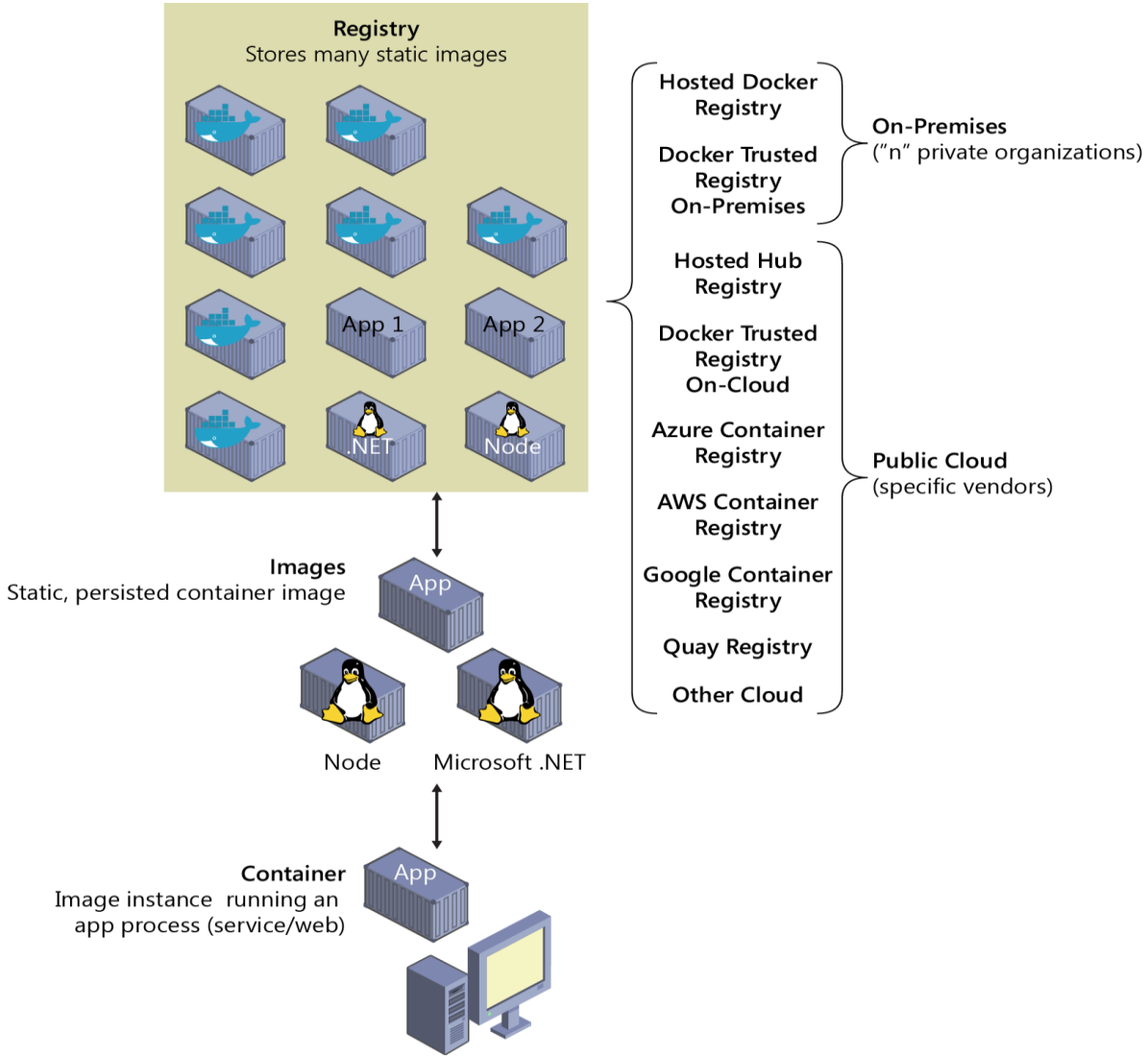


Рисунок 1 – Классификация основных понятий и терминов Docker

ПУТЬ К СОВРЕМЕННЫМ ПРИЛОЖЕНИЯМ НА ОСНОВЕ КОНТЕЙНЕРОВ

Введение в жизненный цикл приложения Docker

Разработчики предпочитают реализовывать контейнеры и Docker, так как они упрощают проведение развертываний и выполнение ИТ-операций, благодаря чему повышается уровень гибкости, продуктивности и оперативности всех задействованных сторон.

Природа контейнеров и технологии Docker позволяют разработчикам легко обеспечивать совместную работу со своим программным обеспечением и зависимыми компонентами в рабочей и операционной ИТ-среде. Благодаря этому могут своевременно выявляться потенциальные проблемы, связанные с неработоспособностью кода на других компьютерах.

Контейнеры используются для разрешения конфликтов приложений между различными средами. Косвенно контейнеры и технология Docker позволяют сократить дистанцию между разработчиками и ИТ-операциями, обеспечивая их эффективную совместную работу. Внедряя контейнерный рабочий процесс, многие клиенты получают присущую DevOps непрерывность, для обеспечения которой ранее приходилось использовать более сложные конфигурации конвейеров выпуска и сборки. Применение контейнеров позволяет упростить конвейеры сборки, тестирования и развертывания в DevOps (рисунок 2).

При использовании контейнеров Docker разработчики владеют их содержимым (приложение и служба, а также зависимости для платформ и компонентов) и механизмами совместной работы контейнеров и служб в качестве приложения, состоящего из коллекции служб.

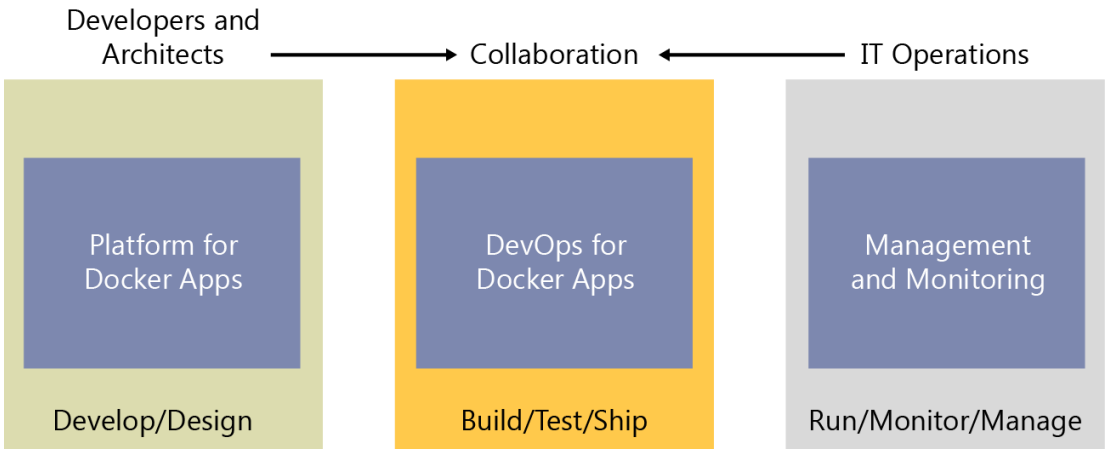


Рисунок 2 – Основные рабочие нагрузки по типам участников жизненного цикла для контейнерных приложений Docker

Взаимозависимости между несколькими контейнерами определяются в файле `docker-compose.yml` или в *манифесте развертывания* соответствующего вида. В то же время представители группы ИТ-операций (ИТ-специалисты и руководители) могут сосредоточить усилия на управлении рабочими средами, инфраструктурой, масштабируемостью, мониторингом и, в конечном итоге, надлежащей доставкой приложений конечным пользователям, не заботясь при этом о том, что содержится в различных контейнерах.

Таким образом, можно провести аналогию между нашим "контейнером" и настоящим контейнером для грузоперевозок. Соответственно, владельцу содержимого контейнера не нужно беспокоиться о том, каким образом будет осуществляться доставка контейнера, а транспортной компании необходимо перевезти его из одной точки в другую и абсолютно не требуется знать, что находится внутри него. Аналогичным образом, разработчики могут создавать собственное содержимое внутри контейнера Docker, не беспокоясь при этом о механизмах его "транспортировки".

На принципиальной схеме, показанной на рисунке 2, в левой части разработчики создают и запускают код локально в контейнерах Docker с помощью Docker для Windows или Mac. Они определяют операционную среду для кода с помощью файла Dockerfile, в котором указывается базовая операционная система для выполнения, а также

шаги сборки, необходимые для встраивания кода в образ Docker. Принципы взаимодействия между несколькими образами определяются разработчиком в упомянутом выше манифесте развертывания файла `docker-compose.yml`. После завершения локальной разработки код приложения и файлы конфигурации Docker отправляются в используемый репозиторий кода (в данном случае в репозиторий Git).

Принципы DevOps определяют контейнеры сборки и непрерывной интеграции с использованием файла Dockerfile, представленного в репозитории кода. Система непрерывной интеграции извлекает базовые образы контейнера из выбранного реестра Docker и выполняет построение пользовательских образов Docker для приложения. После этого образы проверяются и отправляются в реестр Docker, используемый для развертываний в нескольких средах.

Показанная в правой части принципиальной схемы операционная команда осуществляет управление развернутыми приложениями и инфраструктурой в рабочей среде, а также обеспечивает мониторинг среды и приложений, направляя команде разработчиков отзывы и аналитические данные, которые могут использоваться для улучшения приложения. Приложения-контейнеры обычно выполняются в рабочей среде с помощью оркестрации контейнеров, таких как Kubernetes, где для настройки единиц развертывания вместо файлов DOCKER обычно используются диаграммы Helm.

Совместная работа двух команд обеспечивается посредством базовой платформы (контейнеры Docker), которая реализует разделение сфер ответственности на контрактной основе, что позволяет значительно повысить эффективность взаимодействия между командами в рамках жизненного цикла приложения. Разработчики являются владельцами содержимого контейнера, а также его операционной системы и взаимозависимостей. Операционная команда обеспечивает выполнение образов в системе оркестрации с использованием манифеста.

Трудности, связанные с использованием технологии Docker в рамках жизненного цикла приложений.

В ближайшие годы число контейнерных приложений будет возрастать по многим причинам, одной из которых является создание приложений на основе микрослужб.

На протяжении последних 15 лет в качестве основы для тысяч приложений использовались веб-службы, и в ближайшие годы аналогичная картина может сложиться с приложениями на базе микрослужб, выполняющимися в контейнерах Docker.

Также следует отметить, что контейнеры Docker можно использовать и для монолитных приложений, сохраняя при этом большинство преимуществ этой технологии. Однако контейнеры ориентированы не только на работу с микрослужбами.

В результате использования контейнеров Docker и микрослужб в процессе разработки организации возникают новые трудности, в связи с чем требуется продуманная стратегия, направленная на обеспечение эффективной работы множества контейнеров и микрослужб в рабочих системах. В конечном счете, в рабочей среде в корпоративных приложениях может выполняться до нескольких сотен и даже тысяч контейнеров или экземпляров.

В связи с этими трудностями меняются требования к средствам DevOps, из-за чего вам будет необходимо определить новые процессы в рамках операций DevOps и дать ответы на вопросы следующего вида:

- Какие средства разработки можно использовать для решения задач по непрерывной интеграции, непрерывному развертыванию, управлению и операционной деятельности?
- Как компания будет обрабатывать ошибки в контейнерах при выполнении в рабочей среде?
- Как можно заменить программные компоненты в рабочей среде с минимальным временем простоя?

- Как осуществлять масштабирование и мониторинг рабочей системы?
- Как включить процессы тестирования и развертывания контейнеров в конвейер выпуска?
- Как использовать средства и платформы с открытым исходным кодом для работы с контейнерами в Microsoft Azure?

Общие сведения об универсальном комплексном рабочем процессе, связанном с жизненным циклом приложений Docker

На рисунке 3 показан рабочий процесс для жизненного цикла приложения Docker, основное внимание в котором в данном случае акцентировано на конкретных операциях и ресурсах DevOps.

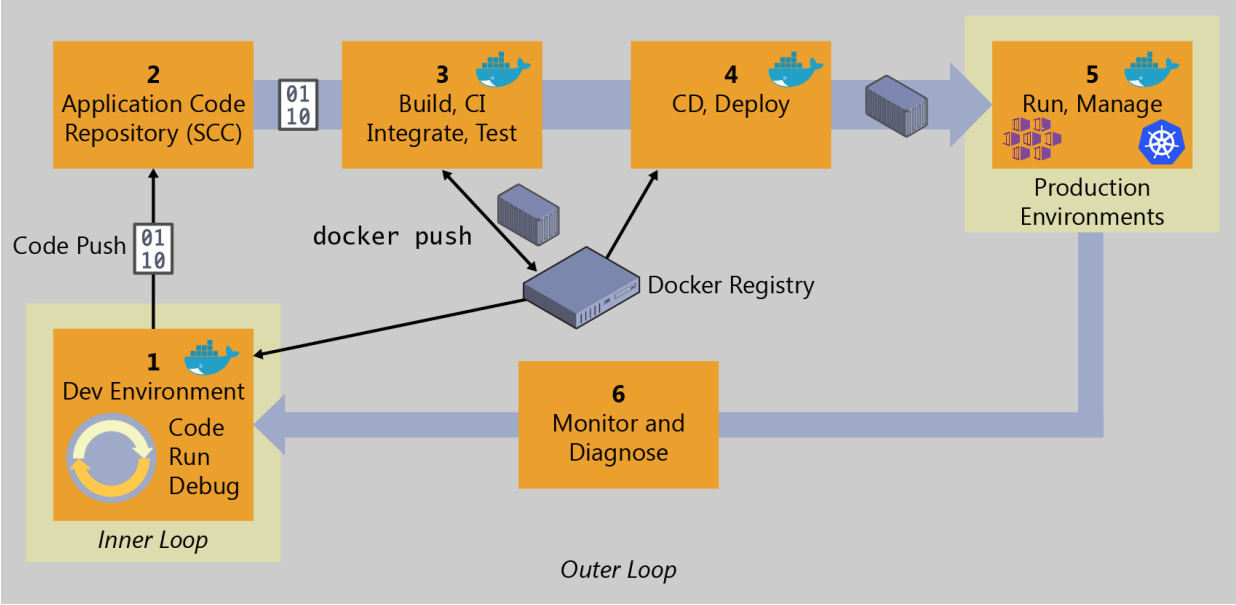


Рисунок 3 – Высокоуровневое представление рабочего процесса для жизненного цикла контейнерного приложения Docker

Все начинается с написания кода разработчиком в рамках рабочего процесса "внутреннего цикла". На этом этапе разработчики определяют все выполняемые в коде действия, после чего отправляют его в репозиторий кода (например, в такую систему управления версиями, как Git). После фиксации кода репозиторий активирует конвейер непрерывной интеграции и остальные этапы рабочего процесса.

В рамках внутреннего цикла осуществляются написание, выполнение, тестирование и отладка кода, а также любые другие действия, которые должны быть выполнены перед локальным запуском приложения. Также на этом этапе разработчик запускает и тестирует приложение в качестве контейнера Docker. Рабочий процесс внутреннего цикла будет описываться в последующих разделах.

Если рассматривать комплексный рабочий процесс DevOps в целом, он представляет собой больше, чем просто технологию или набор средств: это отдельный образ мышления, требующий пересмотра культурных ценностей. Он включает в себя людей, процессы и соответствующие средства, позволяющие ускорить жизненный цикл приложения и сделать его более предсказуемым. Предприятия, внедряющие контейнерный рабочий процесс, как правило, реструктурируют свою организацию с учетом требований, предъявляемых к людям и процессам.

Применение DevOps позволяет ускорить процессы взаимодействия между разными командами в условиях конкурентной борьбы за счет замены подверженных ошибкам ручных процессов автоматизированными технологиями, благодаря чему повышается эффективность отслеживания и самих рабочих процессов. Кроме того, благодаря сочетанию локальных и облачных ресурсов с тесно интегрированными наборами средств организации могут повысить эффективность управления средами и добиться заметной экономии средств.

Реализация рабочего процесса DevOps для приложений Docker позволит вам использовать преимущества технологий Docker практически на любом этапе рабочего процесса, начиная с разработки в рамках внутреннего цикла (написание, выполнение и отладка кода) и этапов сборки, тестирования и непрерывной интеграции, заканчивая развертыванием контейнеров в промежуточных и рабочих средах.

Совершенствование принципов обеспечения качества позволяет выявлять ошибки на ранних стадиях разработки, благодаря чему

снижаются затраты на их устранение. Включая среду и зависимые компоненты в образ и следуя принципам развертывания одного образа в нескольких средах, вы снижаете степень зависимости от конфигурации среды, что позволяет повысить надежность развертывания.

Эффективные инструменты мониторинга и диагностики позволяют получать всеобъемлющие аналитические данные по проблемам с производительностью и поведению пользователей, которые могут выступать в качестве основы для определения приоритетов и направлений для инвестиций в будущем.

DevOps следует воспринимать не как конечную цель, а как путь к этой цели. Внедрение этого подхода должно осуществляться постепенно в рамках проектов соответствующей направленности, в ходе которых вы сможете добиваться успеха, учиться и совершенствоваться.

Преимущества DevOps для контейнерных приложений

Ниже наиболее важные преимущества, которые дает эффективная реализация рабочего процесса DevOps:

- Повышение качества программного обеспечения, скорости его разработки и соответствия требованиям.
- Более раннее и экономически эффективное применение мер по непрерывному улучшению и корректировке.
- Повышение степени прозрачности и эффективности совместной работы между всеми участниками процесса доставки и эксплуатации программного обеспечения.
- Более эффективное управление затратами и подготовленными к работе ресурсами при одновременном снижении рисков, связанных с безопасностью.
- Изначальная поддержка множества существующих решений DevOps, в том числе и с открытым исходным кодом.

Общие сведения о платформе и средствах Майкрософт для контейнерных приложений

На рисунке 4 представлены основные части жизненного цикла приложений Docker, классифицированные по видам деятельности нескольких команд (разработка приложений, процессы инфраструктуры DevOps, управление ИТ-средой и выполнение ИТ-операций). Как правило, в организации специалисты, ответственные за каждое направление, имеют разные профили. То же самое относится и к навыкам.

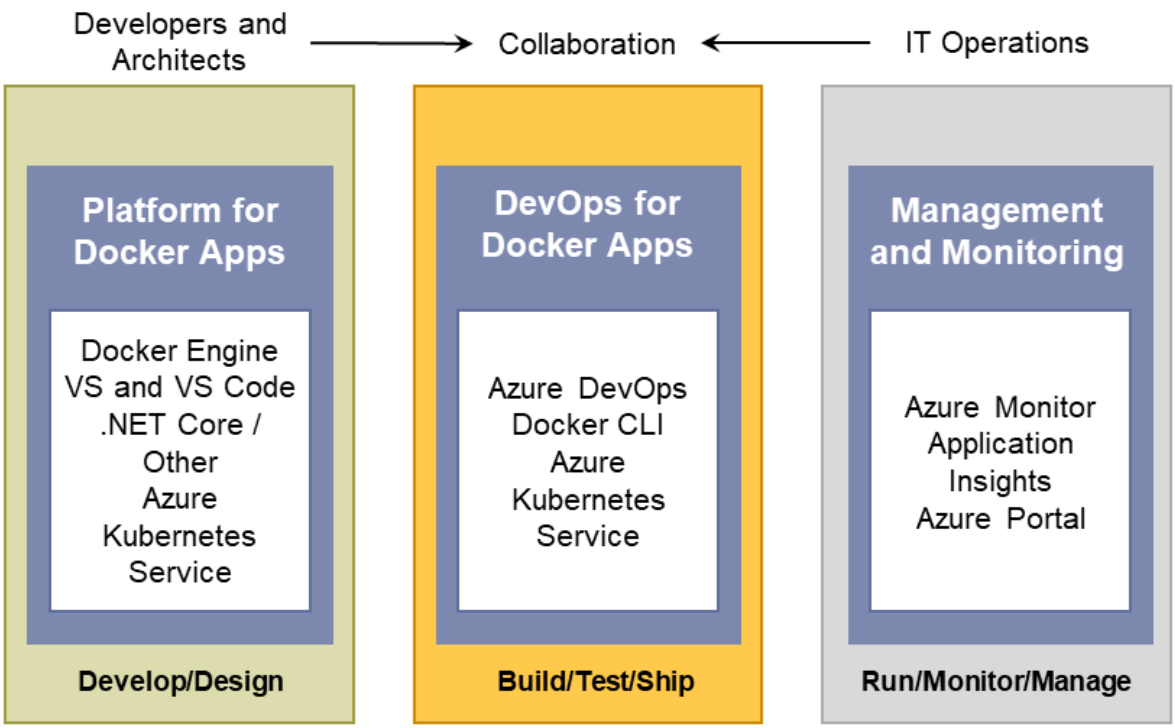


Рисунок 4 – Основные части жизненного цикла контейнерных приложений Docker на базе платформы и средств Майкрософт

Рабочий процесс жизненного цикла контейнерного объекта Docker может изначально носить предписывающий характер на основе выбора продуктов по умолчанию, что позволяет разработчикам быстрее приступать к работе. Однако его фундаментальность состоит в необходимости существования открытой платформы, за счет чего он получит гибкость для адаптации к различным контекстам каждой организации или предприятия.

В таблице 1 показано, что цель Azure DevOps для контейнерных приложений Docker заключается в предоставлении открытого рабочего процесса DevOps с возможностью выбора продуктов (Майкрософт или сторонних производителей) для использования на каждом этапе и одновременном функционировании упрощенного рабочего процесса с уже подключенными продуктами по умолчанию. Таким образом, можно быстро приступить к рабочему процессу DevOps корпоративного уровня по созданию приложений Docker.

Таблица 1 – Рабочие процессы DevOps, открытые для любой технологии

Узел	**Технологии Майкрософт**	**Сторонние (подключаемые к Azure)**
Платформа для приложений Docker	• Microsoft Visual Studio и Visual Studio Code • .NET • Служба Microsoft Azure Kubernetes (AKS) • Реестр контейнеров Azure	• Любой редактор кода • Любой язык (Node.js, Java, Go) • Любой оркестратор и планировщик • Любой реестр Docker
DevOps для приложений Docker	• Azure DevOps Services • Microsoft Team Foundation Server • GitHub • Служба Azure Kubernetes (AKS)	• GitHub, Git, Subversion и т. д. • Jenkins, Chef, Puppet, Velocity, CircleCI, TravisCI и т. д. • Локальный Docker Datacenter, Kubernetes, ОС/ЦОД Mesos и т. д.
Управление и мониторинг	• Azure Monitor	• Marathon, Chronos и т. д.

Платформа Майкрософт и средства для контейнерных приложений Docker, как определено в таблице 1, включают следующие компоненты.

- **Платформа для разработки приложений Docker** Разработка службы или коллекции служб, составляющих "приложение". Платформа разработки предоставляет все рабочие разработчики, необходимые перед отправкой кода в общий репозиторий кода. Разработка служб, развертываемых в качестве контейнеров, похожа на разработку приложений или служб без использования Docker. Можно продолжать использовать предпочитаемый язык (.NET, Node.js, Go и т. д.) и редактор или интегрированную среду разработки, например Visual Studio или Visual Studio Code. Однако вы не считаете Docker конечной средой развертывания, вы разрабатываете в ней свои службы. Создание, запуск,

тестирование и отладка кода выполняются локально в контейнерах, которые представляют собой целевую среду во время разработки. За счет локального предоставления этой целевой среды контейнеры Docker помогают в значительной степени улучшить жизненный цикл DevOps. Visual Studio и Visual Studio Code имеют расширения для интеграции контейнеров Docker в процессе разработки.

- **DevOps для приложений Docker**. Разработчики, создающие приложения Docker, могут использовать Azure DevOps, GitHub или любые другие сторонние продукты, например, Jenkins, чтобы организовать полнофункциональное автоматизированное управление жизненным циклом приложений (ALM).

  Благодаря Azure DevOps и (или) GitHub разработчики могут использовать DevOps с ориентацией на контейнеры для создания быстрого итеративного процесса, обеспечивающего управление исходным кодом из любого места (Azure DevOps-Git, GitHub, любой удаленный репозиторий Git или Subversion), непрерывную интеграцию (CI), внутренние модульные тесты, внутренние интеграционные тесты контейнера или службы, непрерывную поставку (CD) и управление выпусками (RM). Разработчики также могут автоматизировать выпуски приложений Docker в службу Azure Kubernetes (AKS), как в средах разработки, так и в средах промежуточного хранения и производства.

- **Управление и мониторинг**. ИТ-отдел может управлять рабочими приложениями и службами и отслеживать их несколькими способами, объединяя оба направления.
  - **портал Azure Служба Azure Kubernetes** (AKS) помогает настраивать и обслуживать среды Docker. Вы также можете использовать другие оркестраторы для визуализации и настройки кластера.
  - **Средства Docker** Вы можете управлять приложениями-контейнерами с помощью знакомых средств. Чтобы переместить рабочие нагрузки для контейнеров в облако, не требуется изменять существующие методики управления Docker.

Работайте с уже знакомыми средствами управления и подключайтесь к нужному вам оркестратору через стандартные конечные точки API. Для управления приложениями Docker можно также использовать другие средства сторонних разработчиков или даже инструменты CLI Docker.

- **Средства с открытым кодом** Так как AKS предоставляет стандартные конечные точки API для подсистемы оркестрации, самые популярные средства совместимы с AKS и в большинстве случаев будут работать вне поля, включая визуализаторы, мониторинг, средства командной строки и даже будущие инструменты по мере их доступности.
- **Расширенная безопасность GitHub**Расширенная безопасность GitHub предлагает набор инструментов для защиты цепочки поставок программного обеспечения, которые позволяют легко интегрировать безопасность в повседневный рабочий процесс команд, занимающихся разработкой контейнерных приложений.
- **Azure Monitor**. Это решение Azure для отслеживания всех аспектов рабочей среды. Рабочие приложения Docker можно отслеживать, просто настроив пакет SDK в службах для получения созданных системой данных журнала из приложения.

Проектирование приложений Docker

Равноценность контейнера процессу

Контейнер в модели на основе контейнеров представляет отдельный процесс. Определив контейнер как границу процесса, вы можете создавать примитивы, позволяющие масштабировать процессы или помещать их в пакет. Когда вы запускаете контейнер Docker, отображается определение ENTRYPOINT. Оно определяет процесс и время существования контейнера. По завершении процесса время существования контейнера истекает. Существуют как длительные процессы, например веб-серверы, так и кратковременные, например пакетные задания, которые могли реализовываться как веб-задания Microsoft Azure. В случае сбоя процесса работа контейнера завершается и оркестратор принимает управление. Если оркестратору указано поддерживать пять выполняющихся экземпляров, то в случае сбоя одного из них оркестратор создаст ему на замену еще один контейнер. В пакетном задании процесс запускается с параметрами. По завершении выполнения процесса работа считается выполненной.

Возможны ситуации, когда в одном контейнере желательно выполнять несколько процессов. В любом документе архитектуры никогда не существует "никогда", ни всегда "всегда". Для сценариев, требующих нескольких процессов, общим шаблоном является использование супервизора.

Монолитные приложения

В этом сценарии вы можете создать отдельное монолитное веб-приложение или службу и развернуть их как контейнер. Это приложение может не иметь монолитную внутреннюю структуру и состоять из нескольких библиотек, компонентов или даже уровней (прикладной уровень, уровень домена, уровень доступа к данным и т. д.). Внешне оно будет представлять собой единый контейнер — единый процесс, единое веб-приложение или единую службу.

Для управления этой моделью вы развертываете один контейнер, представляющий собой приложение. Для масштабирования просто добавьте дополнительные копии, расположив перед ними подсистему балансировки нагрузки. Управлять одним развертыванием в одном контейнере или виртуальной машине гораздо проще.

Такой монолитный шаблон может конфликтовать с принципом контейнера: "контейнер выполняет одну задачу и в одном процессе". Вы можете включить в один контейнер несколько компонентов, библиотек или внутренних слоев, как показано на рисунке 6.

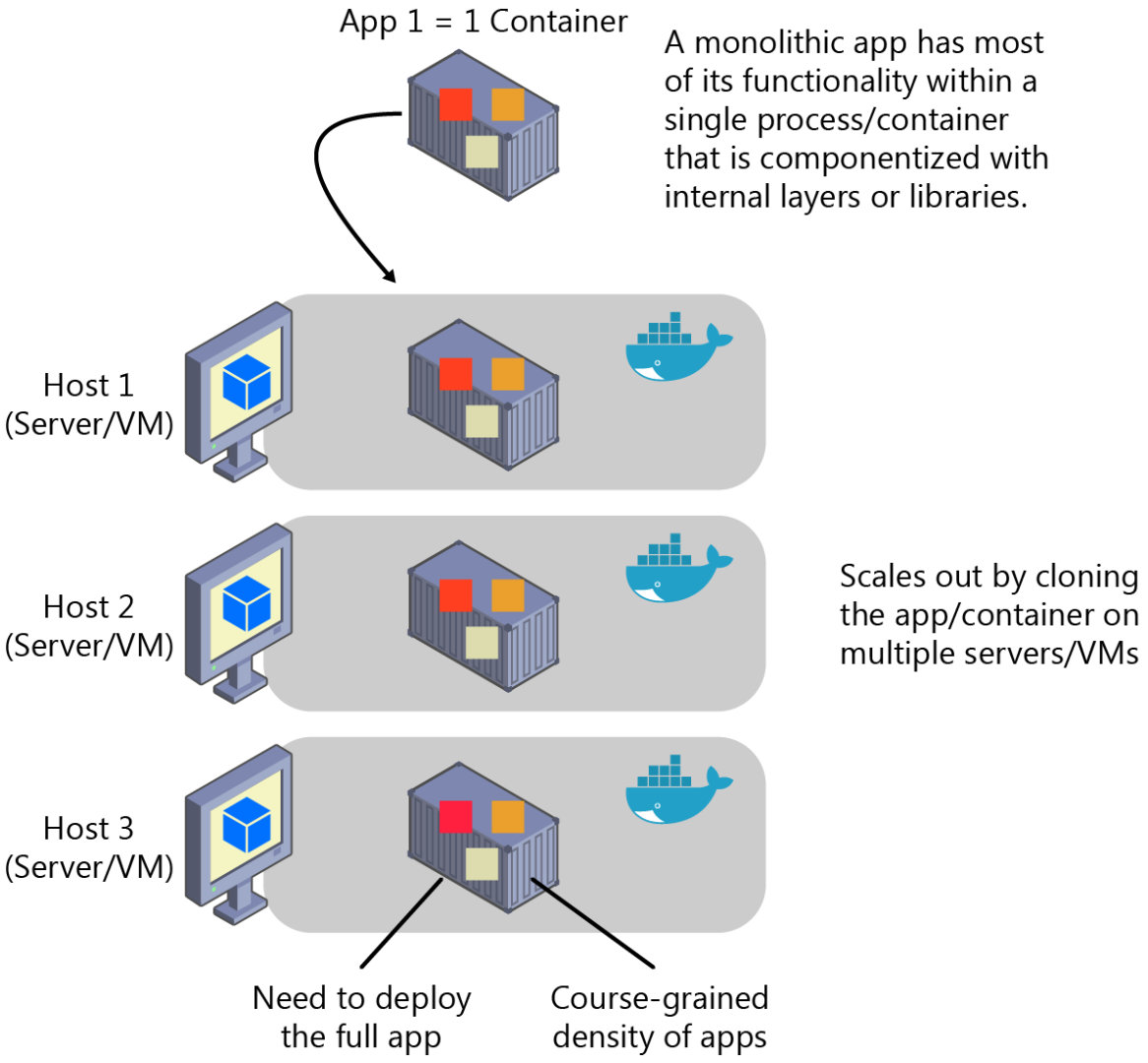


Рисунок 6 – Пример архитектуры монолитного приложения

Все функции монолитного приложения или основная их часть сосредоточены в одном процессе или контейнере, который разбивается на внутренние слои или библиотеки. Недостаток этого подхода становится очевидным, когда приложение разрастается и его необходимо масштабировать. Если масштабируется приложение целиком, все получится. Но в большинстве случаев необходимо масштабировать лишь некоторые части приложения, пока другие компоненты работают нормально.

В примере приложения для электронной торговли, вероятнее всего, потребуется масштабирование компонента со сведениями о товарах. Клиенты чаще просматривают товары, чем приобретают их. Клиенты чаще складывают товары в корзину, чем оплачивают их. Не так много клиентов пишут комментарии или просматривают историю покупок. И у вас, скорее всего, может быть лишь несколько сотрудников в одном регионе, которые управляют содержимым и маркетинговыми кампаниями. При масштабировании монолитных решений весь код развертывается многократно.

Помимо того, что необходимо масштабировать все компоненты, изменения в одном компоненте требуют полного повторного тестирования всего приложения и полного повторного развертывания всех его экземпляров.

Монолитный подход нашел широкое распространение и используется многими организациями при разработке архитектуры. Во многих случаях это позволяет добиться желаемых результатов, но иногда организация сталкивается с ограничениями. Во многих организациях приложения строились по такой модели, так как несколько лет назад с помощью существующих инструментов и инфраструктуры слишком сложно было создавать архитектуры SOA, и проблем не возникало, пока приложение не начинало разрастаться.

С точки зрения инфраструктуры, каждый сервер может выполнять множество приложений в одном узле и применять допустимое

соотношение эффективности использования ресурсов, как показано на рисунке 7.

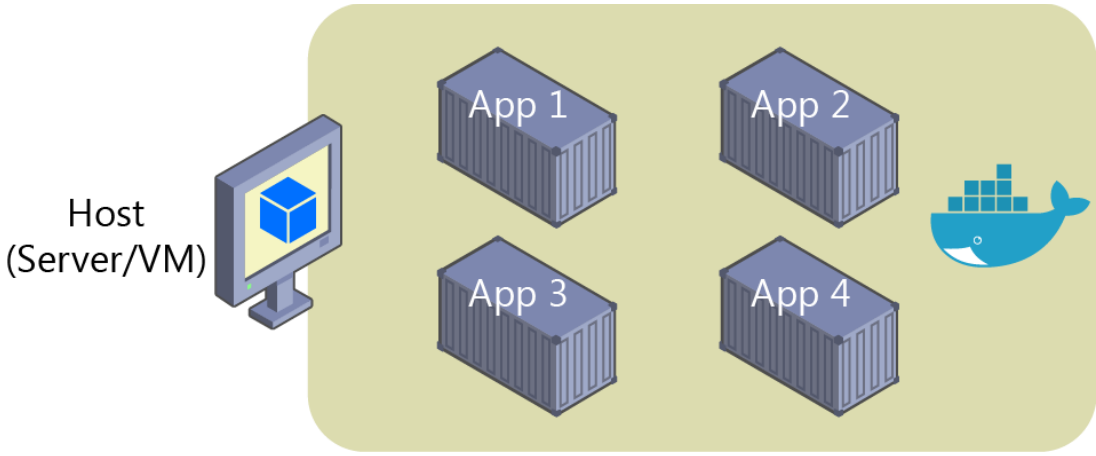


Рисунок 7 – Узел, выполняющий несколько приложений/контейнеров

Наконец, с точки зрения доступности монолитные приложения нужно развертывать целиком. Это означает, что если требуется выполнить *остановку и запуск*, в течение периода развертывания будут затронуты все функциональные возможности и все пользователи. В некоторых случаях использование Azure и контейнеров позволяет свести эти ситуации к минимуму, а также снизить вероятность простоя приложения, как показано на рисунке 7.

Монолитные приложения можно развернуть в Azure с помощью выделенных виртуальных машин для каждого экземпляра. С помощью масштабируемых наборов виртуальных машин Azure можно легко масштабировать виртуальные машины.

Службы приложений Azure также позволяют выполнять монолитные приложения и легко масштабировать экземпляры без управления виртуальными машинами. Службы приложений Azure также могут выполнять отдельные экземпляры контейнеров Docker, упрощая развертывание.

Вы можете развернуть несколько виртуальных машин в качестве узлов Docker и запустить любое количество контейнеров на виртуальную

машину. Затем с помощью Azure Load Balancer вы можете управлять масштабированием, как показано на рисунке 8.

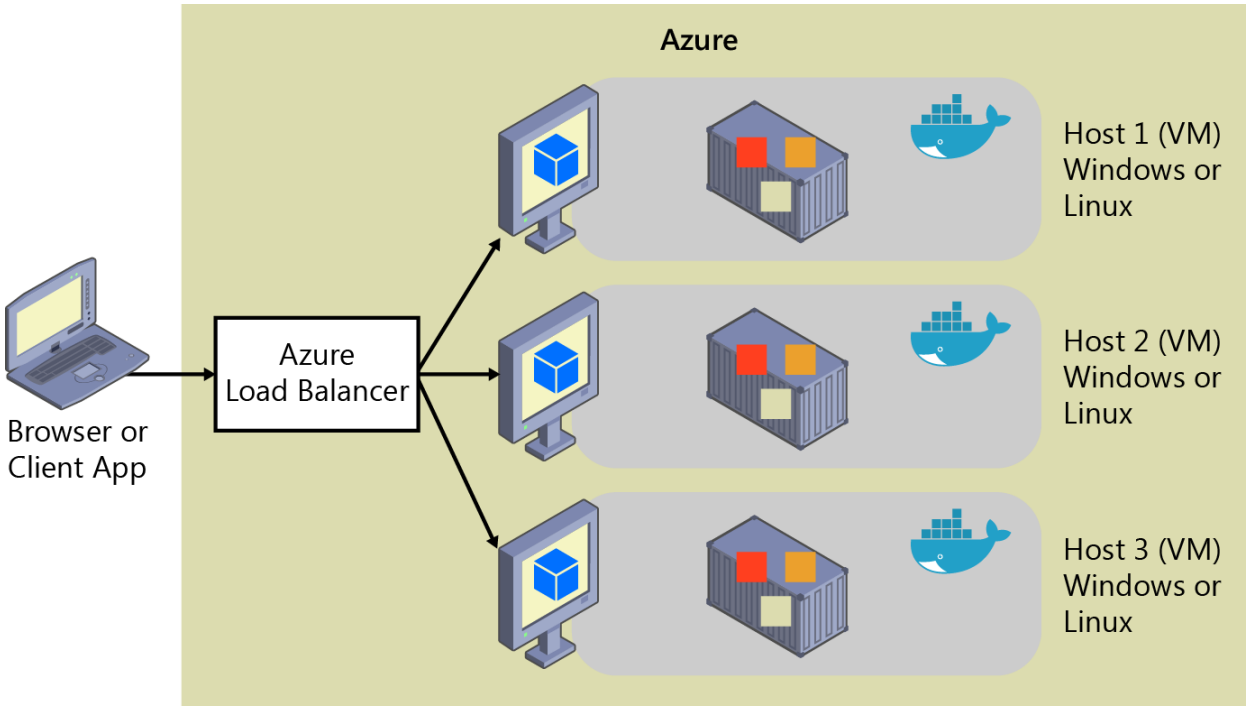


Рисунок 8 – Масштабирование одного приложения Docker с использованием нескольких узлов

Развертыванием самих узлов можно управлять с помощью традиционных методов развертывания.

Вы можете управлять контейнерами Docker из командной строки с помощью таких команд, как `docker run` и `docker-compose up`, а также автоматизировать эту процедуру с помощью конвейеров непрерывной поставки (CD) и, например, выполнить развертывание на узлах Docker из Azure DevOps Services.

Развертывание монолитного приложения в контейнере

Использование контейнеров для управления монолитными развертываниями имеет свои преимущества. Масштабировать экземпляры контейнера гораздо быстрее и проще, чем развертывать дополнительные виртуальные машины.

Развертывание обновлений в виде образов Docker выполняется гораздо быстрее и эффективнее с точки зрения использования сети. Контейнеры Docker обычно запускаются за считаные секунды, что ускоряет выпуск. Демонтировать контейнер Docker можно с помощью команды `docker stop`, и обычно для этого требуется меньше секунды.

Так как контейнеры по своей природе являются неизменяемыми, вам не придется беспокоиться о возможности повреждения виртуальной машины, когда сценарии обновления не учитывают некоторые оставшиеся на диске конфигурации или файлы.

Docker имеет много плюсов для монолитных приложений, и мы лишь слегка затрагиваем эту тему. Более обширные возможности при управлении контейнерами открываются благодаря развертыванию с помощью оркестраторов контейнеров, которые управляют различными экземплярами и жизненным циклом каждого экземпляра контейнера. Когда вы разбиваете монолитное приложение на подсистемы, которые затем можно масштабировать, разрабатывать и развертывать по отдельности, вы переходите на уровень микрослужб.

Публикация отдельного приложения на основе контейнера Docker в службе приложений Azure

Когда вы хотите быстро проверить контейнер, развернутый в Azure, или когда приложение основано на одном контейнере, вы можете воспользоваться удобной функцией предоставления масштабируемых служб на основе одного контейнера в службах приложений Azure.

Она интуитивно понятна и прекрасно интегрируется с Git, что ускоряет работу, так как вы можете взять свой код, выполнить его сборку в Visual Studio и развернуть его прямо в Azure. В обычном случае (без Docker), если вам требовались другие возможности, платформы или зависимости, не поддерживаемые в службах приложений, вам потребовалось бы подождать, пока команда разработчиков Azure не обновит эти зависимости в службе приложений, либо переключиться на другие службы, например Service Fabric, облачные службы или даже

обычные виртуальные машины, где вы можете полнее контролировать процесс и установить необходимый компонент или необходимую платформу для своего приложения.

Как показано на рисунке 9, при использовании Visual Studio 2022 поддержка контейнеров в Службе приложений Azure позволяет включать в среду приложения любые компоненты. Если вы добавили зависимость в приложение, то поскольку оно выполняется в контейнере, вы можете включить такие зависимости в Dockerfile или образ Docker.

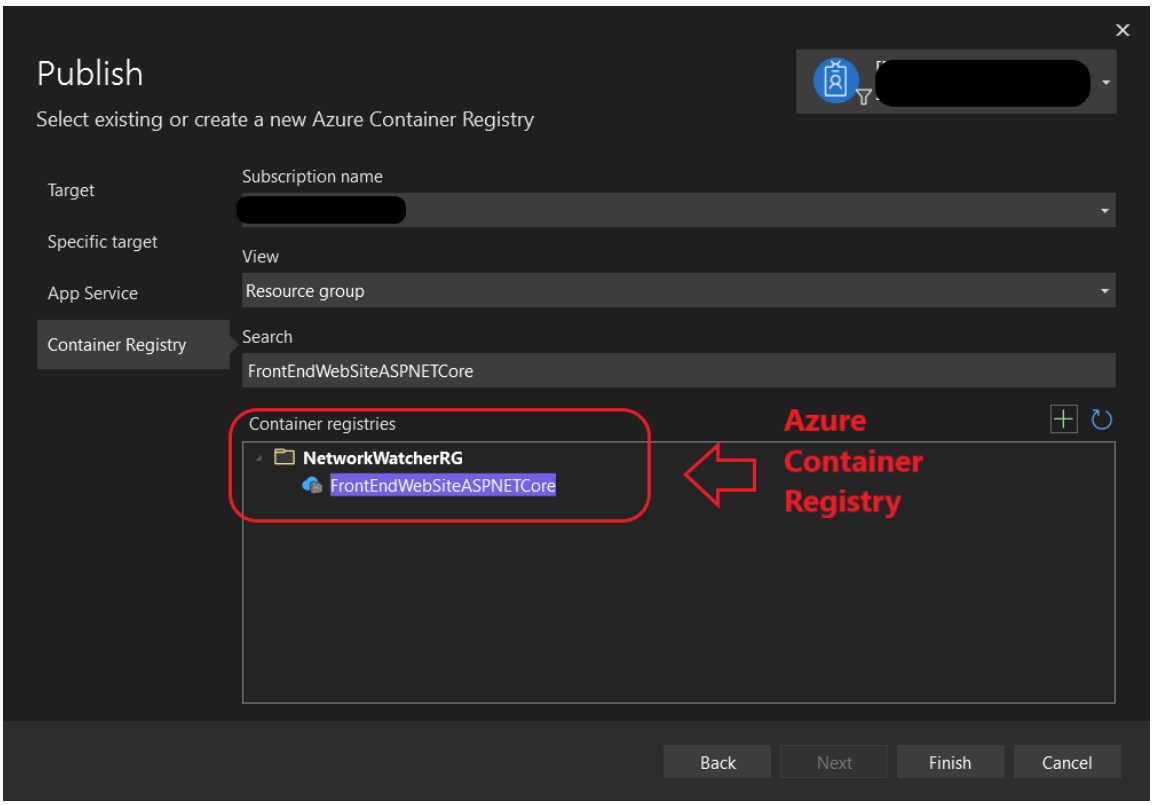


Рисунок 9 – Публикация контейнера в службе приложений из приложений/контейнеров Visual Studio

На рисунке 9 также указано, что поток публикации отправляет образ через реестр контейнеров. Это может быть Реестр контейнеров Azure (реестр, близкий к вашим развертываниям в Azure и защищенный группами и учетными записями в Azure Active Directory) или другой реестр Docker, например Docker Hub или локальные реестры.

Состояние и данные в приложениях Docker

Представьте себе контейнер как экземпляр процесса. Процесс не находится в неизменном состоянии. И хотя контейнер может записывать данные в локальное хранилище, предполагать, что экземпляр всегда будет на месте — это все равно, что надеяться на постоянство одной ячейки памяти. Следует считать, что образы контейнеров, как и процессы, имеют несколько экземпляров и в конечном итоге будут завершены. Если они управляются с помощью оркестратора контейнеров, они могут перемещаться между узлами или виртуальными машинами.

Для управления постоянными данными в приложениях Docker используются следующие решения:

Из узла Docker в качестве томов Docker:

- **Тома** хранятся в той части файловой системы узла, которая находится под управлением Docker.
- **Подключения привязок** можно сопоставить с любой папкой в файловой системе узла, поэтому доступом невозможно управлять из процесса Docker, что может представлять угрозу для безопасности, поскольку контейнер может получить доступ к конфиденциальным папкам операционной системы.
- **Подключения tmpfs** аналогичны виртуальным папкам, которые существуют только в памяти узла и никогда не записываются в файловую систему.

Из удаленного хранилища:

- Служба хранилища Azure, предоставляющая георасределенное хранилище для долгосрочного хранения данных контейнеров.
- Удаленные реляционные базы данных, например База данных SQL Azure, базы данных NoSQL, такие как Azure Cosmos DB, или службы кэша, такие как Redis.

Из контейнера Docker:

- Docker имеет функцию под названием *оверлейная файловая система*. Эта функция выполняет копирование при записи, при котором обновленная информация сохраняется в корневой файловой системе контейнера. Эта информация "накладывается" на изначальный образ, на котором базируется контейнер. Если удалить контейнер из системы, эти изменения будут утеряны. Поэтому, хоть и возможно сохранить состояние контейнера в его локальном хранилище, создание системы на базе этой функции будет противоречить конструкции контейнера, состояние которого по умолчанию не отслеживается.
- Тем не менее тома Docker являются предпочтительным способом обработки локальных данных в Docker. Если вам требуются дополнительные сведения о хранении в контейнерах, ознакомьтесь с разделами Docker storage drivers (Драйверы хранилища Docker) и About images, containers, and storage drivers (Основные сведения об образах, контейнерах и драйверах хранилища).

Ниже подробно описаны эти варианты.

Тома — это каталоги из ОС узла, сопоставленные с каталогами в контейнерах. Если код в контейнере имеет доступ к каталогу, на самом деле он обращается к каталогу на ОС узла. Этот каталог не привязан к времени существования самого контейнера, находится под управлением Docker и изолирован от основных функциональных возможностей хост-компьютера. Поэтому тома данных хранят данные независимо от контейнера. Если удалить контейнер или образ из узла Docker, данные из томов данных не будут удалены.

Тома могут быть именованными или анонимными (по умолчанию). Именованные тома — это следующий этап развития **контейнеров томов данных**. Они упрощают обмен данными между контейнерами. Тома также поддерживают драйверы томов, позволяющие хранить данные на удаленных узлах и т. д.

Подключения привязки доступны довольно давно и позволяют сопоставлять любую папку с точкой подключения в контейнере. Подключения привязки имеют больше ограничений, чем тома, и с ними связаны некоторые проблемы безопасности, поэтому тома являются рекомендуемым способом.

Подключения `tmpfs` являются виртуальными папками, которые существуют только в памяти узла и никогда не записываются в файловую систему. Они быстрые и безопасные, но используют ресурсы памяти и предназначены только для временных данных.

Как показано на рисунке 10, обычные тома Docker могут храниться за пределами самих контейнерами, но в физических границах сервера узла или виртуальной машины. Тем не менее, контейнеры Docker с одного сервера узла или виртуальной машины не могут обращаться к тому на другом сервере узла или виртуальной машине. Другими словами, с помощью этих томов невозможно управлять данными контейнеров, которые выполняются на разных узлах Docker, хотя такой режим работы можно реализовать с помощью драйвера томов, который поддерживает удаленные узлы.

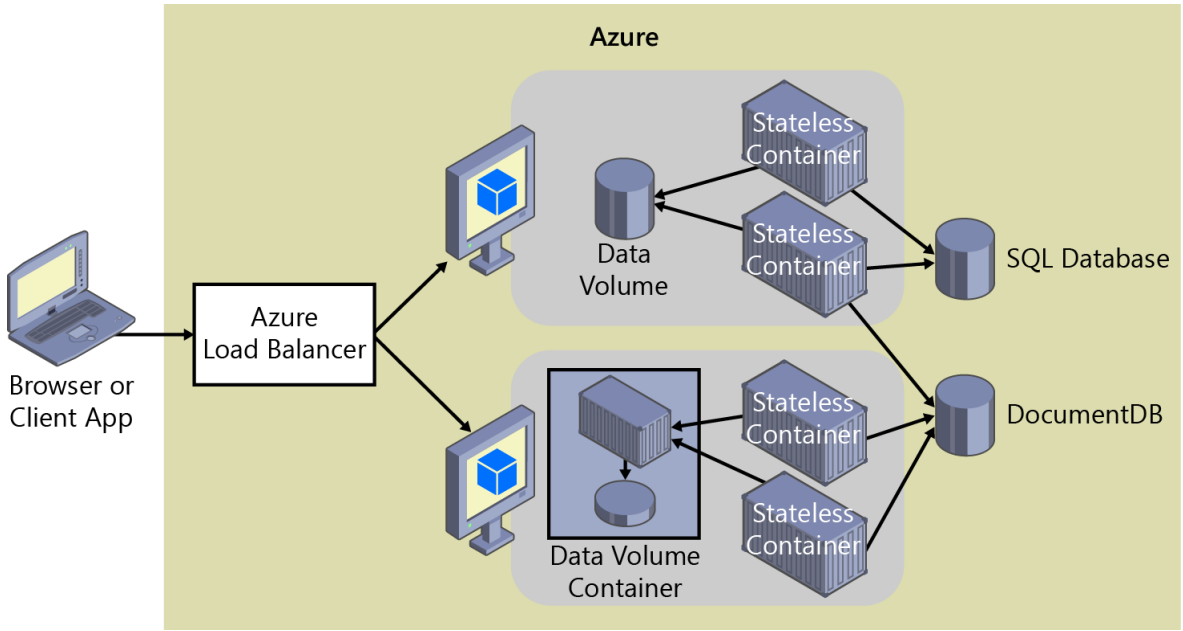


Рисунок 10 – Тома и внешние источники данных для приложений на основе контейнера

Кроме того, когда контейнеры Docker управляются оркестратором, контейнеры могут "перемещаться" между узлами в рамках оптимизации, выполняемой кластером. Поэтому тома данных не рекомендуется использовать для бизнес-данных. Но это хороший инструмент для работы с файлами трассировки, временными файлами или подобными элементами, которые не влияют на целостность бизнес-данных.

Удаленные источники данных и инструменты кэширования, такие как база данных Azure SQL, Azure Cosmos DB или служба кэша, например Redis, можно использовать в упакованных в контейнеры приложениях так же, как они используются при разработке без контейнеров. Это проверенный способ хранения данных бизнес-приложений.

служба хранилища Azure. Бизнес-данные, как правило, необходимо хранить во внешних ресурсах или базах данных, например в службе хранилища Azure. Служба хранилища Azure предоставляет следующие службы в облаке:

- Хранилище BLOB-объектов хранит объекты с неструктурированными данными. Большой двоичный объект может представлять собой текстовые или двоичные данные любого типа, например документ или файл мультимедиа (файлы с изображением, аудио и видео). Хранилище BLOB-объектов иногда также называют хранилищем объектов.
- Хранилище файлов — это совместно используемое хранилище для старых приложений, доступ к которому осуществляется на основе стандартного протокола SMB. Виртуальные машины Azure и облачные службы могут совместно использовать данные файлов в компонентах приложений через подключенный общий ресурс. Локальные приложения могут обращаться к данным файлов в общем ресурсе через REST API файловой службы.
- Табличное хранилище содержит структурированные наборы данных. Хранилище таблиц — это хранилище данных NoSQL типа

"ключ-значение", обеспечивающее быструю разработку и доступ к большим объемам данных.

Реляционные базы данных и базы данных NoSQL. Существует множество вариантов внешних баз данных — от реляционных, таких как SQL Server, PostgreSQL и Oracle, до баз данных NoSQL, например Azure Cosmos DB, MongoDB и т д. Такие базы представляют собой отдельную тему и не рассматриваются в этом руководстве.

Сервисноориентированные приложения

Термин "сервисноориентированная архитектура" (SOA) для разных людей означает разные понятия. Однако общий момент заключается в том, что термин SOA подразумевает структурирование архитектуры приложения путем разделения ее на несколько служб (чаще всего HTTP-службы), которые можно классифицировать различными способами, например как подсистемы или уровни.

Сейчас эти службы можно развернуть как контейнеры Docker, которые помогают решать проблемы развертывания, так как в образ контейнера включены все зависимости. Однако, если при развертывании вы используете отдельные экземпляры, то при необходимости масштабирования приложений SOA вы столкнетесь с проблемами. Эту проблему можно решить с помощью программного обеспечения кластеризации Docker или оркестратора. Вы рассмотрите оркестраторы более подробно в следующем разделе, при изучении подходов на основе микрослужб.

Контейнеры Docker являются полезным (но необязательным) элементом как для традиционных архитектур, ориентированных на службы, так и более сложных архитектур с микрослужбами.

В целом решения кластеризации на базе контейнеров удобны как для традиционной архитектуры SOA, так и для более сложной архитектуры микрослужб, где каждая микрослужба имеет свою модель данных. А благодаря нескольким базам данных также есть возможность масштабировать уровень данных вместо работы с монолитными базами данных, совместно используемыми службами SOA. Однако обсуждение разделения данных касается исключительно архитектуры и проектирования.

Управление микрослужбами и многоконтейнерными приложениями для обеспечения высокого уровня масштабируемости и доступности

Использование оркестраторов для приложений, готовых к развертыванию в рабочей среде, крайне важно, если приложение основано на микрослужбах или разнесено по нескольким контейнерам. Как было сказано ранее, в рамках подхода на основе микрослужб каждая микрослужба имеет собственную модель и данные, поэтому она является автономной с точки зрения разработки и развертывания. Но если у вас есть более традиционное бизнес-приложение, состоящее из нескольких служб (например, SOA) и требующее распределенного развертывания, оно также будет включать в себя несколько контейнеров или служб. Такие системы сложны в масштабировании и управлении, поэтому для создания масштабируемого многоконтейнерного приложения, готового к развертыванию в рабочей среде, оркестратор абсолютно необходим.

На рисунке 11 показано развертывание приложения, состоящего из нескольких микрослужб (контейнеров) в кластере.

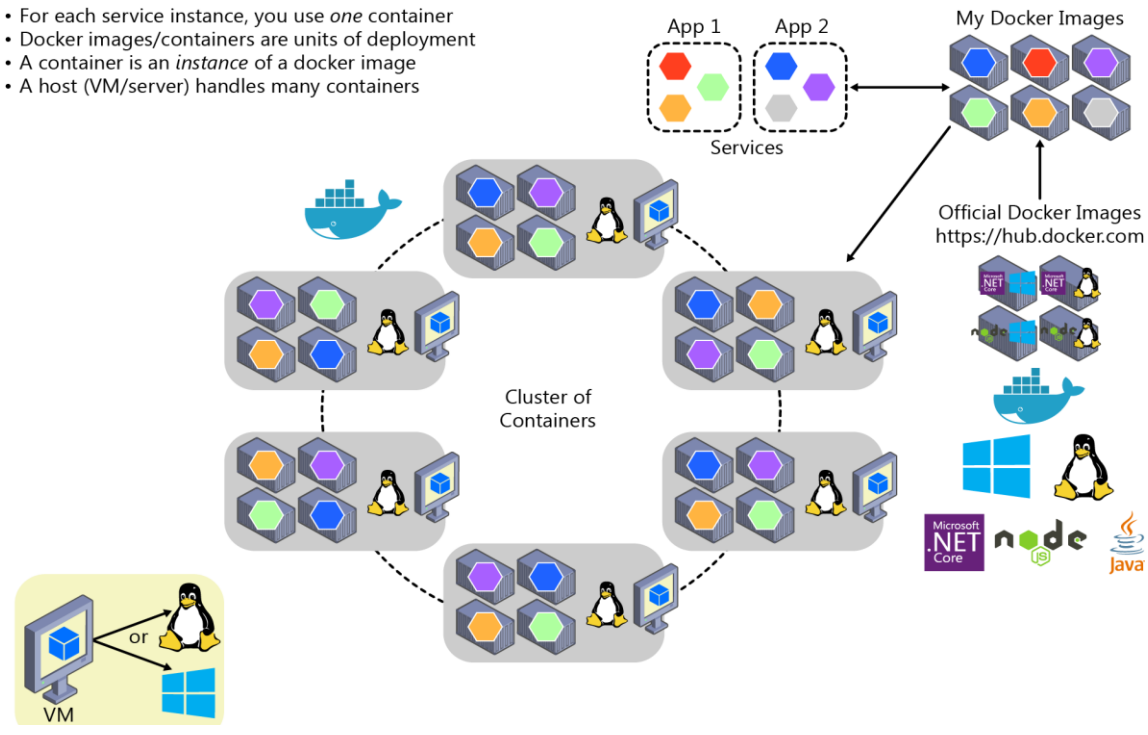


Рисунок 10 – Кластер контейнеров

Такой подход выглядит логичным. Однако как обеспечить балансировку нагрузки, маршрутизацию и оркестрацию для таких составных приложений?

Docker CLI позволяет управлять одним контейнером в одном узле, но этого недостаточно, если требуется управлять множеством контейнеров, развернутых в нескольких узлах для более сложных распределенных приложений. В большинстве случаев необходима платформа управления, которая будет автоматически запускать контейнеры, масштабировать контейнеры с несколькими экземплярами в каждом образе, а также при необходимости приостанавливать или завершать их работу. В идеале она также должна контролировать доступ к ресурсам, таким как сеть и хранилище данных.

Чтобы перейти от управления отдельными контейнерами или простыми составными приложениями к управлению крупными корпоративными приложениями с микрослужбами, следует прибегнуть к платформам оркестрации и кластеризации.

С точки зрения архитектуры и разработки, если вы создаете крупные корпоративные приложения на основе микрослужб, важно знать следующие платформы и продукты, которые поддерживают сложные сценарии:

- **Кластеры и оркестраторы.** Если вам нужно масштабировать приложения на множестве узлов Docker, как в случае с крупным приложением на основе микрослужб, крайне важно иметь возможность управлять всеми этими узлами как единым кластером, абстрагируясь от сложности базовой платформы. Именно такую возможность предоставляют кластеры контейнеров и оркестраторы. Примерами оркестраторов могут служить Azure Service Fabric и Kubernetes. Функции Kubernetes доступны в Azure через службу Azure Kubernetes.
- **Планировщики.** *Планирование* означает, что администратор может запускать контейнеры в кластере, поэтому планировщики

также предоставляют пользовательский интерфейс для этого. Планировщик кластера имеет несколько функций: обеспечение эффективного использования ресурсов в кластере, применение заданных пользователем ограничений, эффективная балансировка нагрузки контейнеров между узлами, а также обеспечение устойчивости к ошибкам и высокой доступности.

Понятия кластера и планировщика тесно связаны друг с другом, поэтому продукты, предоставляемые разными поставщиками, часто включают в себя оба набора возможностей. В следующем разделе приведен список наиболее важных платформ и программного обеспечения, предоставляющих возможности управления кластерами и планирования. Эти оркестраторы, как правило, предлагаются в составе общедоступных облачных платформ, таких как Azure.

Программные платформы для кластеризации контейнеров, оркестрации и планирования

Платформа	Комментарии
Kubernetes 	*Kubernetes* — это решение с открытым кодом, которое предоставляет широкий ряд возможностей: от организации инфраструктуры кластера и планирования контейнеров до оркестрации. Оно позволяет автоматизировать развертывание, масштабирование и выполнение операций с контейнерами приложений в кластерах узлов. *Kubernetes* предоставляет ориентированную на контейнеры инфраструктуру, которая объединяет контейнеры приложений в логические блоки, чтобы упростить управление и обнаружение. Решение *Kubernetes* является зрелым в Linux и менее зрелым в Windows.
Служба Azure Kubernetes (AKS) 	Служба Azure Kubernetes (AKS) является управляемой службой оркестрации контейнеров Kubernetes в Azure, которая упрощает управление, развертывание и эксплуатацию кластера Kubernetes.
Azure Service Fabric 	Service Fabric — это платформа микрослужб от Майкрософт, которая предназначена для создания приложений. Она позволяет оркестрировать службы и создавать кластеры компьютеров. Службы в Service Fabric могут развертываться как контейнеры или обычные процессы. Кластеры *Service Fabric* могут быть развернуты в Azure, локально или в любом облаке. Тем не менее управляемый подход упрощает развертывание в Azure. *Service Fabric* предоставляет дополнительные перспективные модели программирования Service Fabric, такие как службы с отслеживанием состояния и Reliable Actors. Решение *Service Fabric* является зрелым в Windows (годы развития в этой среде) и менее зрелым в Linux. С 2017 года в Service Fabric поддерживаются как контейнеры Linux, так и контейнеры Windows.

Платформа	Комментарии
Сетка Azure Service Fabric 	*Сетка Azure Service Fabric* обеспечивает тот же уровень надежности, критически важной производительности и масштабируемости, что и Service Fabric, но также предлагает полностью управляемую бессерверную платформу. Нет необходимости управлять кластером, виртуальными машинами, хранилищем и конфигурацией сети. Вы можете сконцентрировать усилия на разработке приложений. *Сетка Service Fabric* поддерживает контейнеры Windows и Linux, позволяя осуществлять разработку на любом языке программирования и платформе.
Приложения-контейнеры Azure 	Контейнеры приложений Azure — это управляемая бессерверная служба контейнеров, предназначенная для создания и развертывания современных приложений в большом масштабе.

Использование оркестраторов на основе контейнеров в Azure

Ряд поставщиков облачных служб, включая Azure, Amazon EC2 Container Service и Google Container Engine, предлагают поддержку контейнеров Docker, а также кластеров Docker и оркестрации. Azure поддерживает кластер и оркестратор Docker посредством службы Azure Kubernetes (AKS), Azure Service Fabric и сетки Azure Service Fabric.

Использование службы Azure Kubernetes

Кластер Kubernetes объединяет несколько узлов Docker в пул и предоставляет доступ к ним как к единому виртуальному узлу Docker, что позволяет

развертывать несколько узлов в кластере и осуществлять масштабирование, добавляя любое число экземпляров контейнеров. Кластер отвечает за выполнение всех сложных задач управления, включая масштабирование, обеспечение работоспособности и т. д.

Служба AKS позволяет упростить создание и настройку кластера виртуальных машин в Azure, предварительно настроенных для выполнения упакованных в контейнеры приложений, а также управление им. Благодаря оптимизированной конфигурации популярных средств планирования и оркестрации с открытым кодом служба AKS дает возможность развертывать приложения на основе контейнеров в Microsoft Azure и управлять ими, используя имеющиеся навыки или прибегая к знаниям большого и продолжающего расширяться сообщества.

Служба Azure Kubernetes оптимизирует настройку популярных средств и технологий кластеризации Docker с открытым кодом для Azure. Вы получаете открытое решение, которое обеспечивает переносимость как контейнеров, так и конфигурации приложения. Вы выбираете размер, число узлов и средства оркестрации, а служба Azure Kubernetes делает все остальное.

На рисунке 11 показана структура кластера Kubernetes, где главный узел (виртуальная машина) управляет большинством операций по координации кластера, и вы можете развертывать контейнеры в остальных узлах, управляемых как единый пул с точки зрения приложения. Это позволяет расширять среду до тысяч и даже десятков тысяч контейнеров.

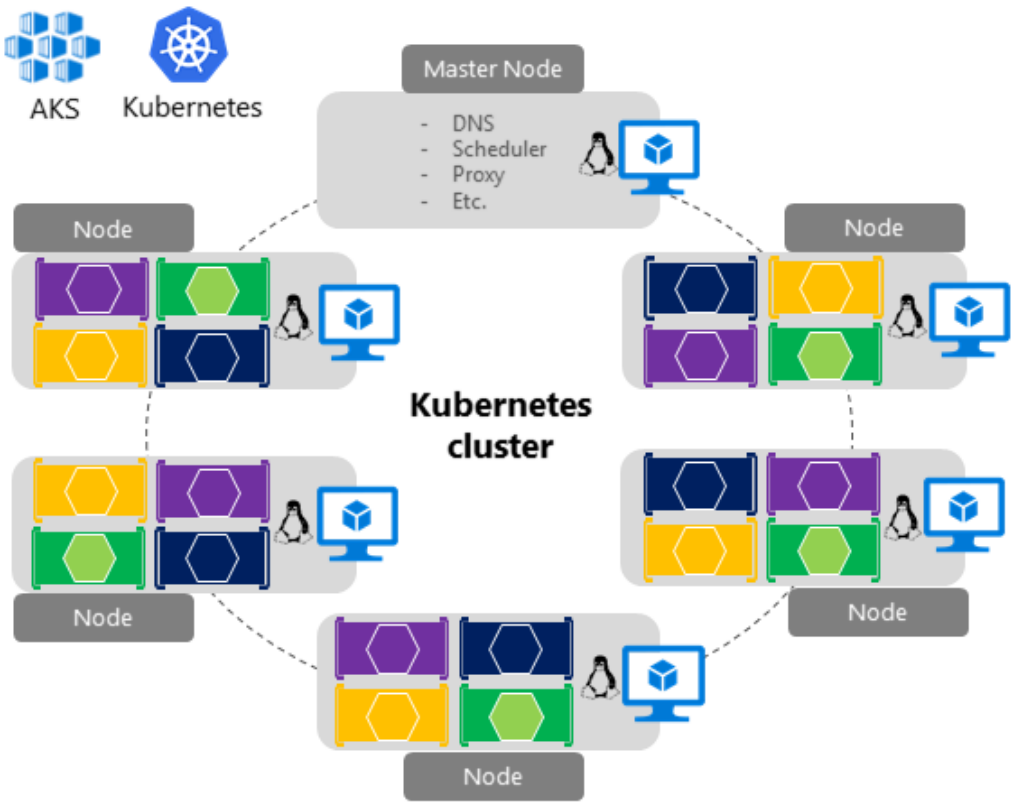


Рисунок 11 – Упрощенная структура и топология кластера Kubernetes

Среда разработки для Kubernetes

Что касается среды разработки, которую компания Docker анонсировала в июле 2018 г., Kubernetes также можно запускать на одном компьютере разработки (Windows 10 или macOS), просто установив Docker Desktop. Позднее можно развернуть ее в облаке (AKS) для дальнейшего тестирования интеграции, как показано на рисунке 12.

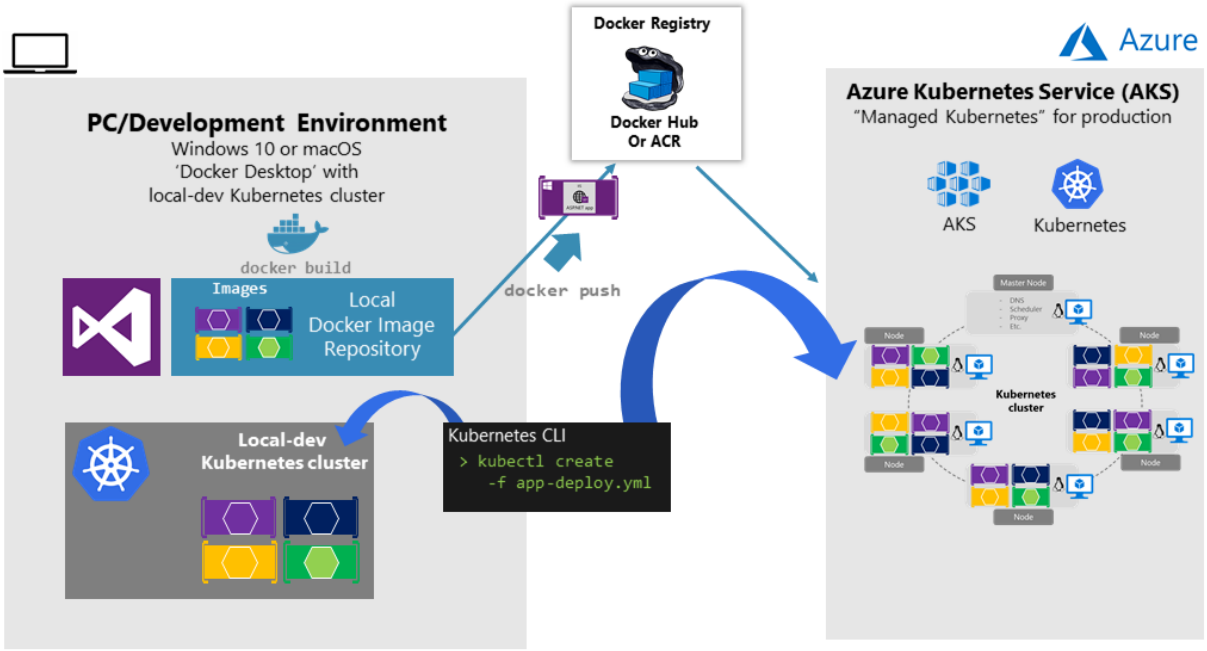


Рисунок 12 Выполнение Kubernetes на компьютере разработки и в облаке

Начало работы со службой Azure Kubernetes (AKS)

Чтобы начать использовать AKS, необходимо развернуть кластер AKS с помощью портала Azure или интерфейса командной строки. Дополнительные сведения о развертывании кластера Kubernetes в Azure см. в статье Развертывание кластера службы Azure Kubernetes (AKS).

Все возможности по умолчанию реализуются с помощью ПО с открытым кодом.

Развертывание с помощью чартов Helm в кластерах Kubernetes

При развертывании приложения в кластере Kubernetes можно использовать оригинальное средство CLI `kubectl.exe`, использующее файлы развертывания на основе собственного формата (`.yaml`-файлы), как уже упоминалось в предыдущем разделе. Однако для более сложных приложений Kubernetes, например при развертывании сложных приложений на основе микрослужб, рекомендуется использовать Helm.

Чарты Helm помогают осуществлять определение, управление версиями, установку, предоставление общего доступа, обновление или откат даже самых сложных приложений Kubernetes. Поддержка Helm осуществляется фондом Cloud Native Computing Foundation (CNCF) в сотрудничестве с корпорацией Майкрософт, Google, Bitnami и сообществом Helm.

Использование Azure Service Fabric

Платформа Azure Service Fabric создана в рамках перехода корпорации Майкрософт от предоставления "готовых" продуктов (зачастую монолитных) к предоставлению служб. При ее создании учитывался опыт разработки и эксплуатации масштабных служб, например базы данных SQL Azure, Azure Cosmos DB, служебной шины Azure или базы данных Кортаны. Она продолжала развиваться по мере внедрения во множестве служб. Важно отметить, что Service Fabric работает не только в среде Azure, но и в автономных развертываниях Windows Server.

Service Fabric помогает решить сложные проблемы при разработке и запуске службы, связанные с эффективным использованием ресурсов инфраструктуры, что позволяет командам решать бизнес-задачи, применяя подход на основе микрослужб.

Service Fabric предоставляет два масштабных решения, которые позволяют создавать приложения в рамках подхода на основе микрослужб.

- Платформа, которая предоставляет системные службы для развертывания, обновления, обнаружения и перезапуска служб, в которых произошел сбой, а также обнаружения служб, управления состоянием и мониторинга работоспособности. В сущности, эти системные службы позволяют задействовать множество характеристик микрослужб, описанных ранее.
- Программные интерфейсы (API) или платформы, позволяющие создавать приложения как микрослужбы: Reliable Actors и Reliable Services. Для создания микрослужбы можно использовать любой код. Однако программные интерфейсы не только упрощают задачу, но и обеспечивают более глубокую интеграцию с платформой. Так вы можете, например, получать информацию о работоспособности и данные диагностики, а также пользоваться преимуществами средств управления надежным состоянием.

Для Service Fabric не важен способ создания службы, поэтому вы можете использовать любую технологию. Но эта платформа предоставляет встроенные интерфейсы API, упрощающие создание микрослужб.

Как показано на рисунке 13, вы можете создавать и запускать микрослужбы в Service Fabric как простые процессы или как контейнеры Docker. Микрослужбы на основе контейнеров можно также сочетать с микрослужбами на основе процессов в рамках одного кластера Service Fabric.

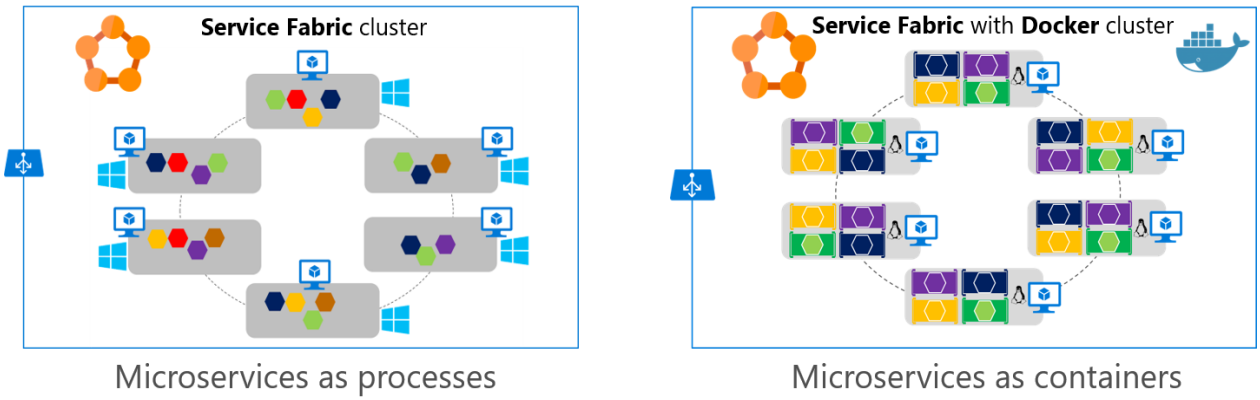


Рисунок 13 – Развертывание микрослужб в виде процессов или контейнеров в Azure Service Fabric

На первом изображении микрослужбы отображаются в виде процессов, где каждый узел выполняет один процесс для каждой микрослужбы. На втором изображении микрослужбы отображаются в виде контейнеров, где каждый узел выполняет Docker с несколькими контейнерами, по одному контейнеру на микрослужбу. В кластерах Service Fabric на основе узлов Linux и Windows могут выполняться соответственно контейнеры Docker Linux и контейнеры Windows.

Как показано на рисунке 14, с точки зрения логической архитектуры микрослужб при создании надежной службы Service Fabric с отслеживанием состояния обычно требуется реализовать два уровня служб. Первый из них — это внутренняя надежная служба с отслеживанием состояния, которая обеспечивает работу нескольких разделов (каждый раздел является службой с отслеживанием состояния). Второй уровень — это внешняя служба или служба шлюза, которая отвечает за маршрутизацию и объединение данных в разных разделах или экземплярах служб с отслеживанием состояния. Служба шлюза также отвечает за взаимодействие с клиентом и выполнение повторных обращений к внутренней службе. Службой шлюза называется реализуемая вами пользовательская служба, но вы также можете использовать готовую службу обратного прокси-сервера Service Fabric.

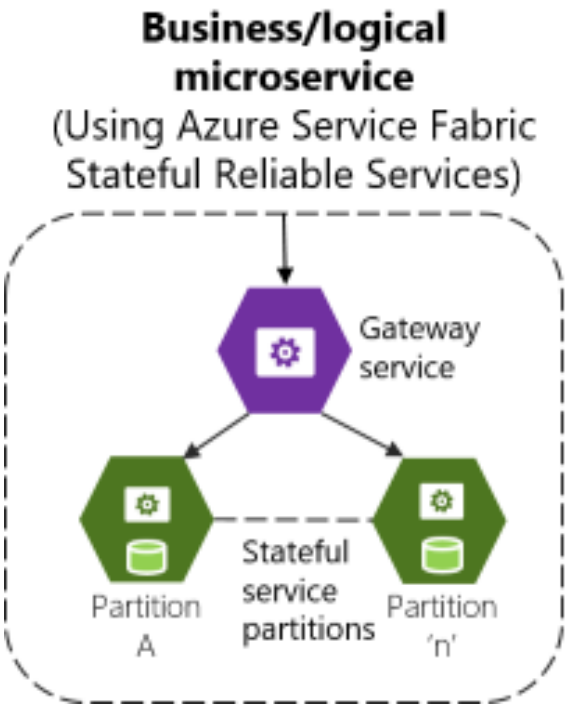


Рисунок 14 – Бизнес-микрослужба с несколькими экземплярами службы с отслеживанием состояния и пользовательской внешней службой шлюза

В любом случае при использовании надежных служб Service Fabric с отслеживанием состояния у вас также есть логическая микрослужба или бизнес-микрослужба (ограниченный контекст), которая состоит из нескольких физических служб. Каждую из служб шлюза и служб раздела можно реализовать как службу веб-API ASP.NET, как показано на рисунке 15. Service Fabric поддерживает несколько надежных служб с отслеживанием состояния в контейнерах.

В Service Fabric службы можно объединять в группы и развертывать как приложение Service Fabric, которое представляет собой модуль упаковки и развертывания для оркестратора или кластера. Таким образом, приложение Service Fabric можно сопоставить с автономной логической микрослужбой или ограниченным контекстом, чтобы развертывать службы в автономном режиме.

Service Fabric и контейнеры

Вы также можете развертывать службы в образах контейнеров в кластере Service Fabric. Как показано на рисунке 15, для каждой службы обычно имеется один контейнер.

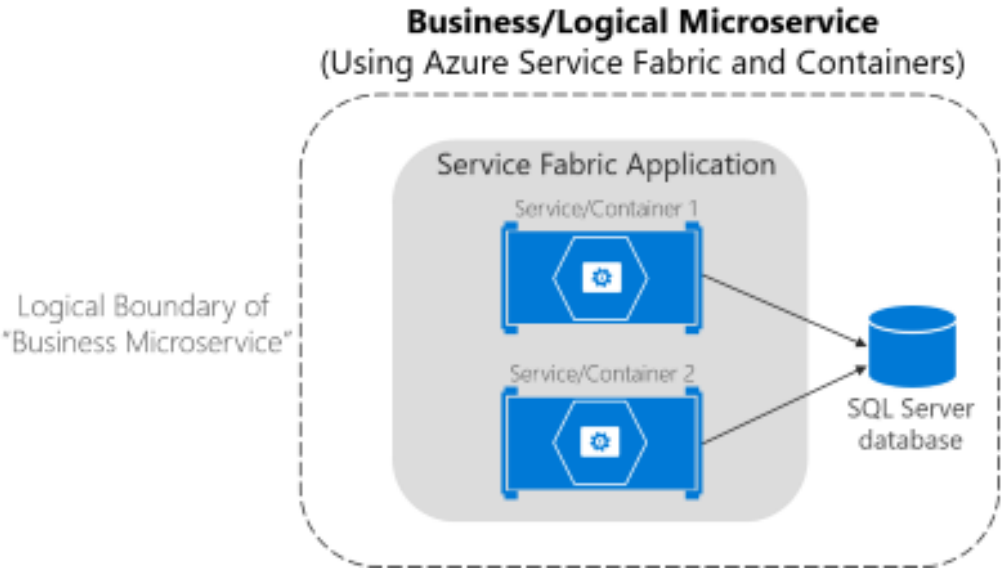


Рисунок 15 – Бизнес-микрослужба с несколькими службами (контейнерами) в Service Fabric

Приложение Service Fabric может выполнять несколько контейнеров, обращающихся к внешней базе данных, и весь набор будет логической границей бизнес-микрослужбы. Однако в Service Fabric также возможны так называемые "сцепленные" контейнеры (два контейнера, которые должны развертываться вместе в составе логической службы). Важным моментом является то, что бизнес-микрослужба — это логическая граница вокруг нескольких связанных элементов. Во многих случаях это может быть одна служба с единственной моделью данных, однако в других она может представлять собой несколько физических служб.

Можно сочетать службы в процессах и службы в контейнерах в рамках одного приложения Service Fabric, как показано на рисунке 16.

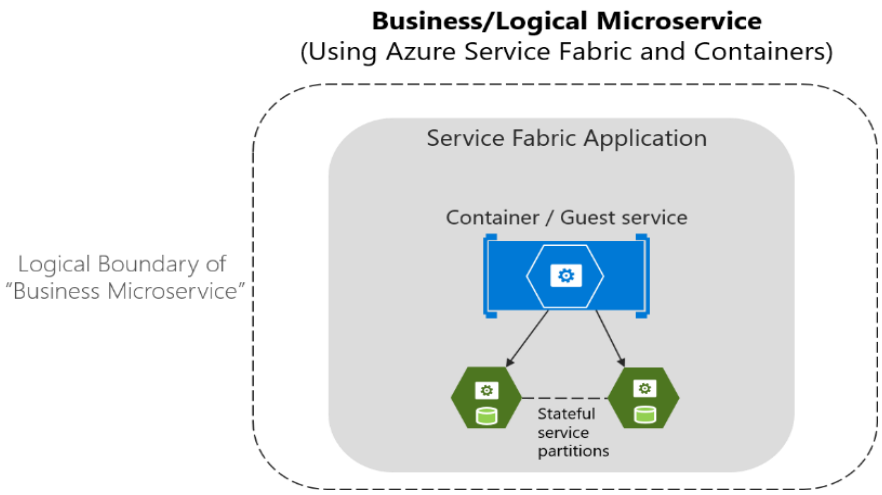


Рисунок 16 – Бизнес-микрослужба, сопоставленная с приложением Service Fabric с контейнерами и службами с отслеживанием состояния

Микрослужбы с отслеживанием и без отслеживания состояния

Как уже упоминалось, у каждой микрослужбы (логического ограниченного контекста) должна быть своя модель предметной области (данные и логика). В случае микрослужб без отслеживания состояния базы данных будут внешними — с применением либо реляционных СУБД, таких как SQL Server, либо СУБД на основе NoSQL, например MongoDB или Azure Cosmos DB.

Однако сами службы могут поддерживать отслеживание состояния в Service Fabric, т. е. данные будут находиться в микрослужбе. Эти данные могут существовать не только на том же сервере, но и внутри процесса микрослужбы, в памяти, а также храниться на жестких дисках и реплицироваться в другие узлы. На рисунке 17 показаны различные подходы.

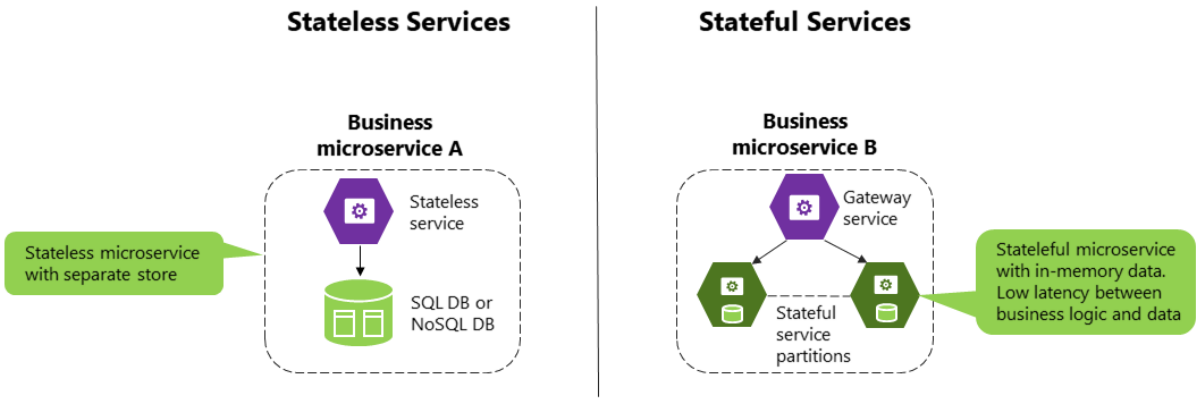


Рисунок 17 – Микрослужбы с отслеживанием и без отслеживания состояния

В службы без отслеживания состояния состояние (постоянное хранение, база данных) хранится за пределами микрослужбы. В службах с отслеживанием состояния состояние хранится внутри микрослужбы. Микрослужбы без отслеживания состояния — вполне допустимый подход, и он проще в реализации, чем микрослужбы с отслеживанием состояния, так как подобен традиционным и общеизвестным шаблонам. Но микрослужбы без отслеживания состояния вызывают задержку при взаимодействии процесса с источниками данных и вдобавок содержат больше "движущихся деталей", что осложняет повышение производительности за счет дополнительного кэша и очередей. Результат таков: вы можете получить более сложные архитектуры со слишком большим количеством уровней.

Напротив, микрослужбы с отслеживанием состояния могут показывать превосходство в сложных сценариях, поскольку при их использовании отсутствуют задержки во взаимодействии логики предметной области с данными. Интенсивная обработка данных, игровые серверы, базы данных как услуга и другие сценарии, требующие низкой задержки, — все они выигрывают от использования служб с отслеживанием состояния, обеспечивая локальное состояние для более быстрого доступа.

Службы без отслеживания состояния и с отслеживанием состояния дополняют друг друга. Например, как видно на диаграмме справа на

рис. 4-14, службу с отслеживанием состояния можно разбить на несколько разделов. Для доступа к этим разделам может потребоваться служба без отслеживания состояния, выступающая в роли службы шлюза и способная обращаться к каждому разделу с помощью ключей разделов.

Использование сетки Azure Service Fabric

Сетка Azure Service Fabric — это полностью управляемая служба, которая позволяет разработчикам создавать и развертывать критически важные приложения без необходимости управлять инфраструктурой. Используйте сетку Service Fabric, чтобы создавать и запускать безопасные распределенные приложения для микрослужб, которые масштабируются по запросу.

Как показано на рисунке 18, приложения, размещенные в сетке Service Fabric, выполняются и масштабируются так, что вам не приходится беспокоиться об инфраструктуре.

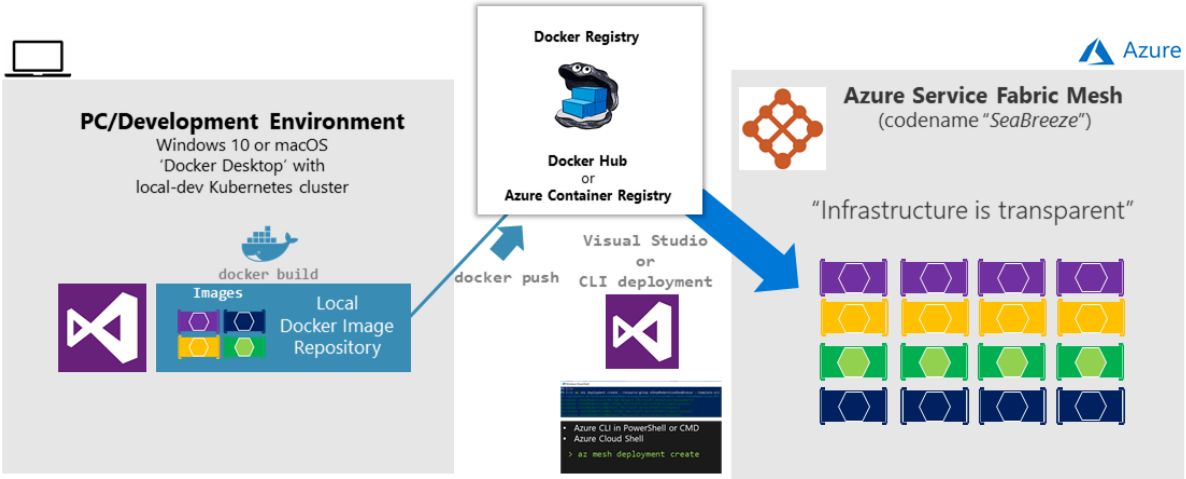


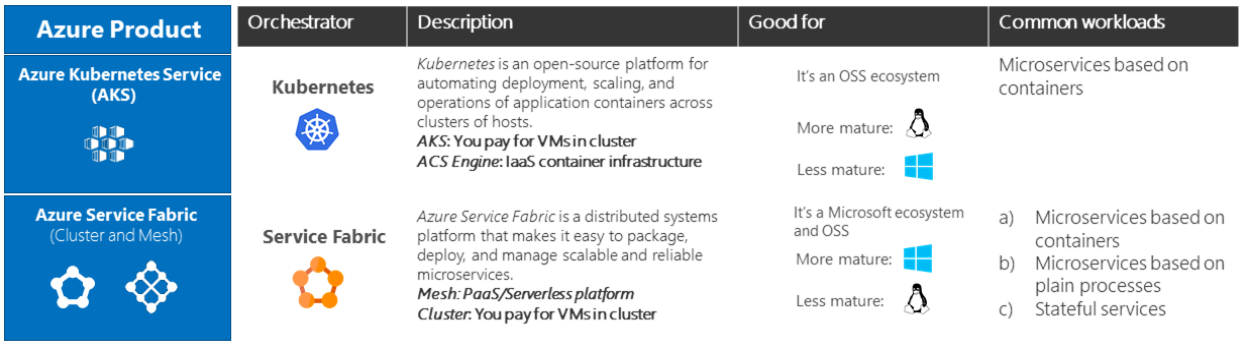
Рисунок 18 – Развертывание приложения для микрослужб или контейнеров в сетке Service Fabric

На самом деле сетка Service Fabric состоит из кластеров из тысяч компьютеров. Все операции в кластере скрыты от разработчика. Нужно просто отправить контейнеры и указать необходимые ресурсы, требования к доступности и ограничения ресурсов. Сетка Service Fabric

автоматически выделяет инфраструктуру, запрошенную развертыванием приложения, а также обрабатывает сбои инфраструктуры, гарантируя высокую доступность ваших приложений. Вы думаете только о работоспособности и скорости реагирования приложения, а не об инфраструктуре.

Выбор оркестраторов в Azure

Ниже приведены рекомендации по выбору оркестратора в зависимости от рабочих нагрузок и фокуса операционной системы.



Azure Product	Orchestrator	Description	Good for	Common workloads
Azure Kubernetes Service (AKS)	Kubernetes	*Kubernetes* is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts. ***AKS:*** **You pay for VMs in cluster** ***ACS Engine:*** **IaaS container infrastructure**	It's an OSS ecosystem More mature: Less mature:	Microservices based on containers
Azure Service Fabric (Cluster and Mesh)	Service Fabric	*Azure Service Fabric* is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices. ***Mesh:*** ***PaaS/Serverless platform*** ***Cluster:*** **You pay for VMs in cluster**	It's a Microsoft ecosystem and OSS More mature: Less mature:	a) Microservices based on containers b) Microservices based on plain processes c) Stateful services

Среда разработки приложений Docker

Выбор средств разработки: интегрированная среда разработки или редактор

Visual Studio Code и CLI Docker (кроссплатформенные средства для Mac, Linux и Windows)

Если вам нужен упрощенный кроссплатформенный редактор, поддерживающий любой язык программирования, вы можете использовать Visual Studio Code и CLI Docker. Эти решения обеспечивают простой, но в то же время эффективный рабочий процесс разработки. Установив Docker для Mac или Docker для Windows (среду разработки), разработчики приложений Docker могут использовать единый интерфейс CLI Docker (среду выполнения), чтобы создавать приложения как для Windows, так и для Linux. Кроме того, Visual Studio Code поддерживает расширения для Docker с IntelliSense для файлов Dockerfile и ярлыками для выполнения команд Docker из редактора.

Visual Studio со средствами Docker (компьютер Windows для разработки)

Мы рекомендуем использовать Visual Studio 2022 (или более поздней версии) с включенными встроенными средствами Docker. С помощью Visual Studio вы можете разрабатывать, запускать и проверять приложения непосредственно в выбранной среде Docker. Нажмите клавишу F5 для отладки приложения (на основе одного контейнера или нескольких) непосредственно в узле Docker или клавиши CTRL+F5 для редактирования и обновления приложения без повторной сборки контейнера. Это самый простой и эффективный способ разработки в Windows контейнеров Docker для Linux или Windows.

Visual Studio для Mac (компьютер Mac для разработки)

При разработке приложений на основе Docker можно использовать Visual Studio для Mac. Visual Studio для Mac — это интегрированная среда разработки с более широкими возможностями по сравнению с Visual Studio Code для Mac.

Языки и платформы

С помощью средств Майкрософт вы можете разрабатывать приложения Docker на самых современных языках. Вот лишь часть их списка:

- .NET и ASP.NET Core
- Node.js
- Go
- Java
- Ruby
- Python
-

Рабочий процесс внутреннего цикла разработки для приложений Docker

Рабочий процесс внутреннего цикла, охватывающий весь цикл DevOps, начинается на компьютере каждого разработчика, где разработчик локально пишет код приложения на предпочитаемых языках и платформах, а затем тестирует его (рисунок 19). Независимо от выбранного языка и платформы этот рабочий процесс имеет одну общую черту: В этом конкретном рабочем процессе вы всегда разрабатываете и тестируете контейнеры Docker не в других средах, а локально.

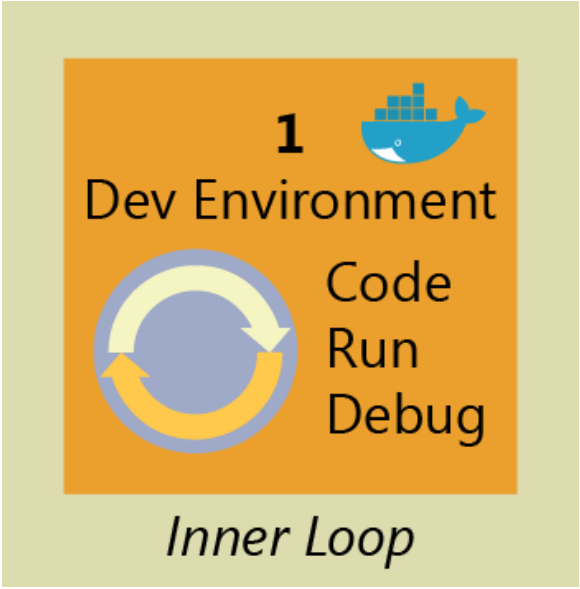


Рисунок 19 – Контекст внутреннего цикла разработки

В каждый контейнер или экземпляр образа Docker входят следующие компоненты:

- выбранная операционная система (например, Windows или дистрибутив Linux);
- файлы, добавленные разработчиком (например, двоичные файлы приложения);
- конфигурация (например, параметры среды и зависимости);
- инструкции касательно того, какие процессы должны выполняться Docker.

Рабочий процесс внутреннего цикла разработки на основе Docker можно настроить как процесс (как описано в следующем разделе). Учтите, что начальные этапы настройки среды здесь не рассматриваются, так как они выполняются только один раз.

Создание одного приложения в контейнере Docker с помощью Visual Studio Code и интерфейса командной строки (CLI) Docker

Приложение состоит из ваших собственных служб и дополнительных библиотек (зависимостей).

На рисунке 20 показаны основные шаги, которые обычно необходимо выполнить при сборке приложения Docker, после чего приводится подробное описание каждого из них.

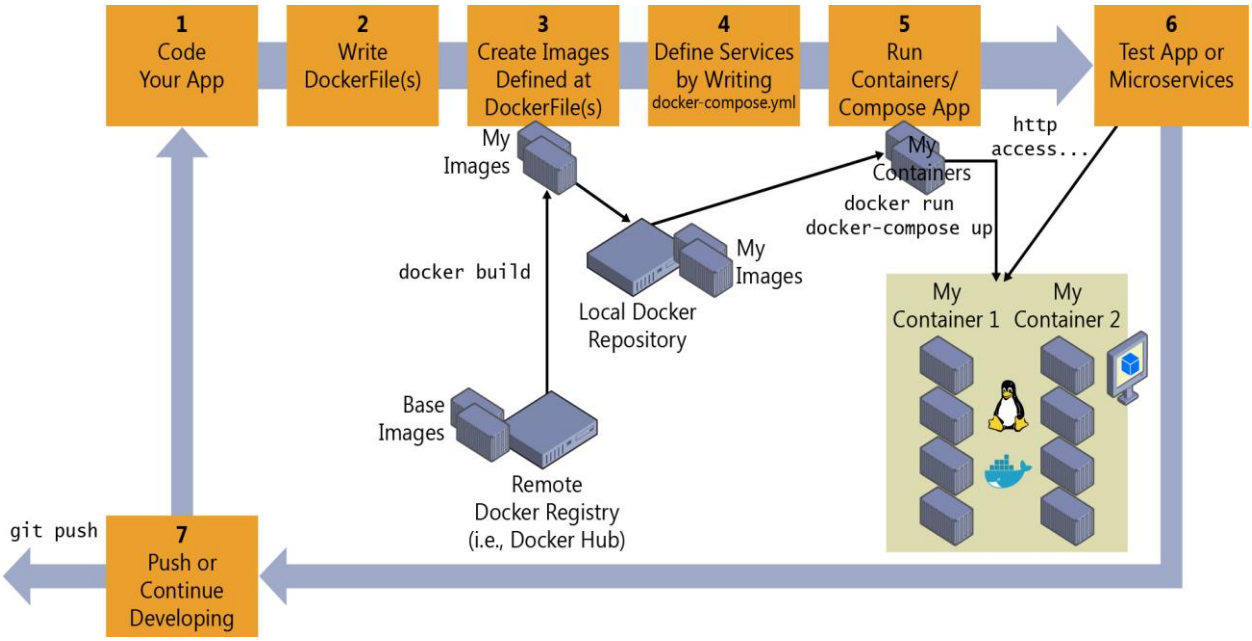


Рисунок 20 – Высокоуровневое представление рабочего процесса для жизненного цикла контейнерных приложений Docker, создаваемых с помощью CLI Docker

Шаг 1. Начало программирования в Visual Studio Code и создание первого приложения или базовой службы

Разработка приложения Docker аналогична разработке приложения без Docker. Разница заключается в том, что при разработке развертывание и тестирование приложения или служб, работающих в

контейнерах Docker, выполняется в локальной среде (например, в Windows или виртуальной машине Linux).

Настройка локальной среды

С помощью последних версий Docker Desktop для Mac и Windows разрабатывать приложения Docker стало еще легче. Настройка очень проста.

Кроме того, вам потребуется редактор кода для разработки приложения с помощью CLI Docker.

Корпорация Майкрософт предлагает Visual Studio Code, простой редактор кода, который поддерживается в macOS, Windows и Linux.

Вы можете использовать CLI Docker и писать код в любом редакторе кода, однако Visual Studio Code в сочетании с расширением Docker упрощает создание файлов `Dockerfile` и `docker-compose.yml`. Вы также можете выполнять задачи и скрипты из интегрированной среды разработки Visual Studio Code для выполнения команд Docker на базе CLI Docker.

Расширение Docker для VS Code предоставляет следующие возможности:

- автоматическое создание файлов `Dockerfile` и `docker-compose.yml`;
- выделение синтаксических конструкций и подсказки при наведении для файлов `docker-compose.yml` и `Dockerfile`;
- IntelliSense (варианты завершения) для файлов `Dockerfile` и `docker-compose.yml`;
- статический анализ кода (ошибки и предупреждения) для файлов `Dockerfile`;
- интеграция палитры команд (F1) с наиболее часто используемыми командами Docker;
- интеграция в обозреватель для управления образами и контейнерами;

- развертывание образов из DockerHub и реестров контейнеров Azure в службе приложений Azure.

Чтобы установить расширение Docker, нажмите клавиши CTRL+SHIFT+P, введите `ext install` и выполните команду "Установить расширение", чтобы открыть список расширений в Marketplace. Затем введите **docker**, чтобы отфильтровать результаты, и выберите расширение поддержки Docker, как показано на рисунок 21.

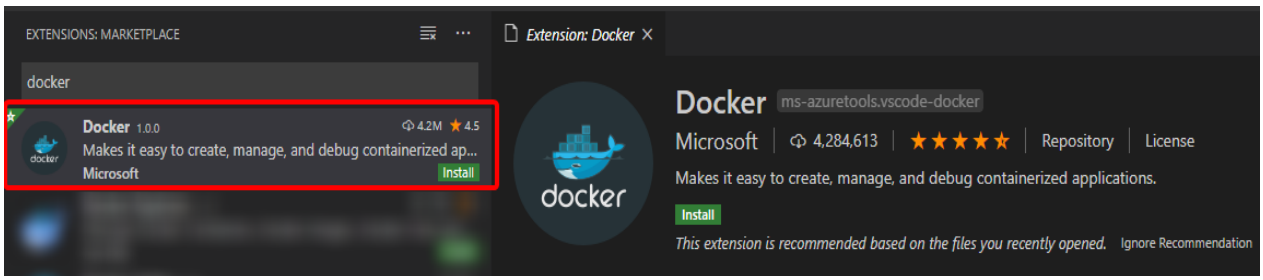


Рисунок 21 – Установка расширения Docker в Visual Studio Code

Шаг 2. Создание файла Dockerfile, связанного с существующим образом (обычная ОС или среды разработки, такие как .NET, Node.js и Ruby)

Для каждого собираемого образа и каждого развертываемого контейнера требуется файл `DockerFile`. Если в приложении имеется только одна пользовательская служба, необходим один файл `DockerFile`. Но если приложение состоит из нескольких служб (как в архитектуре на основе микрослужб), потребуется по одному файлу `Dockerfile` для каждой службы.

Файл `DockerFile` обычно находится в корневой папке приложения или службы и содержит команды, которые требуются Docker для настройки и запуска приложения или службы. Вы можете создать файл `DockerFile` самостоятельно и добавить его в проект вместе с кодом (node.js, .NET и т. д.) или, если у вас нет опыта работы со средой, воспользоваться приведенным ниже советом.

При использовании файлов `Dockerfile` и `docker-compose.yml`, связанных с контейнерами Docker, можно следовать указаниям, которые предоставляются расширением Docker. Вероятно, в дальнейшем вы будете создавать эти файлы, не прибегая к помощи данного средства, но поначалу оно позволяет ускорить обучение.

На рисунке 22 приведены шаги по добавлению файлов Docker в проект с помощью расширения Docker для VS Code.

1. Откройте палитру команд, введите "**docker**" и нажмите "**Добавить файлы Docker в рабочую область**".
2. Выберите платформу приложения (ASP.NET Core)
3. Выберите операционную систему (Linux)
4. При необходимости включите дополнительные файлы Docker Compose
5. Введите порты для публикации (80, 443)
6. Выбор проекта

Рисунок 22 – Добавление файлов Docker с помощью команды Добавить файлы Docker в рабочую область

При добавлении файла Dockerfile указывается базовый образ Docker, который необходимо использовать (например, `FROM mcr.microsoft.com/dotnet/aspnet`). Обычно пользовательский образ создается на основе базового образа, полученного из официального репозитория в реестре Docker Hub (например, образа для .NET или Node.js).

Использование существующего официального образа Docker

Использование официального репозитория стека языка с номером версии гарантирует, что на всех компьютерах (включая компьютеры для разработки, тестирования и работы) будут доступны одни и те же функции языка.

Вот пример файла Dockerfile для контейнера .NET:

DockerfileКопировать

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["src/WebApi/WebApi.csproj", "src/WebApi/"]
RUN dotnet restore "src/WebApi/WebApi.csproj"
COPY . .
WORKDIR "/src/src/WebApi"
RUN dotnet build "WebApi.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "WebApi.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "WebApi.dll"]
```

В этом случае образ основан на версии 6.0 официального образа Docker ASP.NET Core (мультиархитектурного, для Linux и Windows), как

указано в строке `FROM mcr.microsoft.com/dotnet/aspnet:6.0`. (Дополнительные сведения по этой теме см. на страницах Образ Docker ASP.NET Core и Образ Docker .NET.)

Кроме того, в файле Dockerfile можно указать, что средство Docker должно прослушивать порт TCP, который будет использоваться во время выполнения (например, порт 80 или 443).

В Dockerfile можно задать дополнительные параметры конфигурации, в зависимости от используемого языка и платформы. Например, строка `ENTRYPOINT` со значением `["dotnet", "WebMvcApplication.dll"]` указывает Docker запускать приложение .NET. Если для создания и запуска приложения .NET используется пакет SDK и .NET CLI (`dotnet CLI`), этот параметр будет другим. Ключевой момент здесь заключается в том, что строка ENTRYPOINT и другие параметры зависят от языка и платформы, выбранных для приложения.

Использование репозиториев мультиархитектурных образов

В репозитории могут содержаться варианты одного и того же образа для разных платформ, например образ Linux и образ Windows. Это позволяет поставщикам, таким как Майкрософт, которые создают базовые образы, создать один репозиторий для охвата нескольких платформ (т. е. Windows и Linux). Например, репозиторий dotnet/aspnet в реестре Docker Hub обеспечивает поддержку Linux и Windows Nano Server при использовании одного и того же имени образа.

При запросе образа dotnet/aspnet с узла Windows извлекается вариант для Windows, а при запросе образа с тем же именем с узла Linux — вариант для Linux.

Шаг 3. Создание пользовательских образов Docker и внедрение в них собственных служб

Для каждой пользовательской службы в приложении необходимо создать связанный образ. Если приложение состоит из одной службы или веб-приложения, достаточно одного образа.

В рамках "рабочего процесса внешнего цикла DevOps" образы создаются автоматическим процессом сборки при отправке исходного кода в репозиторий Git (непрерывная интеграция), поэтому образы будут создаваться в этой глобальной среде из вашего исходного кода.

Однако перед переходом к этому внешнему циклу необходимо убедиться в том, что приложение Docker действительно работает правильно, чтобы в систему управления версиями (Git и т. д.) не передавался неправильно работающий код.

Поэтому каждый разработчик должен вначале полностью выполнить внутренний цикл, чтобы провести тестирование локально, прежде чем отправлять в систему управления версиями полностью готовую функцию или изменение.

Чтобы создать образ в локальной среде и использовать DockerFile, можно использовать команду docker build, как показано на рисунке 23, так как она сразу помечает образ и создает образы для всех служб в приложении с помощью простой команды.

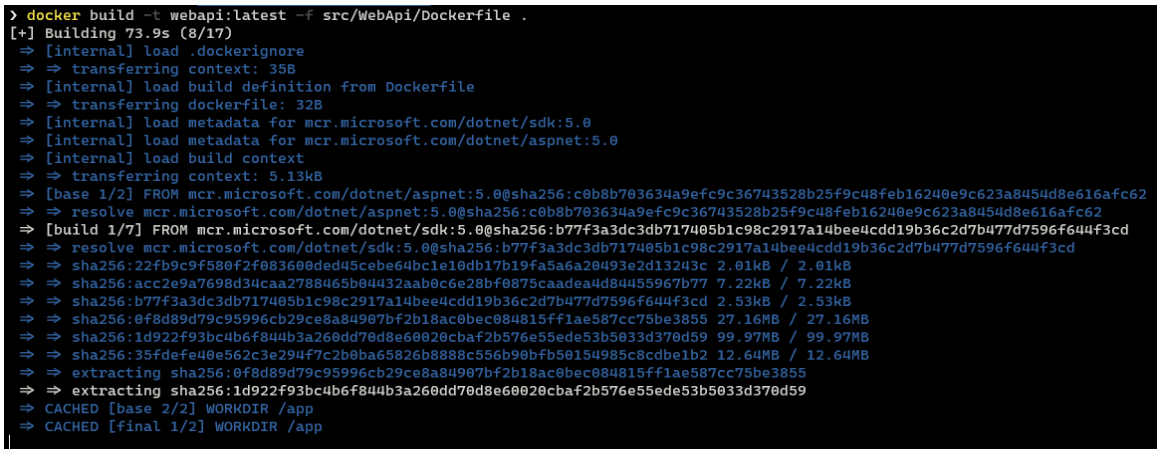


Рисунок 23 – Выполнение команды docker build

При необходимости вместо непосредственного выполнения команды `docker build` из папки проекта можно сначала создать развертываемую папку с нужными библиотеками .NET, выполнив команду `dotnet publish`, а затем использовать команду `docker build`.

В этом примере создается образ Docker с именем `webapi:latest` (`:latest` — это тег, например определенная версия). Этот шаг можно выполнить для каждого пользовательского образа, который требуется создать для составного приложения Docker с несколькими контейнерами. Однако в следующем разделе мы увидим, что это проще сделать с помощью `docker-compose`.

Вы можете найти образы, имеющиеся в локальном репозитории (на компьютере разработки), с помощью команды `docker images`, как показано на рисунке 24.

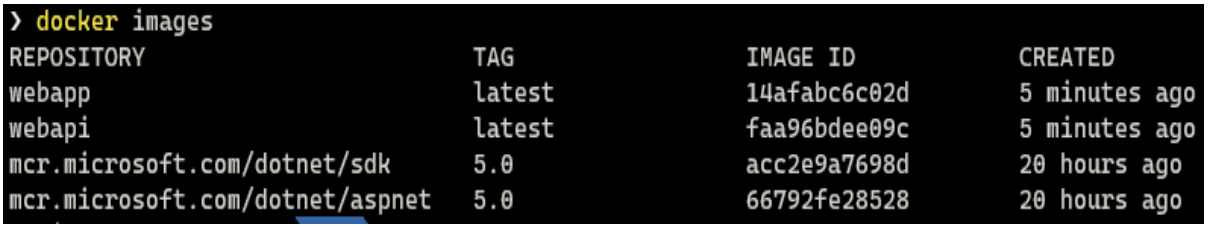


Рисунок 24 – Просмотр существующих образов с помощью команды docker images

Шаг 4. Определение служб в файле docker-compose.yml при сборке составного приложения Docker с несколькими службами

В файле `docker-compose.yml` можно задать ряд связанных служб для развертывания в качестве составного приложения с помощью команд развертывания, описанных в следующем разделе.

Создайте этот файл в основной или корневой папке решения. Его содержимое должно быть аналогично приведенному в следующем файле `docker-compose.yml`:

yml Копировать

```yml
version: "3.4"
```

```yaml
services:
  webapi:
    image: webapi
    build:
      context: .
      dockerfile: src/WebApi/Dockerfile
    ports:
      - 51080:80
    depends_on:
      - redis
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:80

  webapp:
    image: webapp
    build:
      context: .
      dockerfile: src/WebApp/Dockerfile
    ports:
      - 50080:80
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:80
      - WebApiBaseAddress=http://webapi

  redis:
    image: redis
```

В данном случае в этом файле определены три службы: служба веб-API (ваша пользовательская служба), веб-приложение и служба Redis (популярная служба кэширования). Каждая служба развертывается как контейнер, поэтому для каждой из них требуется определенный образ Docker. Для этого конкретного приложения:

- Служба веб-API строится на основе DockerFile в каталоге `src/WebApi/Dockerfile`.
- Порт узла 51080 перенаправляется на доступный порт 80 в контейнере `webapi`.
- Служба веб-API зависит от службы Redis.

- Веб-приложение обращается к службе веб-API, используя внутренний адрес: `http://webapi`.
- Служба Redis использует последний общедоступный образ Redis, извлеченный из реестра Docker Hub. Redis — это популярная система кэширования для серверных приложений.

Шаг 5. Сборка и запуск приложения Docker

Если в приложении имеется только один контейнер, его можно запустить путем развертывания на узле Docker (в виртуальной машине или на физическом сервере). Однако если приложение состоит из нескольких служб, его также необходимо *скомпоновать*. Рассмотрим разные варианты.

Вариант А. Запуск одного контейнера или службы

Образ Docker можно запустить с помощью команды docker run, как показано в этом примере:

КонсольКопировать

```
docker run -t -d -p 50080:80 webapp:latest
```

Для этого конкретного развертывания мы будем перенаправлять запросы, отправленные на порт 50080 узла, на внутренний порт 80.

Вариант Б. Компоновка и запуск многоконтейнерного приложения

В большинстве корпоративных сценариев приложение Docker будет состоять из нескольких служб. В таких случаях можно выполнить команду `docker-compose up` (рисунок 25), которая использует ранее созданный файл docker-compose.yml. В результате ее выполнения выполняется сборка всех пользовательских образов и развертывается скомпонованное приложение со всеми связанными контейнерами.

```
> docker-compose up
Creating explore-docker-vscode_webapi_1  ... done
Creating explore-docker-vscode_webapp_1  ... done
Attaching to explore-docker-vscode_webapp_1, explore-docker-vscode_webapi_1
webapp_1  | warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]
webapp_1  |       Storing keys in a directory '/root/.aspnet/DataProtection-Keys' that may not be p
ected data will be unavailable when container is destroyed.
webapp_1  | warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
webapp_1  |       No XML encryptor configured. Key {19eedc7c-d3d0-4443-af35-93d4d078119c} may be pe

webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Now listening on: http://[::]:80
webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Application started. Press Ctrl+C to shut down.
webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Hosting environment: Development
webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Content root path: /app
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Now listening on: http://[::]:80
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Application started. Press Ctrl+C to shut down.
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Hosting environment: Development
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Content root path: /app
```

Рисунок 25 – Результаты выполнения команды docker-compose up

После выполнения команды `docker-compose up` приложение и связанные с ним контейнеры развертываются в узле Docker, как показано в представлении виртуальной машины на рисунке 26.

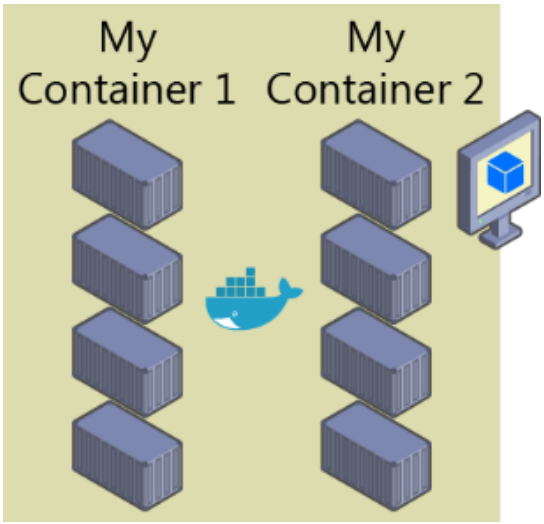


Рисунок 26 – Виртуальная машина с развернутыми контейнерами Docker

Шаг 6. Тестирование приложения Docker (в локальной виртуальной машине для непрерывного развертывания)

Этот шаг будет зависеть от того, что делает ваше приложение.

В случае простого веб-API Hello World на основе .NET, развернутого в виде единственного контейнера или службы, для доступа к службе достаточно указать TCP-порт из файла Dockerfile.

В узле Docker откройте браузер и перейдите на этот сайт. Вы должны увидеть работающее приложение или службу, как показано на рисунке 27.

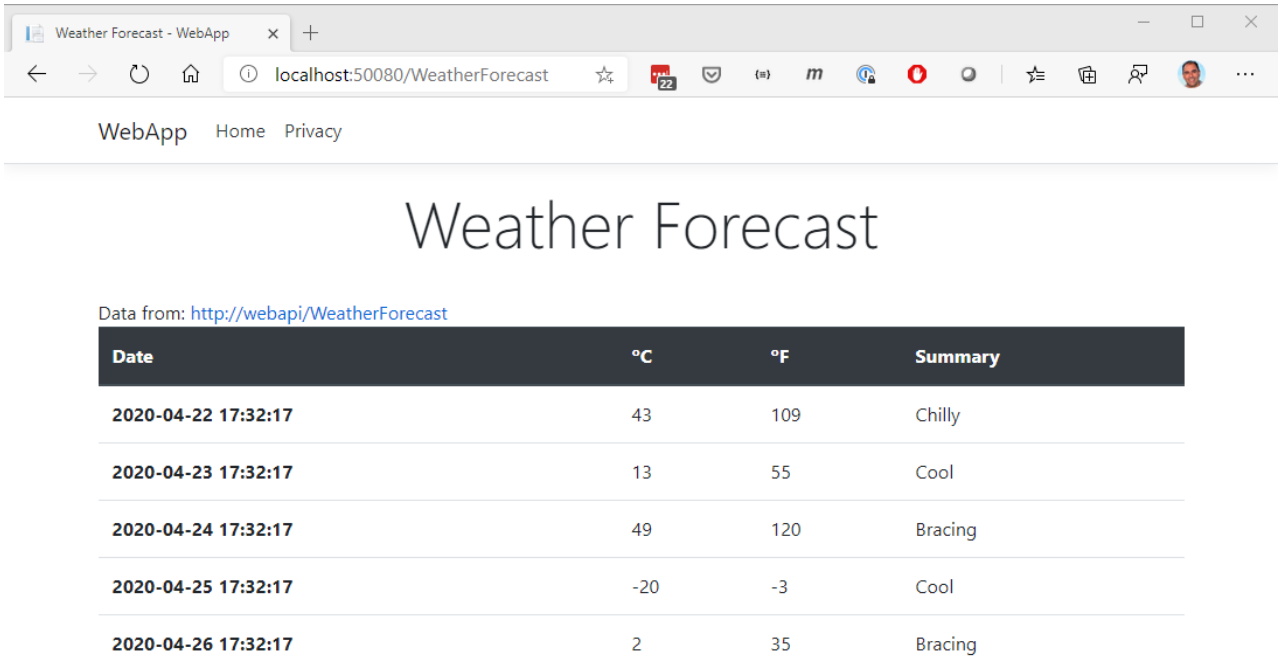


Рисунок 27 – Локальное тестирование приложения Docker с помощью браузера

Обратите внимание на то, что используется порт 50080, однако внутренние запросы перенаправляются на порт 80, поскольку именно так было выполнено развертывание с помощью команды `docker compose`, как это описывалось ранее.

Это можно проверить с помощью браузера, используя cURL из терминала, как показано на рисунок 28.

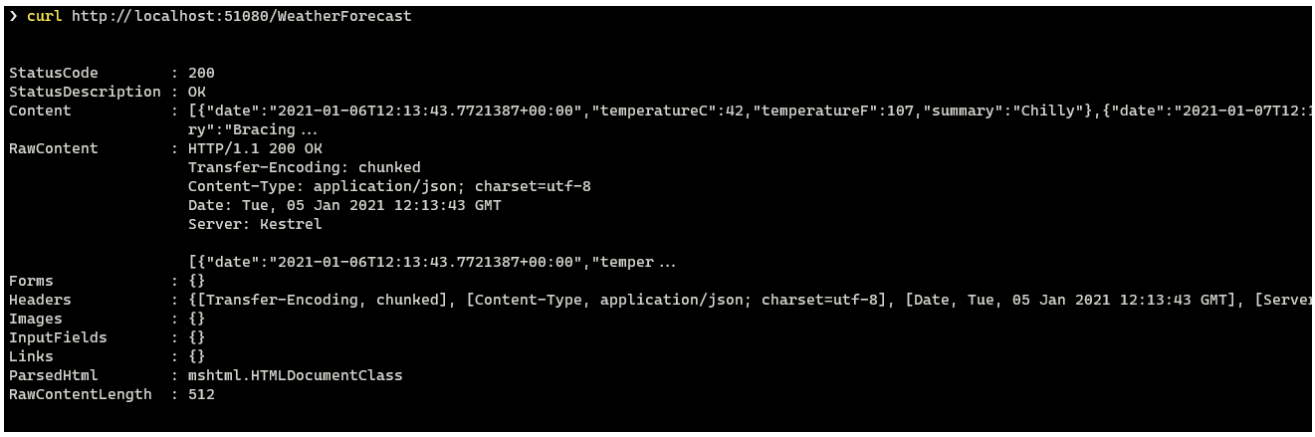


Рисунок 28 – Локальное тестирование приложения Docker с помощью CURL

Отладка контейнера, запущенного в Docker

Visual Studio Code поддерживает отладку Docker при использовании Node.js и других платформ, таких как контейнеры .NET.

Запуск, мониторинг рабочих сред Docker и управление ими

Концепция. Корпоративные приложения должны выполняться на высоком уровне доступности и масштабируемости, ИТ-операции должны контролировать и отслеживать среды и сами приложения.

Этот последний принцип жизненного цикла контейнерных приложений Docker касается методов выполнения и мониторинга приложений, а также управления ими в масштабируемых и высокодоступных производственных средах.

Запуск и выполнение контейнерных приложений в рабочей среде (в архитектуре инфраструктуры и с технологиями платформы) в значительной степени связаны и основаны на выбранной архитектуре и платформах разработки.

Запуск составных и основанных на микрослужбах приложений в рабочих средах

Приложения, состоящие из нескольких микрослужб, нужно развертывать в кластерах оркестратора, чтобы упростить развертывание и сделать его возможным с учетом доступных ИТ-ресурсов. Без кластера оркестратора было бы трудно развернуть и масштабировать сложное приложение микрослужб.

Введение в оркестраторы, планировщики и кластеры контейнеров

Ранее в этой электронной книге *кластеры* и *планировщики* упоминались в рамках обсуждения архитектуры и разработки программного обеспечения. Kubernetes и Service Fabric являются примерами кластеров Docker. Оба эти оркестратора могут выполняться в составе инфраструктуры Службы Azure Kubernetes.

Когда приложения масштабируются по нескольким системам узла, привлекательной становится возможность управления каждой системой узла и абстрагирования от сложности базовой платформы. Именно за это и отвечают оркестраторы и планировщики. Давайте кратко рассмотрим их:

- **Планировщики**. "Планирование" обозначает способность администратора загрузить на систему узла файл службы, который описывает выполнение конкретного контейнера. Запуск контейнеров в кластере Docker обычно называется планированием. Хотя планирование обозначает конкретную процедуру загрузки определения службы, в более общем смысле планировщики отвечают за подключение к системе

инициализации узла для управления службами с любой необходимой емкостью.

Планировщик кластера преследует несколько задач: эффективное использование ресурсов кластера, работа с заданными пользователем ограничениями размещения, быстрое планирование приложений, чтобы они не оставались в состоянии ожидания, а также обеспечение определенной степени равнодоступности и устойчивости к ошибкам постоянной доступности.

- **Оркестраторы**. Платформы расширяют возможности управления жизненным циклом до сложных многоконтейнерных рабочих нагрузок, развернутых в кластере узлов. Используя абстрагирование инфраструктуры узлов, инструменты оркестрации позволяют пользователям обрабатывать весь кластер как единый целевой объект развертывания.

  Процесс оркестрации включает в себя средства и платформы, позволяющие автоматизировать все аспекты управления приложениями — первоначальное размещение или развертывания для каждого контейнера, перемещение контейнеров на разные узлы в зависимости от состояния или работоспособности узла, а также управление версиями, развертывание обновлений и функции мониторинга работоспособности, которые поддерживают масштабирование и отработку отказа, и многое другое.

  Оркестрация — это общий термин, который охватывает планирование контейнеров, управление кластерами и, возможно, подготовку дополнительных узлов.

Возможности, предоставляемые оркестраторами и планировщиками, сложно разработать и создать с нуля, поэтому обычно рекомендуется использовать решения оркестрации, предлагаемые поставщиками.

Краткие итоги

- **Решения на основе контейнеров** предоставляют важные преимущества с точки зрения сокращения расходов, так как контейнеры позволяют устранять проблемы, вызванные отсутствием зависимостей в рабочих средах, таким образом значительно повышая эффективность рабочих операций DevOps.
- **Docker** стал стандартом де-факто в области контейнеризации приложений и поддерживается большинством крупных поставщиков в экосистемах Windows и Linux, включая Майкрософт. В будущем Docker можно будет встретить в любом центре обработки данных, как в облаке, так и в локальной среде.
- **Контейнер Docker** превращается в стандартную единицу развертывания любого серверного приложения или службы.
- **Оркестраторы Docker**, как, например, представленные в службе Azure Kubernetes (AKS) и Azure Service Fabric, являются основными и необходимыми компонентами для любого приложения на основе микрослужб или многоконтейнерного приложения с высокими требованиями к сложности и масштабируемости.
- **Среда DevOps**, поддерживающая взаимодействие непрерывной интеграции и непрерывного развертывания (CI/CD) с рабочими средами Docker, обеспечивает гибкость и в итоге сокращает время выхода приложений на рынок.
- **Azure DevOps Services** значительно упрощают функционирование среды DevOps с помощью развертывания в средах Docker из конвейеров CI/CD. Это заявление применяется к простым средам Docker, а также сложным оркестраторам микрослужб и контейнеров на основе Azure.