Добрый день. И добро пожаловать на очередную лекцию курса Golang для инженеров, посвященную взаимодействию Docker-а и Golang-а, в которой мы рассмотрим: Docker-образы для разработчиков, как их собирать, и как они используются в production-е; как работает, вообще, приложение на Golang в production-е; посмотрим на Docker CDK и его задачи, применительно к Golang-у, что мы можем сделать, зачем он нам нужен; и я дам несколько практических советов по, соответственно, применению Docker-а CDK и Docker-контейнера в вашей повседневной работе. Погнали.

И первая тема, с которой мы столкнемся, уж коли речь заходит про Docker, это Docker-образы для разработчиков. Если наша компания переезжает на Golang или наша инфраструктурная команда переезжает на Golang, соответственно, встает образ Docker-контейнеров. Как делать Docker-контейнер для Golanga, в чем специфика и так далее.

В рамках этого курса лекций я не рассказываю, что такое Docker. Такая краткая (НРЗ 01:06). Docker – это система контейнеризации, то есть у вас есть операционная система, которая внутри себя запускает какие-то библиотеки, там есть какие-то служебные программы, какое-то, в общем, окружение. И что Docker делает? Он изолирует окружение в отдельно-взятом контейнере, как бы делает тонкую прослойку систем по (НРЗ 01:25). То есть отдельно окружение, в нем свои библиотеки, свой shell. Это все работает отдельно от главной системы, но не создает свой как бы пул ресурсов, в отличие от систем виртуализации. То есть там не надо эмулировать аппаратный слой, с аппаратным слоем Docker работает напрямую.

И вот эти самые контейнеры, они все содержат только нужные библиотечки, нужную инфраструктуру, нужное как бы окружение, чтобы ваша программа работала. Если вы делали контейнеры для «Питона» или PHP, то вы, наверное,

замечали, что там нужно установить интерпретатор, нужно накатить все библиотеки, нужно собрать как бы все это окружение. И контейнер получается большой. Хотя Docker по своей сути постулирует то, что контейнер должен быть настолько маленьким, насколько возможно. Но, естественно, с интерпретируемыми языками совсем маленьких контейнеров мы позволить себе не можем.

Ну, и соответственно, Golang, приложение на Golang нам позволяет уменьшить размер контейнера, не держа все пакеты на целевой машине. Нам нужны только системные зависимости. То есть если, например, наше приложение внутри себя вызывает LS, нам нужно установленное LS на машине. Если приложение внутри себя взаимодействует с интернетом, нам нужно, чтобы был какой-то сокет в системе и так далее.

Соответственно, как мы решаем эту проблему с Golang-ом, мы используем два Docker-образа. Мы используем образ build, в который установят все библиотечки Golang-а, в котором будет компилятор Golang-а. Мы соберем с помощью него как раз системное приложение под нужную систему. И вторым контейнером мы будем это приложение уже деплоить. Deploy, соответственно, будет содержать только системное окружение, минимальный набор, соответственно, утилей для того, чтобы наше приложение работало. И будет запускать, так называемые, артефакты build-а. То есть те самые исполняемые приложения, которые мы собрали. Давайте посмотрим на практике, как это работает.

Ну и давайте разберемся с нашим Docker-ом на примере простого приложения. Я взял framework. Framework называется echo. В принципе, это обычная оболочка на (НРЗ 03:42) http, значит, сервера, которая там имеет несколько дополнительных фишек, типа там всяких TLS-ов и прочих свистелок. Фишка в том, что она позволяет не определять отдельно handler, а просто функции прям сразу в

сервере. И за счет чего, ну довольно-таки просто увидеть все в одной картине. То есть мы здесь видим, что у нас здесь 2 сетевых ресурса, значит, «/» и «ping». И httpPort, на котором может работать наше приложение, по умолчанию 8080. Tj tcnm это наш http-сервер, который имеет две функции – не работать ни с какими базами данных и, соответственно, ну в собранном виде будет занимать совсем немножко, с учетом того, что библиотека тоже там не занимает какое-то огромное количество байт.

Значит, давайте посмотрим, какой контейнер я использую для сборки. Это наш значит Dockerfile. Для тех, кто плохо знаком с синтаксисом. Значит, FROM – это у нас получить какой-то уже определенный контейнер из Docker-a, который мы в дальнейшем можем использовать для сборки наших собственных контейнеров. В данном случае golang:1.16-alpine, значит. Alpine вообще значит, что взят самый минимальный build, на него сверху накачан Golang, и там больше нет ничего. То есть там, например, есть в Docker-е сборке там ubuntu и debian. В них там есть уже какие-то системные библиотеки ubuntu-овские или debian-овские. И здесь нет ничего кроме системных нужных библиотек и ядра. И соответственно, Go, установленного со всеми его зависимостями.

Соответственно, определяем рабочую директорию, копируем наши, значит, go.sum, go.mod, запускаем go mod download. Значит, копируем наши go-файлы main.go. То есть go sum download скачает все зависимости сервера, скопируем, значит, наш go. И строим server, компилируем server.

После компиляции, соответственно, запускаем сам контейнер. Давайте запустим build этого. Я буду использовать команду Docker build. Значит, в команде Docker build, поскольку у меня Docker файлы размещены в директориях, я буду указывать директорию через команду файлик. И укажу ему тэг mycontainer. Или давайте

укажем тэг myserver. Building, загружаем, поскольку я уже проходил этот процесс build-а несколько раз, у меня все уже готово.

В дальнейшем я буду использовать, значит, (НРЗ 06:44), есть такая (НРЗ 06:46). Десктопное приложение, которое позволяет вам удобно посмотреть ваши контейнеры, ваши Image-и, ваши, соответственно, подсоединенные какие-то диски сетевые. И позволяет вам создавать какой-то код, который будет потом запускаться в Docker-контейнерах.

Соответственно, смотрим на myserver, на контейнер, смотрим его Image id. Смотрим, когда он был создан. Это время первого создания контейнера. И видим его размер. 400 мегабайт, как бы ну казалось бы да фигня какая-то? Но, во-первых, для приложения на Golang-е, которое занимает 2,7 мегабайт, само по себе, ну то есть стандартный размер упаковки приложения, 400 мегабайт сверху – это довольно-таки много. Это, во-первых. А во-вторых, ну если у вас таких вот приложений будет на сервере, скажем, 20, то здесь уже у вас будет 8 гигабайт памяти, которую вы просто отдадите в никуда, на то, чтобы хранить там все зависимости, которые подтянулись, хранить компилятор Golang-а и исходные файлы, которые вам тоже в самом контейнере не нужны.

Давайте посмотрим, как можно оптимизировать размер этого контейнера. Значит, как я уже говорил в лекции, нам нужно использовать 2 образа. Один образ у нас, значит, будет build-контейнером, другой образ у нас будет run-контейнером. Соответственно, мы используем, мы берем тот же самый golang:1.16-alpine, помечаем его – как build. Можно это пропустить, эту фазу, вполне спокойно, я потом поясню, для чего она нужна. Уже, в принципе, по подсветке видно, для чего она нужна. Уже, в принципе, по подсветке видно, для чего она нужна. Ну и дальше все те же самые команды, да, там копируем наши go-модули, их скачиваем, копируем go-файлы и build-им наш сервак.

Затем собираем, так называемый, deploy-контейнер. То есть здесь мы берем какой-то образ, в данном случае я беру образ debian10 – базовый, он небольшой. Устанавливаем рабочую директорию, копируем наши образы. Вот в этот from, это как раз будет указание на тот контейнер, где мы собирали. Потому что так-то Docker нам позволяет в одном Docker-файле указать сколько хочешь контейнеров. Если мы, например собираем одно (НРЗ 09:04) приложение, у которого есть backend, frontend – это все надо скомпилировать, положить в какую-то папочку на сервер.

У нас может быть 3 контейнера. Одно собирает backend, одно – frontend. И потом одно финальное приложение (НРЗ 09:15). Там есть (НРЗ 09:17), и оно копирует все файлы из обоих контейнеров, build-контейнеров. И нам нужно их как-то пометить. Если мы уберем вот этот build, то здесь у нас будут номера. То есть вот этот контейнер будет нулевой, следующий за ним будет первый и так далее. Я нахожу это неудобным, нахожу плохим стилем, когда вам приходится догадываться из какого ж контейнера сейчас потечет. В общем, здесь надо указывать конкретные тэги.

Значит, мы берем этот debian, копируем из нашего build-контейнера. То есть в этот момент у нас существует два контейнера. Копируем файлик server в server, в корень, показываем порт, устанавливаем nonroot пользователя, чтобы ничего у нас там не поломалось, и никаких дырок не было, и просто его запускаем. То есть здесь у нас нет никаких библиотек, никакого компилятора Golang-а. Есть вот это вот готовое собранное приложение по debian, такой исполняемый файлик. И, собственно говоря, все.

Давайте соберем. Я также его протэгирую. Здесь вон видно, что у нас есть Metadata. Потом у нас, значит, build, потом у нас все вот это вот копируется. И у нас просто выполняется одной командой, копия, экспортируется в свои. И

запаковывается это все в какой-то готовый артефакт, который уже не содержит в себе build образа.

И можно посмотреть, опять же, списки Image-ей. Здесь я накосячил. В смысле, поскольку тэг одинаковый, у меня теперь здесь no name контейнер. Тот наш на 400 мегабайт, а вот новый контейнер у нас весит 27 мегабайт, что гораздо более приятно, и позволит нам тратить гораздо меньше системных ресурсов и оперативной памяти на поднятие контейнера. Вот это наш сервер уже упакованный и готовый к работе.

Иногда по каким-то причинам мы не хотим давать разработчикам контроль над Docker-файлами. Ну например, у нас есть какие-то специфические настройки для наших контейнеров, у нас есть какой-то специфический свой секьюрити, который выгружается в контейнеры. И мы хотим это все сделать на стороне нашего pipeline-а. То есть то же самое, что я сейчас проверну, вот этот двойной Docker образ, мы хотим сделать на стороне нашего pipeline-а.

В чем проблема Docker-а? В том, что единственное, что поддерживает сам по себе Docker-контейнер – это, это вгружение переменных из (НРЗ 11:53). То есть вы можете через $ и фигурные скобки писать всякие окружения переменных, точно также, как и в bash-скриптах, например. Они будут вгружаться в Docker-файле на вашем runner-е. И там выполнятся.

И вам придется настраивать runner таким образом, чтобы у вас все переменные там были. Надо будет этот Docker-файлик где-то держать, где-то подтягивать. Это не очень, на самом деле, удобно, когда у вас есть какой-то pipeline. А особенно, если у вас есть несколько разных golang приложений, у которых разные, например, версии. И вам нужно контейнеры пересобирать каждый раз.

Build-контейнеры, production -контейнеры вам нужно каждый раз подбирать заново.

Поэтому, когда у вас есть задача отделить разработку от build-а, то есть вы build делаете сами на стороне DevOps-а, разработчик делает только разработку, и он только определяет финальную версию контейнера, который у него есть, то есть production -контейнер. Тогда вам нужно построить pipeline в вашей системе CI/CD. Соответственно, в Jenkins-е, в GitLab CI, в Github actions, где угодно. В принципе, работает она одинаково. После commit-а кода, происходит build артефакта. Как раз того самого бинарного уже исходничка в нужной версии, под нужную систему. Происходит Deploy артефакт в production-контейнер, его упаковка. И production-контейнер определяется разработчиком, например. Может вами определятся. И потом происходит запуск в production-е.

В чем разница? Разница в том, что в первом случае я показал, как это делается чисто средствами (НРЗ 13:34). Сейчас я покажу, как это делается средствами CI/CD pipeline так, чтобы у вас на выходе получался только упакованный контейнер, а разработчик только определял Docker-файл уже с финальным build-ом. И не трогал те моменты, где сам build, непосредственно, происходит. Давайте посмотрим на это.

Итак, мы находимся в GitLab. Для своей ежедневной деятельности, я использую GitLab. Соответственно, на Github actions, на CircleCI, там Bamboo, Jenkins, что угодно – будет выглядеть примерно то же самое, но концепт один и тот ж.

Итак, смотрите, у меня то же самое приложение мое, вот это вот – с моим «Hello, Docker-ом», с веб-сервером. И у меня для него есть Docker-файл, можно увидеть, что он выглядит абсолютно иначе. То есть у меня здесь только ENTRYPOINT. Значит я вгружаю артефакт от куда-то снаружи. Я, как разработчик, только здесь

меня интересует, по сути, моя система, и я могу какие-то библиотечке поставить. Это все, над чем я имею контроль. Build происходит в каком-то другом месте. Build у нас происходит в GitLab CI. Я создал шаблончик CI, и здесь у меня несколько стадий. То есть у меня есть стадия, по сути, build и стадия Deploy. Здесь я не показываю, как загружать приложение, например в артефакторе или в какой-то репозиторий. Здесь нас больше будет интересовать, как вот то же самое, что у нас происходит с Docker-файлом, повторить средствами CI/CD.

Ну, по сути, здесь все очень просто. Можно увидеть, так сказать, последствия моего debug-а. Здесь я вывожу просто, что есть папочки. Здесь у нас просто запускается go build. Потому что go module mod запустится при build-е этого сервера. То есть GitLab устроен как? GitLab работает на Docker-е. То есть runner-ы GitLab-а.., вернее, они могут работать на Docker-е, это обычный стандартный режим. И вы можете указать runner-ом default-ный Image. То есть вы можете сказать, что вот мой runner сейчас работает на конкретном Image-е golang. И там уже есть у меня, значит, компилятор, есть у меня все тузы, которые Go нужны.

И дальше я указываю перемены. Эти перемены будут вгружаться внутри Docker-контейнер, как переменные окружения, поэтому я их могу использовать из Dockerfile. Вот, как раз вот таким образом. Да, вот это у меня переменная окружения. И их я могу использовать также в скриптах.

Значит, теперь у нас запустился этот Image. И допускается первая стадия. В первой стадии я собираю бинарник и (НРЗ 16:28), так сказать, то есть заливаю артефакт. Потому что после каждой стадии контейнеры сбрасываются, контейнеры запускаются заново, и собранный артефакт мне надо где-то хранить. А храню я его по пути в binaries после сборки. И значит на стадии уже Deploy-ина, второй стадии... Ну, так называемого – Deploy. То есть, по сути, никакого Deploy нету. Я указываю dependencies для того, чтобы собрать артефакты вот от сюда. И

после этого, я собираю Docker. Да, у меня Docker в Docker-е, тут можно увидеть. Я использую, отдельный Image для моей работы, который в себе содержит, непосредственно, Docker SDK, и запускает Docker в Docker-е. Напоминает этот мем, что он вам, значит, поставили больше экранов в экраны, чтобы вы смогли смотреть в экраны, когда смотрите в экраны – здесь то же самое.

Значит, собираю я здесь Docker Image через тот Docker-файл, который там указан. И можно заметить, что в моем Docker-файле у меня копируется просто артефакт в корень сервер. И после этого, соответственно я сохраняю этот контейнер, покую в виде tar архива. То есть контейнеры, которые у вас собраны, Image-и, вы можете упаковать и потом положить, например, в репозиторий. И на какой-то стороне, где есть другой Docker, его вгрузить с этого Image-а. Так работают, в принципе, практически все репозитории Docker-ские.

Ну и значит, потом я просто его кладу в артефакты. Мог бы вместо того, чтобы положить в артефакты, например, залить вот этот самый файлик в артефакторе с какой-то метаинформацией – это не важно в данном моменте. То есть то же самое, что мы делали, значит, здесь – в build и Deploy-е. Разделили на два участка. Вот эта часть у меня теперь выполняется внутри GitLab-а. Кажется, что это очень многословно и не очень нужно. Но, если вы хотите полностью контролировать процесс build-а, процесс того, что у вас происходит на build части, либо, наоборот, разработчики представляют только build-контейнеры, в этом build-контейнере вы можете запустить ваш build и получить артефакт, потом его упаковать. Ну то есть концепция должна быть ясна.

И здесь мы, соответственно, получаем после работы CI/CD, можно посмотреть на build, на compile – там ничего интересного. У нас там всегда качаются библиотечки. Потом смотрим, что у нас получилось. В итоге получилась папочка binaries. Мы

загружаем в артефакты. И после этого, на этапе упаковки мы видим, что здесь собирается контейнер, получаются Image-и и загружаются артефакты.

Артефакты можно скачать отсюда. То есть у нас тут есть результат работы pack, результат работы compile. В одно из них – у нас лежит наше приложение, в другом – запакованный Image этого самого архива, запакованный архив этого самого Image-а на 27 мегабайт, который мы можем потом куда-то сохранить и использовать, например, в нашем production pipeline -е.

Ну и напомню некоторые общие положения по работе контейнера именно в production окружении, которые характерны не только для Golang-а, но и в целом для, вообще, инфраструктуры.

Соответственно, первое, как правило, контейнеры ограничены ресурсами сверху, ограничены (НРЗ 19:57), ограничена память, потому что система, опять же, не резиновая. Ограничивать контейнеры важно для того, чтобы ваша система понимала сколько там максимум вообще может быть. Опять же, единственный, не единственный, но наиболее приятный и удобный способ работать с контейнером – это загружать environment variables сверху. То есть, как правило, система берет на себя менеджмент переменных окружения. И загружает эти переменные окружения в контейнеры. Это важно понимать, когда ты разрабатываешь свое приложение. То есть можно просто флагами, можно вгружать файлы конфигурации. Иногда без этого никак, но, как правило, стандартным способом является вгрузка переменных окружения. То же самое работает для всяких конфигурационных инфраструктурных приложений.

Если ваше приложение, куда-то сохраняет данные, убедитесь, что оно сохраняет их в монтированную файловую систему. Почему? Потому что контейнеры по своей природе эфемерны, мы про это еще сейчас поговорим, и они могут быть удалены.

В этом случае файловая система удаляется тоже, она не сохраняется внутри контейнера. Поэтому существует несколько способов. К контейнеру привязать, соответственно, так называемый, volume. Куда-то смонтировать ее, в какую-то папку. И убедитесь, что ваше приложение пишет именно туда.

У контейнеров существует, какой-то определенный набор метрик, которые вы можете собирать и использовать: память, процессор, соответственно, входной-выходной трафик. Можете сделать свои метрики, которые будут сверху собираться то же. И в общем-то это ваш основной способ следить за, так называемым, performance-ом, за здоровьем приложения, за тем, как оно, соответственно, работает. И сколько ресурсов оно потребляет.

Нету отладочных инструментов. То есть ну как вы привыкли, локально там запускаешь, открываешь все порты и какой-то трафик роутишь, какие-то профайлеры подключаешь, что-то смотришь. В Docker можно прокинуть отладочные инструменты локально. Естественно, в production-е у вас так не получится, потому что открывать отладочные порты – это огромная, вот такая вот дырка в безопасности. Ну как минимум…, и как максимум, это еще и замедляет работу самого контейнера, потому что он перед тем, как, например, вызвать какие-то функции, должен сохранить отладочную информацию, поставить какие-то символы. Ну в общем, никто этим заниматься, естественно, не будет. Поэтому для разработчиков это поначалу непривычно. Особенно для тех, кто не работал никогда с облачными решениями. И для таких, так называемых, олдскульных разработчиков, которые еще привыкли заливать все в систему, и в системе какую-то еще отладку проводить, какие-то там профилировщики на ходу подключать. В общем, этого всего у вас в контейнерах не будет, скорее всего.

Ну и про эфемерность я уже сказал. Ваше приложение должно стабильно работать, если оно вдруг, по какой-то причине было перезапущено. Поэтому

никакие мы данные в памяти стараемся не хранить. Неважные данные мы, соответственно…, мы должны быть готовы, что мы ее сейчас потеряем. Приложение должно быть готово, что в какой-то момент оно просто может взять и выключится, потому что система, например, ресурсов себе забрала, чтоб какую-то операцию сделать. Или система перераспределила контейнеры там на другую виртуальную машину или другое ядро. Ну в общем, там много есть вариантов, почему Docker может ваше приложение убить. Вернее, не Docker, а система над Docker-ом, Docker swarm, Kubernetes, Openshift и какие-то прочие системы. Соответственно, будьте готовы. То же самое касается инфраструктурных приложений, они могут быть убиты. И соответственно, старайтесь это учитывать.