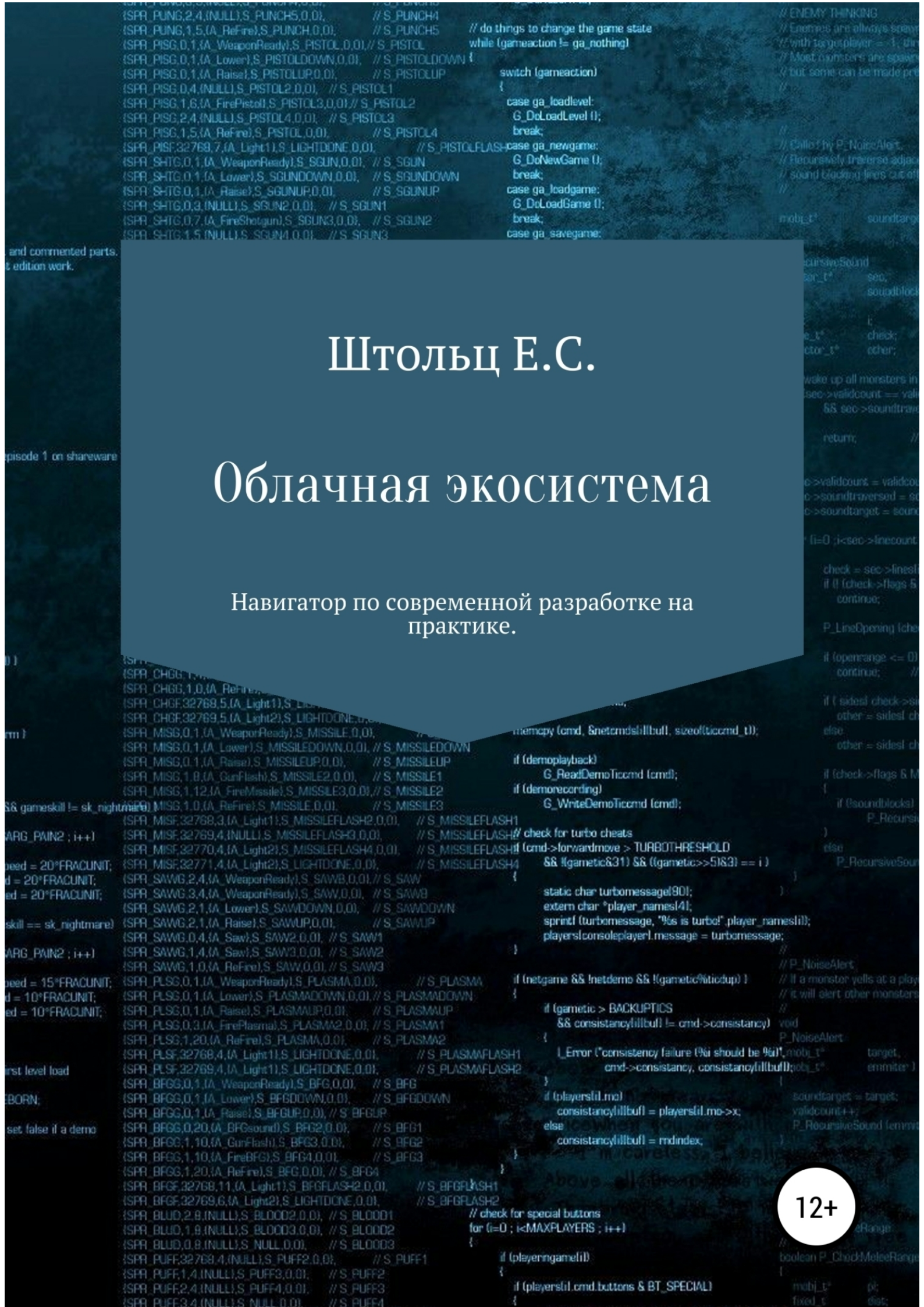

Штольц Е.С.

Облачная экосистема

Навигатор по современной разработке на практике.

12+

Евгений Штольц

Облачная экосистема

«ЛитРес: Черновики»
2020

Штольц Е. С.

Облачная экосистема / Е. С. Штольц — «ЛитРес: Черновики», 2020

В этой книге главный Архитектор Департамента Архитектуры компетенций Cloud Native в Сбербанк делится знанием и опытом с читателем созданием и перехода на облачную экосистему, так и созданием и адаптацией приложений под неё. В книге автор старается провести читателя по пути, минуя ошибки и сложности. Для этого демонстрируется практическое применение и даются пояснения, чтобы читатель смог ими воспользоваться как инструкцией для учебных и рабочих целей. Читателем может быть как разработчики разных уровней, так и специалисты по экосистеме, желающие не потерять актуальность своих умений в уже изменившимся мире.

Евгений Штольц
Облачная экосистема

Аннотация

В книге на практике рассматриваются более 70 (76) инструментов:

* платформы Google Cloud Platform, Amazone WEB Services, Microsoft Azure;

* консольные утилиты: cat, sed, NPM, node, exit, curl, kill, Dockerd, ps, sudo, grep, git, cd, mkdir, rm, rmdir, mongos, Python, df, eval, ip, mongo, netstat, oc, pgrep, ping, pip, pstree, systemctl, top, uname, VirtualBox, which, sleep, wget, tar, unzip, ls, virsh, egrep, cp, mv, chmod, ifconfig, kvm, minishift;

* стандартные инструменты: NGINX, MinIO, HAProxy, Docker, Consul, Vagrant, Ansible, kvm;

* инструменты DevOps: Jenkins, GitLab CI, BASH, PHP, Micro Kubernetes, kubectl, Velero, Helm, "http нагрузочное тестирование";

* облачные Traefic, Kubernetes, Envoy, Istio, OpenShift, OKD, Rancher,;

* несколько языков программирования: PHP, NodeJS, Python, Golang.

Контейнеризация

История развития инфраструктуры

Limoncelli (автор "The Practice of Cloud System Administration"), работавший долгое в Google Inc, считает, что 2010 год – год перехода от эры традиционного интернет к эре облачных вычислений.

* 1985-1994 – время мэйнфреймов (больших компьютеров) и внутрикорпоративного обмена данных, при котором

можно легко планировать нагрузку

* 1995-2000 – эра появления интернет-компаний,

* 2000-2003

* 2003-2010

* 2010-2019

Увеличение производительно отдельно машины меньше, чем прирост стоимости, например, увеличении производительности в 2 раза приводит к увеличению стоимости существенно большей, чем в 2 раза. При этом, каждое следующее увеличении производительности обходится кратно дороже. Следовательно, каждый новый пользователь обходился дороже.

Позже, в период 2000-2003 годах, смогла сформироваться экосистема, предоставляющая принципиально другой подход:

* появление распределённых вычислений;

* появление маломощных массовой аппаратуры;

* созревание OpenSource решений, позволяющие устанавливать программное обеспечение на множество машин, не связанное связкой лицензия процессор;

* созревание телекоммуникационной инфраструктуры;

* увеличении надёжности за счёт распределения точек отказов;

* возможность наращивания производительности при потребности в будущем за счёт добавления новых компонентов.

Следующим этапом слала унификация, наибольшее проявлявшаяся в 2003—2010 годах:

* предоставление в дата центре не места в шкафу (power-location), а уже унифицированного железа закупленного оптом для всего цента;

* экономия на ресурсах;

* виртуализация сети и компьютеров.

Следующую веху положил Amazon в 2010 году и ознаменовал эру облачных вычислений. Этап характеризуется строительством масштабных дата центов с заведомым избытком по мощностям для получения меньшей стоимости вычислительных мощностей за счёт опта в расчёте на экономию для себя и выгодную продажу их избытка в розницу. Такой подход применяется не только к вычислительной мощности, инфраструктуре, но и программному обеспечению, формирую его как сервисы для удешевления их использования за счёт их продажи в розницу как большим компаниям, так и начинающим.

Необходимость однотипности окружения

Обычно начинающие разработчики, разрабатывающие под Linux, предпочитают работать из-под Windows, чтобы не изучать незнакомую ОС и набивать на ней свои шишки, ведь раньше всё было далеко не всё так просто и так отлажено. Часто разработчики вынуждены работать из-под Windows из-за корпоративных пристрастий: 1С, Directum и другие системы работают только на Windows, и вся остальная, а главное сетевая, инфраструктура заточена под эту операционную систему. Работа из Windows приводит к большим потерям рабочего времени как разработчиков, так и DevOps на устранения как мелких, так и крупных отличий в операционных системах. Эти отличия начинают проявляться от самых простых задач, например, что может быть проще сверстать страничку на чистом HTML. Но неправильно настроенный редактор поставит в BOM и переводы строк, принятые в Windows: "\n\r" вместо "\n"). BOM при склейке шапки, тела и подвала страницы создаст отступы между ними, они не будут видны в редакторе, так как эти они образуются байтами метаинформации о типе файла, в Linux которые не имеют такого значения и воспринимаются как перевод отступ. Другие переводы строк в GIT не позволяет увидеть сделанные вами отличие, ведь отличие в каждой строке.

Теперь возьмём Front разработчика. С первого взгляда, что сложного, ведь JS (JavaScript), HTML и CSSнативно интерпретируются браузером. Раньше делалась вёрстка всех разных страниц – проверялась дизайнером и заказчиком и отдавалась PHP разработчику на интеграцию с фреймворком или cms. Для того, чтобы не править шапку на каждой страницы, а потом долго выяснять, когда они начали отличаться и какая из них правильнее использовался HAML. HAML добавляет дополнительный синтаксис в HTML, чтобы избежать булирования: циклы, подключения файлов, в нашем случае единую шапку и подвал страницы. Но он требует специальной программы, перегоняющий HTML в чистый HTML. В MS Windows это решается установкой программы компилятора и подключением её к IDE, благо все эти возможности есть в IDE WEBStorm. С CSSи его размером, дублями, зависимостями и поддержкой для разных браузеров всё гораздо сложнее – там использовался LESS, а сейчас возглавил более функциональный SASS и библиотеками поддержки разных браузеров, который требует компилятора RUBY и подобная связка, обычно, с первого раза не работает. А для JS использовался CoffeScript. Всё это нужно прогонять через программы сжатия и валидации (валидировать HTML обычно не нужно).

Когда проект начинает расти и перестаёт быть отдельными страницами с "JS вставками", а становится SPA (Single Page Application, одно страничными веб приложениями), где всё создаётся JS, и уже сборщиков (Galp, Grunt), менеджеров пакетов и NodeJS не собирается, сложностей становится всё больше. Все эти программы бесплатные и изначально разрабатывались для Linux, предназначены для работы из консоли BASH и под Windows ведёт себя не всегда хорошо и трудно автоматизируются в графических интерфейсах, не смотря на старания IDE разработчиков. Поэтому, многие WEB разработчики перешли с MS Windows на MacOS, который является ответвление UNIX систем, в него изначально встроен BASH.

Docker как легковесные виртуальные машины

Изначально, проблема изоляции положений и проектов решалась виртуализацией – системным программный обеспечением, которое эмулирует на определённом уровне среду,

которой может быть аппаратное обеспечение (компьютер как набор компонентов, таких как процессор, оперативная память, сетевое устройство и другие при необходимости) или, реже, операционная система. Системный администратор выбирает объём оперативной памяти (не более свободной), процессор, сетевое устройство. Устанавливает операционную систему и, при необходимости, драйвера, устанавливает необходимые программы. Если нужно рабочее место для второго разработчика – совершает те же действия. Для установки программ смотрит в каталог /bin первого и устанавливает недостающие. И тут возникает первая тихая проблема, пока не проявившаяся, что программы устанавливаются разных версий, но это будет головной болью уже разработчиков, если у одного разработчика наработает, а у другого нет, или головной болью этого сисадмина – если у разработчика работает, на продакшне – нет.

С ростом числа рабочих мест, добавляются следующие проблемы:

* Вам доступно менее 30% производительности от родительской системы, ведь все команды, которые должен выполнять процессор, выполняются программой виртуализации. Повысить производительность позволяет режим процессора VT-X, при котором процессор напрямую выполняет команды из виртуального окружения, а в случаи несовместимости – кидает исключение. Правда эти броски в сотни раз затратней обычных команд, поэтому взрослые системы виртуализации (VirtualBox, VMVare, и другие) стараются отфильтровать и модифицировать потенциально несовместимые команды, что позволяет существенно повысить производительность.

* Каждое рабочее место приходится создавать заново, для этого системный администратор пишет скрипт автоматизирующий этот процесс, но он естественно не идеален, и его приходя постоянно актуализировать, вносить патчи несовместимостей того, что устанавливали программисты.

* Линейное увеличение занимаемого дискового пространства от числа контейнеров, и экспоненциальное от версий продукта, при том, что один инстанс занимает очень много места. То есть, каждая песочница содержит инстанс эмуляции программы, образ операционной системы, все установленные программы и код разработчика, что весьма немало. Единственное, что одно – установка самой виртуальной машины, и то, только в рамках одного физического сервера. Например, если разработчиков у нас 10, то и размер будет в 10 раз больше, при 3 версиях продукта – в 30.

Все эти недостатки для WEB призван решить Docker. Здесь и далее мы будем говорить о Docker для Linux, и не будем рассматривать несколько отличающиеся реализацию ядра у FreeBSD, и реализацию для Windows 10 Professional, внутри которой закупается либо урезанное ядро Linux, либо независимая разработка контейнеризации Windows. Основная идея заключается не в создании прослойки (виртуализации аппаратного обеспечения, своей ОС и гипервизора), а в разграничении прав. Вы не множите поставить в контейнер MS Windows, но можете поставить и RedHut, и Debian, так как ядро одно, а отличии в файлах, создавая песочницу (отдельный каталоги и запрещая выходить за его приделы) с этими файлами. Также, мы говорим о WEB решениях, так как для нативных решений могут возникнуть проблемы, когда программе не обходимо иметь монопольный доступ из контейнера (песочницы Docker) к ядру ОС, например, для нативной отрисовки окон. Также можно ограничить объём памяти, процессорного времени, количества процессов.

Легковесная виртуализация или невесомая изоляция – взгляд на реализацию Docker

Давайте взглянем на историю появления предпосылок появления Docker, именно предпосылок, так как сам Docker не реализует ни изоляции, ни тем более виртуализации, а организует работу с ней с первой. В отличии от виртуализации напоминающей ангар со своим миром и своим фундаментом, на который можно наложить, что душе пожелает, например, выдаём и лужайку, то для изоляции можно провести аналогию с забором. Изоляция появлялась

в ядре Linux постепенно, частями, отвечающих за разные уровни, а параллельно появлялись программы обеспечивающие интерфейс и концепцию применения этой изоляции в реальных проектах. Изоляция состоит из 6 типов ограничения ресурсов.

Первым в ядро была изоляция файловой системы, позволяющий создать песочницу с помощью команды chroot ещё в 1979 году, из вне песочница видна полностью, но при переходе внутрь папка, над которой выполнена команда становится корневой, и вернуться уже не удастся. Следующий было разграничение процессов, так песочница существует и хостовая система, пока существует процесс с pid (номером) 1. Для песочницы он свой, вне песочницы он обычный процесс. Далее подтянулись стальные разграничения CGroups: групп пользователей, памяти и другие. Всё это существует в ядре любого Linux, независимо от того, установлен Docker или нет у вас. На всём протяжении истории принимались попытки, OpenSource и коммерческие, создавать контейнера, разрабатывая функционал самими и подобные решения находили своих пользователей, но они не проникли в широкие массы. Docker в начале своего существования использовал довольно стабильное, но сложное в использовании решение контейнеризации LXC. Постепенно он заменил LXC на нативные CGroup. Также Docker поддерживает солёность своего образа (об этом далее), но сам её не реализует, а использует UnuonFS (UFS).

Docker и дисковое пространство

Поскольку Docker не реализует функционал, а использует заложенный в ядро Linux, и не имеет под капотом графического интерфейса, сам он занимает очень мало места.

Поскольку контейнер использует ядро хостовой ОС, то в базовом образе (обычно ОС) содержатся только дополняющие его пакеты. Так Docker образ Debian занимает 125Mb, а ISO образ – 290Mb. Для проверки, что используется одно ядро в контейнере выведем о нём информацию: uname -a или cat /proc/version, а информацию о самом окружении контейнера cat /etc/ussue

Docker создаёт образ на основе инструкции в файле Dockerfile, который может находиться удалённо или локально, и по нему может быть создан в любой момент. Поэтому, если вы не используете образ в данный момент, то можно удалить его. Исключением является образ, созданный на основе контейнера командой Docker commit, но так создавать не очень правильно, и всегда можно выделить из образа Dockerfile командой Docker history и удалить образ. Преимуществом хранения образов является то, что не требуется ожидать, пока он создаётся: скачивается ОС и библиотеки.

Сам Docker использует образ под названием Image, который создаётся на основе инструкций в файле Dockerfile. При создании нескольких контейнеров на его основе место практически не увеличивается, так как контейнер – всего лишь процесс и конфиг настроек. При изменении файлов в контейнере сами файлы не сохраняются, а сохраняются внесённые изменения, который будут удалены после переброски контейнера. Это гарантирует в 99% случаев полностью идентичное окружение, и как следствие не важно помещать подготовительные операции, общие для всех контейнеров по установке специфичных программ в образ, побочным эффектом, которого является отсутствии их дублирования. Чтобы иметь возможность сохранять данные используется монтирование папок и файлов к хостовой (родительской) системе. Поэтому вы можете на обычном компьютере запустить сто и более контейнеров, при этом не увидите изменения в свободном местное на диске. При этом, если разработчики пользуются гит, а как же без него, и часто копятся, то может отпасть необходимость в монтировании папок с исходным кодом.

Образ докеров не представляет из себя монолитный образ вашего продукта, а – слоёный пирог образов, слои которого кэшируется. Это позволяет значительно сэкономить время на создание образа. Кэширование можно отключить ключом команды build –no-cache=true, если Docker не распознаёт, что данные изменяемы. Докер может увидеть изменения в инструкции

ADD, добавляющий файл из хостовой системы в контейнер по хэшу файла. Так, если вы создадите два контейнера, один с NGINX, а другой с MySQL, оба которых основаны на ОС Ubuntu 14.04, то будет существовать три слоя образа: MySQL, NGINX и Ubuntu. Сдали образа можно посмотреть командой Docker history. Также это работает и для ваших проектов – при копировании в Ваш образ 2 версий кода командой ADD с вашим продуктом, у вас будет 3 слоя и два образа: базовый, с кодом первой версии и кодом второй версии, независимо от количества контейнеров. Количество слоёв ограниченно 127. Важно заметить, что при клонировании проекта нужно указать версию, а не просто git clone, а git clone –branch v1 и git clone –branch v2, иначе Docker закэширует слой, создаваемый командой Git Clone и при создании второго образа мы получим тоже самое.

Docker не занимает ресурсы, а лишь их ограничивает, если это задано в настройках при создании контейнера (для памяти ключ m, для процессора – c). Поскольку Docker поддерживает разные файловые системы контейнеризации, настраивать, унифицированного интерфейса нет. Но, в любом случае, потребляется ресурс столько, сколько требуется, а не столько сколько выделено, как в виртуальных машинах.

Такая забота о занимаемом дисковом пространстве и невесомость самих контейнеров влечёт безответственность в скачивании образов и создании контейнеров.

Сборка мусора контейнером

За счёт того, что контейнер даёт намного большие возможности, чем виртуальная машина, ситуация осложняется оставление мусора после работы Docker. Проблема решается просто запуском сборщика мура, появившегося в версии 1.13 или более сложно для более ранних версий написанием нужно Вам скрипта.

Так же, как просто создать контейнере docker run name_image, также просто его и удалить docker rm -f id_container. Часто для того, чтобы просто поэкспериментировать, удобно запустить контейнер в интерактивном режиме docker run -ti name_image bash и мы стразу же окажемся в контейнере. Когда мы выйдем из него Cntl+D, он будет остановлен. Для того, чтобы поле выхода он был автоматически удалён используйте параметр –rm. Но, поскольку контейнеры столь невесомы, их так просто создать, их часто бросают и не удаляют, что приводит к их стремительному росту. Посмотреть на работающие можно командой docker ps, а и на остановленные – docker ps -a. Для предотвращения этого используйте сборщик мусора docker containers prune, который появился в версии 1.13 и который удалит все остановленные контейнера. Для более ранних версий используйте скрипт docker rm $(docker ps -q -f status=exited). Если её запуск для Вас не желателен, скорее всего вы неправильно используете Docker, так как запасть контейнер из образа практически также быстро и просто, как и восстановить его работу. Если в контейнере нужно сохранять состояние, то для этого используется монтирование папок или томов.

Чуть более сложная ситуация обстоит с образами. При создании контейнера, если нет образа, он будет скачен. Поскольку, один образ может являться для нескольких контейнеров, то при удалении самого контейнера он не удаляется. Его придётся удалять вручную docker rmi name_image, а если он используется – просто будет выдано предупреждение. За экономию дискового пространства приходится платить тем, что Docker не может просто определить, нужен образ ещё или нет. С версии 1.13 он может, с помощью команды docker imgae prune -a может проанализировать, какие образа не используются контейнерами и их удалить. Здесь нужно быть более осторожным, если Docker не может получить образ снова, но допущение подобной ситуации не очень правильно. Одной такой ситуацией является создание кластерного образа, при этом конфиг Dockerfile, описывающий процесс его создания, был утерян, в противном случае из Dockerfile можно получить образ командой docker build name_image. Правильно же сразу же принять меры и восстановить Dockerfile из образа, посмотрев на команды создающие образа с помощью Docker history name_image. Второй ситуаций является создание образа из

работающего контейнера командой Docker commit, а не из Dockerfile, так активно популяризуемого, но также активно осуждаемого.

Так как образ состоит из слоёв, совместно используемых в разных образах, то в разных нештатных ситуациях эти слои остаются. Поскольку отдельно мы их использовать не можем, то безопасно их удалить командой docker image prune.

Для сохранения результатов работы контейнера можно примонтировать папку хостовой машины к папке контейнера. Мы можем явно указать папку на хостовой машине, например, docker run -v /page_host:/page_container nama_image, или дать возможность сгенерировать её docker run -v /page_container nama_image. Для удаления сгенерированных папок (томов), которые уже не используются контейнерами введите команду Docker volume prune. Для сборки неиспользуемых сетей, также есть свой сборщик мусора.

Также есть единый сборщик мусора, по факту, просто объединяющий специализированные в один с логически совместимыми параметрами docker system prune. Имеется тенденция его ставить в крон. Можно также посмотреть на занимаемое место всеми контейнерами, всеми образами и всеми томами с помощью команды docker system df, а также без группировки – docker system df -v.

Многие проблемы, описанные здесь созданием мусора, решаются программой Docker-compose. К тому же она существенно упрощает жизнь, если только вы не запустили контейнера разово для экспериментов. Так команда Docker-compose up запускает контейнера, а docker-compose down -v их удаляет, также удаляются все зависимости между ними. Все параметры запуска контейнера описываются в Docker-compose.YML, а также связи между ними. Благодаря этому, при изменении параметров запуска контейнеров не нужно заботиться, чтобы удалить старые и создать новые, не нужно прописывать все параметры контейнеров – просто запасть с параметром up, и она либо пересоздаст, либо обновит конфигурацию контейнера.

Для недопущения захламления системы в Docker имеется встроенное настраиваемое ограничение на количество контейнеров и образов, напоминающее о необходимости производить чистку системы запуском сборщика мусора.

Экономия времени на создание контейнера

Мы уже познакомились в предыдущей теме об образах, об их слоях и кэшировании. Давайте рассмотрим их с точки зрения времени создания контейнера. Почему же это столь важно, ведь по аналогии с виртуализацией, системный администратор запустил создание контейнера и пока он передаёт его программисту, к этому времени он уже точно соберётся. Важно заметить, что много с тех пор изменилось, а именно поменялись принципы и требования к экосистеме и её использования. Так, например, если раньше разработчик, разработав и проверив свой код на своём рабочем месте отправлял его QA менеджеру для тестирования на соответствии бизнес-требованиям, и уже когда дойдёт очередь у него к этому коду, тестировщик на своём рабочем месте запустит этот код и проверит. Теперь инфраструктурой занимается DevOps, который налаживает непрерывный процесс доставки разработанных программистами фич, а контейнеры создают в автоматическом режиме при каждом отправкой, в ветку продакшна для проведения автоматического тестирования. При этом, чтобы работа одних тестов не влияла на работу других, под каждый тест создаётся отдельный контейнер, а зачастую тесты идут параллельно, чтобы моментально показать результат разработчику, пока он помнит, что он сделал и не переключил своё внимание на другую задачу.

Для стандартных программы: не нужно устанавливать, не нужно поддерживать

Мы, часто используем огромного количество готовых решений. При выборе решения, мы сталкиваемся с дилеммой: с одной стороны оно более универсальное и более проверенное, чем мы можем себе позволить сделать, с другой оно достаточно сложное, чтобы самим разобраться как правильно его установить и настроить, чтобы установить все зависимости, разрешить конфликты, настроить на первоначальное использование. Теперь установка и настройка стала

гораздо проще, стандартизированный, во многом отсутствуют низкоуровневые проблемы. Но прежде, чем продолжать, давайте отвлечёмся и посмотрим на процесс от начала получения до начала использования приложения в рамках истории:

* В те времена, когда все программы писались на ассемблере, программы распространялись по почте, у уже пользователи устанавливали и тестировали, ведь тестирование в компаниях не было предусмотрено. В случае возникновения проблем, пользователь сообщал компании разработчику о проблемах и после их устранения, получал по почте уже исправленную версию на диске. Процесс очень долгий и пользователь сам тестировал.

* Во время распространения на дисках уже компании писали свои программные продукты на более высокоуровневых языках, проверяли под разные версии ОС. Здесь и далее будем рассматривать свободное ПО. Программа уже содержала MakeFile, который сам компилировал программу и её устанавливал.

* С появление интернета массово ПО устанавливается с помощью пакетных менеджеров, при выходе которых, из удалённого репозитория ОС оно скачивается и устанавливается. Он старается следить и поддерживать совместимость совместимость программ. Дальнейшее изучение и использование программы: как запустить, как настроить, как понять что она работает ложится на пользователя или системного администратора.

* С появлением Docker Hub и WEB приложения скачиваются и запускаются контейнером. Его, обычно, для начальной работы не нужно настраивать.

Для контейнеров и образов в целом у сервера можно настроить объёма свободно места и занимаемого пространства. По умолчания для всех контейнеров и образов отводится 10G, при этом из этого объёма должно оставаться как dm.min_free_space=5%, но лучше поместить в конфиг, который возможно придётся создать как /etc/docker/daemon.json:

```
{
"storage-opts": [
"dm.basesize=50G",
"dm.min_free_space=5%",
]
}
```

Можно ограничить ресурсы, потребляемые контейнером в его настройках:

* -m 256m – максимальный размер потребления оперативной памяти (здесь 256Mb);
* -c 512 – вес приоритета на использование процессора (по умолчанию 1024);
* —cpuset="0,1" – номера разрешённых к использованию ядер процессора.

Передача и распространение продукта

Для передачи проекта, например, заказчику, и распространения между разработчиками и серверами можно использовать установочные скрипты, архивы, образа и контейнера. Каждый из этих способов распространения проекта имеет свои особенности, недостатки и преимущества. Давайте о них поговорим и сравним.

строк, но главное, что у неё есть специальный режим, включаемый ключом -p, который выводит динамически нужно нам количество строк, при поступлении новых – обновляет вывод, например, docker logs name_container | tail -p.

Когда приложений становится слишком много, чтобы вручную мониторить их работу по отдельности целесообразно логи приложений централизовать. Для централизации могут быть использованы многочисленные программы, которые собирают логи от разных сервисов и направляют их в центрально хранилище, например, Fluentd. Для хранения логов удобно использовать ElasticSearch, просто записывая их в поисковик. Очень желательно, чтобы логи были в структурированном формате – JSON. Это позволит их сортировать, отбирать нужные, выявлять тенденции с помощью встроенных агрегатных функций, проводить анализ и прогно-

зирование, а не только поиск по тексту. Для анализа веб-интерфейс Kubana, в ходящий в состав стека Elastic.

Не только для долгоиграющих приложений важно логирование. Так для тестовых контейнеров, удобно получить вывод пройденных тесто. Так можно сделать, прописав в Dockerfile в секции CMD: NPM run, который прогонит тесты.

Хранилище образов:

* публичный и приватный Docker Hub (http://hub.docker.com)

* для закрытых и секретных проектов можно создать собственное хранилище образов. Образ называется registry

Docker для создания приложений и одноразовые работы

В отличии от виртуальных машин запуск, которых сопряжён с значительными человеческими и вычислительными затратами Docker часто применяется для выполнения одноразовых действий, когда программное обеспечение требуется запустить разово, и желательно не тратить усилии на установку и удаление его. Для этого запускается контейнер, который примонтирован к папке с нашим приложением, который выполняет над ним требуемые действия и после их выполнения удаляется. В качестве примера можно проект на JavaScript, для которого нужно провести сборку и выполнить тесты. При этом сам проект не использует NodeJS, а содержит только конфиги сборщика, например, WEBPack-ка, и написанные тесты. Для этого запускаем контейнер сборки в итеративном режиме, в котором можно управлять процессом сборки, если потребуется и после выполнения сборки контейнер остановится и само удалится, например, можно запустить в корне приложение что-то подобное: docker run -it –rm -v $(pwd):app node-build. Аналогичным образом можно провести тесты. В результате приложение собрано и протестировано на тестовом сервере, но при этом программное обеспечение не требующееся для его работы на продакшн сервере не буде усыновлено и потреблять ресурсы, и может быть перенесено на продакшн сервер, например с помощью контейнера. Чтобы не писать документацию на запуске сборки и тестированию можно положить два соответствующих конфига Docker-compose-build.yml и Docker-compose-test.yml и вызывать их Docker-compose up -f ./ docker-compose-build.

Управление и доступ

Мы управляем контейнерами с помощью команды Docker. Теперь, допустим, появилась необходимость управлять удалённо. Использовать VNC, SSH или другое для управления командой Docker, наверное, будет слишком трудоёмким если задача усложнится. Правильно, сперва будет разобраться, что из себя представляет Docker, ведь команда Docker и программа Docker не одно и тоже, а точнее команда Docker является консольным клиентом для управления клиент серверным приложение Docker Engine. Команда взаимодействует с сервером Docker Machine через Docker REST API, который и предназначен для удалённого взаимодействия с сервером. Но, в таком случае необходимо позаботиться об авторизации и SSL-шифровании трафика. Обеспечивается это созданием ключей, но в общем случае, если стоит задача централизованного управления, разграничения прав и обеспечения безопасности – лучше посмотреть в сторону продуктов, изначально обеспечивающих это и использующих Docker в качестве именно запуска контейнеров, а не как систему.

По умолчанию, для безопасности, клиент общается с сервером через Unix сокет (специальный файл /var/run/socket.sock), а не через сетевой сокет. Для работы через Unix сокет можно использовать указать программе отправки запросов curl его использовать curl –unix-socket /var/run/docker.sock http:/v1.24/containers/json, но это возможность поддерживается с версии curl 1.40, которая пока не поддерживается в CentOS. Для решения этой проблемы и взаимодействия между удалёнными серверами используем сетевой сокет. Для активации его останавливаем сервер systemctl stop docker и запускаем его с настройками dockerd -H tcp://0.0.0.0:9002 & (слушать всех на порту 9002, что не допустимо для продакшна). После

запуска команда docker ps не будет работать, а будет docker -H 5.23.52.111:9002 ps или docker -H tcp://geocode1.essch.ru:9202 ps. По умолчанию Docker использует для http порт 2375, а для https – 2376. Чтобы не менять везде код и каждый раз не указывать сокет, пропишем его в переменных окружения:

```
export DOCKER_HOST=5.23.52.111:9002
Docker ps
Docker info
unser DOCKER_HOST
```

Важно прописать export для обеспечения доступности переменной всем программам: дочерним процессам. Также не должно стоять пробелов вокруг =. Теперь мы так можем обращаться из любого места находящимся в одной сети сервером Dockerd. Для управления удалёнными и локальными Docker Engine (Docker на хостах) разработана программа Docker Machine. Взаимодействие с ней осуществляется с помощью команды Docker-machine. Для начала установим:

```
base=https://github.com/docker/machine/releases/download/v0.14.0 &&
curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/usr/local/bin/docker-machine &&
chmod +x /usr/local/bin/docker-machine
```

Группа взаимосвязанных приложений

У нас уже есть несколько разных приложений, допустим, таких как NGINX, MySQL и наше приложение. Мы их изолировали в разных контейнерах, теперь она не конфликтуют между собой и NGINX и MySQL мы не стали тратить время и усилия на изготовление собственной конфигурации, а просто скачали: Docker run mysql, docker run Nginx, а для приложения docker build .; docker run myapp -p 80:80 bash. Как видно, казалось бы, всё очень просто, но есть два момента: управление и взаимосвязь.

Для демонстрации управления возьмём контейнер нашего приложения, и реализуем две возможности – старт и создание (переборку). Для ручного старта, когда мы знаем, что контейнер уже создан, но просто остановлен, достаточно выполнить docker start myapp, но для автоматического режима этого недостаточно, а нам нужно написать скрипт, который бы учитывал, существует ли уже контейнер, существует ли для него образ:

```
if $(docker ps | grep myapp)
then
docker start myapp
else
if ! $(docker images | grep myimage)
docker build .
fi
docker run -d –name myapp -p 80:80 myimage bash
fi
```

. А для создания нужно удалить контейнер, если он существует:

```
if $(docker ps | grep myapp)
docker rm -f myapp
fi
if ! $(docker images | grep myimage)
docker build .
fi
docker run -d –name myapp -p 80:80 myimage bash
```

. Понятно, что нужно общими параметры, название образа, контейнера вывести в переменные, проверить, что Dockerfile есть, он валидный и только после этого удалять контейнер и много другого. Для понимания реального масштаба, на вдаваясь во взаимодействие контейнеров, о клонировании (масштабирование) этих групп и тому подобного, а только упомяну, то, что команда Docker run может превышать один – два десятка строк. Например, десяток проброшенных портов, монтируемых папок, ограничения на память и процессор, связи с другими контейнерами и ещё немного специфичных параметров. Да, это нехорошо, но делить в таком варианте на множество контейнеров сложно, из-за отсутствия карты взаимодействия контейнеров. Но возникает вопрос: Не много ли нужно делать, чтобы просто предоставить пользователю возможность стартовать контейнер или пересобрать? Зачастую ответ системного администратора сводится к тому, что давать доступы можно только избранным. Но и тут есть решение: Docker-compose – инструмент для работы с группой контейнеров:

```
#docker-compose
version: v1
services:
myapp:
container-name: myapp
images: myimages
ports:
– 80:80
build: .
```

. Для старта docker-compose up -d, а для переборки docker down; docker up -d. Причём, при изменении конфигурации, когда не нужна полная переборка, произойдёт просто её обновление.

Теперь, когда мы упрости процесс управления одиночным контейнером, давайте поработаем с группой. Но тут, для нас изменится только сам конфиг:

```
#docker-compose
version: v1
services:
mysql:
images: mysql
Nginx:
images: Nginx
ports:
– 80:80
myapp:
container-name: myapp
build: .
depence-on: mysql
images: myimages
link:
– db:mysql
– Nginx:Nginx
```

. Здесь мы видим всю картину в целом, контейнера связаны одной сетью, где приложение может обращаться к mysql и NGINX по хостам db и NGINX, соответственно, контейнер myapp

будет создан, только когда после поднятия базы данных mysql, даже если это займёт некоторое время.

Service Discovery

С ростом кластера вероятность падения нод вырастает и ручное обнаружение произошедшего усложняется, для автоматизации обнаружение вновь появившихся сервисов и их исчезновение предназначены системы Service Discovery. Но, чтобы кластер мог обнаруживать состояние, учитывая, что система децентрализована – ноды должны уметь обмениваться сообщениями с друг другом и выбирать лидера, примерами могут быть Consul, ETCD и ZooKeeper. Мы рассмотрим Consul исходя их следующих его особенностей: вся программа – одни файл, крайне прост в работе и настройке, имеет высокоуровневый интерфейс (ZooKeeper его не имеет, полагается, что со временем должны появиться сторонние приложения, его реализующие), написан на не требовательном языке к ресурсам вычислительной машины (Consul – Go, ZooKeeper – Java) и пренебрегли его поддержкой в других системах, такай как, например, ClickHouse (поддерживает по умолчанию ZooKeeper).

Проверим распределение информации между нодами с помощью распределенного key-value хранилища, то есть, если мы в одну ноду добавили записи, то они должны распространиться и на другие ноды, при этом жёстко заданного координирующего (Master) ноды у неё не должно быть. Поскольку Consul состоит из одного исполняемого файла – скачиваем его с официального сайта по ссылке https://www.consul.io/downloads. html на каждой ноде:

```
wget https://releases.hashicorp.com/consul/1.3.0/consul_1.3.0_linux_amd64.zip -O consul.zip
unzip consul.zip
rm -f consul.zip
```

Теперь необходимо запустить одну ноду, пока, как master consul -server -ui, а другие как slave consul -server -ui и consul -server -ui . После чего остановим Consul, находящийся в режиме master, и запускаем его как равного, в результате Consul – переизберут временного лидера, а в случае иго падения, переизберут снова. Проверим работу нашего кластера consul members:

```
consul members;
```

И так проверим распределение информации нашего хранилища:

```
curl -X PUT -d 'value1' .....:8500/v1/kv/group1/key1
curl -s .....:8500/v1/kv/group1/key1
curl -s .....:8500/v1/kv/group1/key1
curl -s .....:8500/v1/kv/group1/key1
```

Настроим мониторинг сервисов, подробнее в документации https://www.consul.io/docs/agent/options. html #telemetry, для этого https://medium.com/southbridge/monitoring-consul-with-statsd-exporter-and-prometheus-bad8bee3961b

Чтобы не настраивать, воспользуемся контейнером и режимом для разработки с уже настроенным IP-адресом на 172.17.0.2:

```
essh@kubernetes-master:~$ mkdir consul && cd $_

essh@kubernetes-master:~/consul$ docker run -d –name=dev-consul -e CONSUL_BIND_INTERFACE=eth0 consul
Unable to find image 'consul:latest' locally
latest: Pulling from library/consul
e7c96db7181b: Pull complete
3404d2df15cb: Pull complete
1b2797650ac6: Pull complete
```

42eaf145982e: Pull complete
cef844389e8c: Pull complete
bc7449359c58: Pull complete
Digest:
sha256:94cdbd83f24ec406da2b5d300a112c14cf1091bed8d6abd49609e6fe3c23f181
Status: Downloaded newer image for consul:latest
c6079f82500a41f878d2c513cf37d45ecadd3fc40998cd35020c604eb5f934a1

essh@kubernetes-master:~/consul$ docker inspect dev-consul | jq ' .[] | .NetworkSettings.Networks.bridge.IPAddress'
"172.17.0.4"

essh@kubernetes-master:~/consul$ docker run -d –name=consul_follower_1 -e CONSUL_BIND_INTERFACE=eth0 consul agent -dev -join=172.17.0.4
8ec88680bc632bef93eb9607612ed7f7f539de9f305c22a7d5a23b9ddf8c4b3e

essh@kubernetes-master:~/consul$ docker run -d –name=consul_follower_2 -e CONSUL_BIND_INTERFACE=eth0 consul agent -dev -join=172.17.0.4
babd31d7c5640845003a221d725ce0a1ff83f9827f839781372b1fcc629009cb

essh@kubernetes-master:~/consul$ docker exec -t dev-consul consul members
Node Address Status Type Build Protocol DC Segment
53cd8748f031 172.17.0.5:8301 left server 1.6.1 2 dc1 < all>
8ec88680bc63 172.17.0.5:8301 alive server 1.6.1 2 dc1 < all>
babd31d7c564 172.17.0.6:8301 alive server 1.6.1 2 dc1 < all>

essh@kubernetes-master:~/consul$ curl -X PUT -d 'value1' 172.17.0.4:8500/v1/kv/group1/key1
true

essh@kubernetes-master:~/consul$ curl $(docker inspect dev-consul | jq -r ' .[] | .NetworkSettings.Networks.bridge.IPAddress'):8500/v1/kv/group1/key1
[
{
"LockIndex": 0,
"Key": "group1/key1",
"Flags": 0,
"Value": "dmFsdWUx",
"CreateIndex": 277,
"ModifyIndex": 277
}
]

essh@kubernetes-master:~/consul$ firefox $(docker inspect dev-consul | jq -r ' .[] | .NetworkSettings.Networks.bridge.IPAddress'):8500/ui

С определением местоположения контейнеров необходимо обеспечить авторизацию, для этого используются хранилища ключей.

dockerd -H fd:// –cluster-store=consul://192.168.1.6:8500 –cluster-advertise=eth0:2376

* –cluster-store – можно получать данные о ключах
* –cluster-advertise – можно сохранять

docker network create –driver overlay –subnet 192.168.10.0/24 demo-network
docker network ls

Простая кластеризация

В данной стать мы не будем рассматривать как создать кластер вручную, а воспользуемся двумя инструментами: Docker Swarm и Google Kubernetes – наиболее популярные и наиболее распаренные решения. Docker Swarm проще, он является частью Docker и поэтому, имеет наибольшую аудиторию (субъективно), а Kubernetes предоставляет гораздо большие возможности, большее число интеграций с инструментами (например, распределённое хранилище для Volume), поддержка в популярных облаках и более просто масштабируемый для больших проектов (большая абстракция, компонентный подход).

Рассмотрим, что такое кластер и что он нам хорошего принесёт. Кластер— это распределённая структура, которая абстрагирует независимые сервера в одну логическую сущность и автоматизирует работу по:

* случае падения одного сервера, переброс контейнеров (создания новых) на другие сервера;
* равномерное распределение контейнеров по серверам для отказоустойчивости;
* создание контейнера на сервере, подходящем по свободным ресурсам;
* Разворачивание контейнера, в случае выхода его из строя;
* единый интерфейс управления из одной точки;
* выполнения операций с учётом параметров серверов, например, размера и типа диска и особенностей контейнеров, заданных администратором, например, связанные контейнера единым точкой монтирования размещаются на данном сервере;
* унификация разных серверов, например, на разных ОС, облачных и не облачных.

Теперь мы перейдём от рассмотрения Docker Swarm к Kubernetes. Обе эти системы – системы оркестрации, обе работаю с контейнерами Docker (Kubernetes также поддерживает RKT и Containerd), но взаимодействий между контейнерами принципиально другое из-за дополнительного уровня абстракции Kubernetes – POD. И Docker Swarm, и Kubernetes управляют контейнерами на основе IP адресов и распределяют их по нодам, внутри которых всё работает через localhost, проксируемый мостом, но в отличии от Docker Swarm, который работает для пользователя с физическими контейнерами, Kubernetes для пользователя работает с логическими – POD. Логический контейнер Kubernetes состоит из физических контейнеров, сетевое взаимодействие, между которыми происходит через их порты, поэтому они не дублируются.

Обе системы оркестрации используют оверлейную сеть (Overlay Network) между нодами хостами для эмуляции нахождения управляемых единиц в едином локальном сетевом пространстве. Данный тип сети является логическим типом, который использует для транспорта обычные сети TCP/IP и призвана эмулировать нахождение нод кластера в единой сети для управления кластером и обмена информации между его нодами, тогда как на уровне TCP/IP они не могут быть связанны. Дело в том, что, когда разработчик разрабатывает кластер, он может описать сеть только для одной ноду, а при развёртывании кластера создаётся несколько его экземпляров, причём их количество может динамически меняться, а в одной сети не может существовать трёх нод с одним IP адресами и подсетями (например, 10.0.0.1), а требовать от разработчика указывать IP адреса неправильно, поскольку не известно, какие адреса свободны и сколько их потребуется. Данная сеть берёт на себя отслеживания реальных IP адресов узлов, которые могут выделяться из свободных рандомно и меняться по мере пересоздания нод в кла-

стере, и предоставляет возможность обращаться к ним через ID контейнеров/ POD. При таком подходе пользователь обращается к конкретным сущностям, а не динамики меняющимся IP адресам. Взаимодействие осуществляется с помощью балансировщика, который для Docker Swarm логически не выделен, а в Kubernetes он создаётся отдельной сущностью для выбора конкретной реализации, как и другие сервисы. Подобный балансировщик должен присутствовать в каждом кластере и, но называется в рамках экосистемы Kubernetes сервисом (Service). Его можно объявить, как отдельно как Service, так и в описании с кластером, например, как Deployment. К сервису можно обратиться по его IP-адресу (посмотреть в его описании) или по имени, которое регистрируется как домен первого уровня во встроенном DNS сервере, например, если имя сервиса, заданного в метаданных my_service, то к кластеру можно обратиться через него так: curl my_service;. Это является довольно стандартным решением, когда компоненты системы вместе с их IP адресами меняются со временем (пересоздаются, добавляются новые, удаляются старые) – направлять трафик через прокси сервер, IP – или DNS адреса для внешней сети остаются постоянными, а внутренние могут меняться, оставляя заботу согласование их на прокси сервере.

Обе системы оркестрации использую оверлейную сеть Ingress для предоставления к себе доступа из внешней сети через балансировщик, который согласует внутреннею сеть с внешней на основе таблиц соответствия IP адресов ядра Linux (iptalbes), разделяя их и позволяя обмениваться информацией, даже при наличии одинаковых IP адресов во внутренней и внешней сети. А, вот, для поддержания соединения между этими потенциально конфликтными на IP уровне сетями используется оверлейная Ingress сеть. Kubernetes предоставляет возможность создать логическую сущность – контроллер Ingress, который позволит настроить сервис LoadBalancer или NodePort в зависимости от содержимого трафика на уровне выше HTTP, например, маршрутизацию на основе путей адресов (application router) или шифрование трафика TSL/HTTPS, как это делаю GCP и AWS.

Kubernetes – результат эволюции через внутренне проекты Google через Borg, затем через Omega, на полученном опыте от экспериментов сложилась довольно масштабируемая архитектура. Выделим основные типы компонентов:

* POD – обычный POD;

* ReplicaSet, Deployment – масштабируемые POD;

* DaemonSet – в каждой ноде кластера он создаётся;

* сервисы (отсортированы по мере важности): ClusterIP (по умолчанию, базовый для остальных), NodePort (перенаправляет порты, открытые в кластере, у каждого POD, на порты из диапазона 30000—32767 для обращения к конкретным POD из внешней), LoadBalancer (NodePort с возможностью создания публичного IP-адреса для доступа в интернет в таких публичных облаках, как AWS и GCP), HostPort (открывает порты на хостовой машине соответствующие контейнеру, то есть если в контейнере открыт порт 9200, он будет открыт и на хостовой машине для прямого перенаправления трафика) и HostNetwork (контейнеры в POD будут находиться в сетевом пространстве хоста).

Мастер как минимум содержит: kube-APIserver, kube-sheduler и kube-controller-manager. Состав слейва:

* kubelet – проверка работоспособности компонента системы (ноды), создание и управление контейнерами. Он находится на каждой ноде, обращается к kube-APIserver и приводит в соответствие ноду, на которой расположен.

* cAdviser – мониторинг ноды.

Допустим у на есть хостинг, и мы создали три AVS сервера. Теперь необходимо на каждый сервер установить Docker и Docker-machine, о том, как это сделать было рассказано выше. Docker-machine сама является виртуальной машиной для Docker контейнеров, мы соберём лишь внутренний для неё драйвер – VirtualBox – чтобы не устанавливать дополнительные

пакеты. Теперь, из операций, которые должны быть выполнены на каждом сервере останется создать Docker машины, остальные же операции по настройки и созданию контейнеров на них можно выполнять из master-ноды, а они буду автоматически запущены на свободных нодах и перераспределяться при изменении их количества. Итак, запустим Docker-machine на первой ноде:

docker-machine create –driver virtualbox –virtualbox-cpu-count "2" –virtualbox-memory "2048" –virtualbox-disk-size "20000" swarm-node-1
docker-machine env swarm-node-1 // tcp://192.168.99.100:2376
eval $(docker-machine env swarm-node-1)

Запускаем вторую ноду:
docker-machine create –driver virtualbox –virtualbox-cpu-count "2" –virtualbox-memory "2048" –virtualbox-disk-size "20000" swarm-node-2
docker-machine env swarm-node-2
eval $(docker-machine env swarm-node-2)

Запускаем третью ноду:
docker-machine create –driver virtualbox –virtualbox-cpu-count "2" –virtualbox-memory "2048" –virtualbox-disk-size "20000" swarm-node-3
eval $(docker-machine env swarm-node-3)

Подключимся к первой ноде инициализируем в ней распределённое хранилище и передаём ему адрес ноды менеджера (лидера):
docker-machine ssh swarm-node-1
docker swarm init –advertise-addr 192.168.99.100:2377
docker node ls // выведет текущий
docker swarm join-token worker

Если токены будут забыты, их можно получить, выполнив в ноде с распределённым хранилищем команды docker swarm join-token manager и docker swarm join-token worker.

Для создания кластера необходимо зарегистрировать (присоединить) все его будущие ноды командой Docker swarm join –token … 192.168.99.100:2377, для аутентификации используется токен, для их обнаружения необходимо, чтобы они находились в одной подсети. Посмотреть все сервера можно командой docker node info

Команда docker swarm init создаст лидера кластера, пока одинокого, но в полученном ответе будет на неё будет придана команда, необходимая для подключение других нод к этому кластеру, важная информация в котором – токен, например, docker swarm join –token … 192.168.99.100:2377. Подключимся к оставшимся нодам по SSH командой docker-machine SSH name_node и выполним её.

Для взаимодействия контейнеров используется сеть bridge, которая является свитчем. Но для работы нескольких реплик нужна подсеть, так как все реплики будут с одинаковыми портами, а проксирование производится по ip с помощью распределённого хранилища, при этом не имеет значение физически они расположены на одном сервере или разных. Следует сразу заметить, что балансировка осуществляется по правилу roundrobin, то есть поочерёдно к каждой реплике. Важно создать сеть типа overlay для создания DNS по верх неё и возможности обращаться к репликам по им именам. Создадим подсеть:

docker network create –driver overlay –subnet 10.10.1.0/24 –opt encrypted services

Теперь нам нужно наполнить на кластер контейнерами. Для этого создаём не контейнер, а сервис, который является шаблоном для создания контейнеров на разных нодах. Количество создаваемых реплик указывается при создании в ключе –replicas, причём распределение случайное по нодам, но по возможности равномерное. Помимо реплик, сервис имеет балансировщик нагрузки, порты с которого на которые (входные порты для всех реплик) производится проксирование указывается в ключе -p, а Server Discovery (обнаружение работающих реплик, определение их ip-адресов, перезапуск) реплик балансировщик осуществляет самостоятельно.

docker service create -p 80:80 –name busybox –replicas 2 –network services busybox sleep 3000

Проверим состояние сервиса docker service ls, состояние и равномерность распределения реплик контейнеров docker service ps busybox и его работу wget -O- 10.10.1.2. Сервис – более высокоуровневая абстракция, которая включает в себя контейнер и организующее его обновление (далеко не только), то есть для обновления параметров контейнера не нужно его удалять и создавать, а просто обновить сервис, а уже сервис сперва создаст новый с обновлённой конфигурацией, а только после того, как он запустится удалит старый.

Docker Swarm имеет свой балансировщик нагрузки Ingress load balacing, который балансирует нагрузку между репликами на порту, объявленном при создании сервиса, в нашем случае это 80 порт. Входной точкой может быть любой сервер с этим портом, но ответ будет получен от сервера, на который был перенаправлен запрос балансировщиком.

Мы также можем сохранять данные на хостовую машину, как и в обычном контейнере, для этого есть два варианта:

docker service create –mount type=bind, src=…, dst=.... name=.... ..... #

docker service create –mount type=volume, src=…, dst=.... name=.... ..... # на хост

Развёртывание приложения производится через Docker-compose, запущенного на нодах (репликах). При обновлении конфигурации Docker-compose нужно обновить Docker stack, и кластер будет последовательно обновлён: одна реплика будет удалена и в место неё будет создана новая в соответствии с новым конфигом, далее следующая. Если произошла ошибка – буде произведён откат к предыдущей конфигурации. Что ж, приступим:

docker stack deploy -c docker-compose.yml test_stack

docker service update –label-add foo=bar Nginx docker service update –label-rm foo Nginx docker service update –publish-rm 80 Nginx docker node update –availability=drain swarm-node-3 swarm-node-3

Docker Swarm

$ sudo docker pull swarm

$ sudo docker run –rm swarm create

docker secrete create test_secret docker service create –secret test_secret cat /run/secrets/test_secret проверка работоспособности: hello-check-cobbalt пример pipeline: trevisCI –> Jenkins –> config -> https://www.youtube.com/watch?v=UgUuF_qZmWc https://www.youtube.com/watch?v=6uVgR9WPjYM

Docker в масштабах компании

Давайте посмотрим в масштабах компании: у нас есть контейнера и есть сервера. Не важно, это две виртуалки и несколько контейнеров или это сотни серверов и тысячи контейнеров, проблема в том, чтобы распределить контейнера на серверах нужен системный администратор и время, если мало времени и много контейнеров, нужно много системных администраторов, иначе они будут неоптимальное распределены, то есть сервер (виртуальная машина)

работает, но не в полную силу и ресурсы продают. В этой ситуации для оптимизации распределения и экономии человеческих ресурсов предназначены системы оркестрации контейнеров.

Рассмотрим эволюцию:

* Разработчик создаёт необходимые контейнера руками.

* Разработчик создаёт необходимые контейнера используя для этого заранее подготовленные скрипты.

* Системный администратор, с помощью какой-либо системы управления конфигурации и развёртывания, таких как Chef, Pupel, Ansible, Salt, задаёт топологию системы. В топологии указывается какой контейнер на каком месте располагается.

* Оркестрация (планировщики) – полуавтоматическое распределение, поддержание состояния и реакция на изменение системы. Например: Google Kubernetes, Apache Mesos, Hashicorp Nomad, Docker Swarm mode и YARN, который мы рассмотрим. Также появляются новые: Flocker (https://github.com/ClusterHQ/Flocker/), Helios (https://github.com/spotify/helios/).

Существует нативное решение Docker-swarm. Из взрослых систем наибольшую популярность показали Kubernetes (Kubernetes) и Mesos, при этом первый представляет из себя универсальную и полностью готовую к работе систему, а второй – комплекс разнообразных проектов, объединённых в единый пакет и позволяющие заменить или изменять свои компоненты. Существует также огромное количество менее популярных решений, не продвигаемы такими гигантами, как Google, Twitter и другими: Nomad, Scheduling, Scalling, Upgrades, Service Descovery, но мы их рассматривать не будем. Здесь мы будем рассматривать наиболее готовое решение – Kubernetes, которое завоевало большую популярность за низкий порог вхождения, поддержки и достаточную гибкость в большинстве случае, вытеснив Mesos в нишу кастомизируемых решений, когда кастомизация и разработка экономически оправдана.

У Kubernetes есть несколько готовых конфигураций:

* MiniKube – кластер из одной локальной машины, предназначен для преодоления порога вхождения и экспериментов;

* kubeadm;

* kops;

* Kubernetes-Ansible;

* microKubernetes;

* OKD;

* MicroK8s.

Для самостоятельного запуска кластера можно воспользоваться

KubeSai – бесплатный Kubernetes

Наименьшая структурная единица называется POD, которая соответствует YML-файлу в Docker-compose. Процесс создания POD, как и других сущностей производится декларативно: через написание или изменение конфигурационного YML-файла и применение его к кластеру. И так, создадим POD:

```
# test_pod.yml
# kybectl create -f test_pod.yaml
containers:
– name: test
image: debian
```

Для запуска нескольких реплик:

```
# test_replica_controller.yml
# kybectl create -f test_replica_controller.yml
```

apiVersion: v1
kind: ReplicationController
metadata:
name: Nginx
spec:
replicas: 3
selector:
app: Nginx // метка, по которой реплика определяет наличие запущенных контейнеров
template:
containers:
– name: test
image: debian

Для балансировки используется разновидность service (логическая сущность) – LoadBalancer, кроме которого существует ещё ClasterIP и Node Port:

appVersion: v1
kind: Service
metadata:
name: test_service
apec:
type: LoadBalanser
ports:
– port: 80
– targetPort: 80
– protocol: TCP
– name: http
selector:
app: WEB

Плагины overlay сети (создаётся и настраивается автоматически): Contig, Flannel, GCE networking, Linux bridging, Calico, Kube-DNS, SkyDNS. #configmap apiVersion: v1 kind: ConfigMap metadata: name: config_name data:

Аналогично секретам в Docker-swarm существует секрет и для Kubernetes, примером которых могут быть настройки NGINX :

#secrets
apiVersion: v1
kind: Secrets
metadata:name: test_secret
data:
password: ....

А для добавления секрета в POD, нужно его указать в конфиге POD:

....
valumes:
secret:
secretName: test_secret
…

У Kubernetes больше разновидностей Volumes:

* emptyDir;
* hostPatch;
* gcePersistentDisc – диск на Google Cloud;
* awsElasticBlockStore – диск на Amazon AWS.
volumeMounts:
– name: app
nountPath: ""
volumes:
– name: app
hostPatch:
....

Особенностью буде UI: Dashbord UI
Дополнительно имеется:
* Main metrics – сбор метрик;
* Logs collect – сбор логов;
* Scheduled JOBs;
* Autentification;
* Federation – распределение по дата-центрам;
* Helm – пакетный менеджер, аналог Docker Hub.
https://www.youtube.com/watch?v=FvlwBWvI-Zg

Команды Docker

Docker – более современный аналог контейнеров RKT.

В Linux, когда завершается процесс с PID=1, то зарывается и NameSpace, что приводит к завершению работы ОС, в случае контейнера, аналогично, так как он является частным случаем ОС. Разграничение процессов сам по себе не даёт дополнительного оверхед, также как и мониторинг и ограничение ресурсов на процессы, ибо такие же возможности по настройки предоставляет systemd в хостовой ОС. Виртуализация сети происходит полностью: и localhost и мост, что позволяет создать мосты от нескольких контейнеров к одному localhost и тем самым сделать его единым для них, что активно используется в POD Kubernetes.

Запуск временного контейнера в интерактивном режиме -it. Для входа нужно нажать Ctrl +D, которое отравит сигнал на завершение работы, после чего он будет удалён –rm во избежание засорения системы остановленными современными контейнерами. Если образ создан таким образом, что в контейнере приложение запускается в оболочке, что неправильно, то сигнал буде отравлен приложению, а контейнер продолжит работать с оболочкой, в таком случае для выхода в отдельном терминале нужно будет его убить по его имени –name name_container. Например,:

Docker run –rm -it –name name_container ubuntu BASH

В начале CLI Docker имел простой набор команд, позволяющий управлять жизненным циклом контейнеров. Среди них:

* Docker run для запуска контейнера;
* Docker ps для просмотра запущенных контейнеров;
* Docker rm для удаления контейнера;
* Docker build для создания своего образа;
* Docker images для просмотра существующих контейнеров;
* Docker rmi для удаления образа.

Но с ростом популярности, команд становилось всё больше и их было решено сгруппировать в группы, так вместо простого "Docker run" появилась команда "Docker container", которая имеет 25 команд в 19 версии Docker. Это очистка, и остановка с восстановлением, и логи и

различные виды подсоединений к контейнеру. Таже судьба постигла и работы с образами. Но, старые команды остались пока из-за совместимости и удобства, ведь в большинстве случаев требуется базовый набор. На нём и остановимся:

Запуск контейнера:
docker run -d –name name_container ubuntu bash
Удалить работающий контейнер:
docker rm -f name_container
Вывод всех контейнеров:
docker ps -a
Вывод работающих контейнеров:
docker ps
Вывод контейнеров с потребляемыми ресурсами:
docker stats
Вывод процессов в контейнере:
docker top {name_container}
Подключиться к контейнеру через оболочку sh (BASH в контейнерах alpine нет):
docker exec -it sh
Чистка системы от неиспользуемых образов:
docker image prune
Удалить весящие образа:
docker rmi $(docker images -f "dangling=true" -q)
Показать образа:
docker images
Создать образ в папке dir с Dockerfile:
docker build -t docker_user/name_image dir
Удалить образ:
docker rmi docker_user/name_image dir
Подключиться к Docker hub:
docker login
Отравить последнюю редакцию (тэг ставится и смещается автоматически, если не указан иной) образ на Docker hub:
docker push ocker_user/name_image dir:latest
Более широкий список в https://niqdev.github.io/devops/docker/.
Создание Docker Machine можно описать следующими этапами:
Создание виртуальной машины VirtualBox
docker-machine create name_virtual_system
Создание виртуальной машины generic
docker-machine create -d generic name_virtual_system
Список виртуальных машин:
docker-machine ls
Остановить виртуальную машину:
docker-machine stop name_virtual_system
Запустить остановленную виртуальную машину:
docker-machine start name_virtual_system
Удалить виртуальную машину:
docker-machine rm name_virtual_system
Подключиться к виртуальной машине:
eval "$(docker-machine env name_virtual_system)"
Отключить Docker от виртуальной машины:

eval $(docker-machine env -u)
Зайти по SSH:
docker-machine ssh name_virtual_system
Выйти из виртуальной машины:
exit
Запуск команды sleep 10 в виртуальной машине:
docker-machine ssh name_virtual_system 'sleep 10'
Запуск команд в среде BASH:
docker-machine ssh dev 'bash -c "sleep 10 && echo 1"'
Скопировать папку dir в вируальную машину:
docker-machine scp -r /dir name_virtual_system:/dir
Сделать запрос к контейнерам виртуальной машины:
curl $(docker-machine ip name_virtual_system):9000
Пробросить порт 9005 хостовой машины на 9005 виртуальной машины
docker-machine ssh name_virtual_system -f -N -L 9005:0.0.0.0:9007
Инициализация мастера:
docker swarm init
Запуск множества контейнеров с одинаковыми EXPOSE:
essh@kubernetes-master:~/mongo-rs$ docker run –name redis -p 6379 -d redis
f3916da35b6ba5cd393c21d5305002b78c32b089a6cc01e3e2425930c9310cba
essh@kubernetes-master:~/mongo-rs$ docker ps | grep redis
f3916da35b6b redis"docker-entrypoint.s…" 8 seconds ago Up 6 seconds 0.0.0.0:32769->6379/tcp redis
essh@kubernetes-master:~/mongo-rs$ docker port reids
Error: No such container: reids
essh@kubernetes-master:~/mongo-rs$ docker port redis
6379/tcp –> 0.0.0.0:32769
essh@kubernetes-master:~/mongo-rs$ docker port redis 6379
0.0.0.0:32769
Сборка – первое решение скопировать все файлы и установить. В результате при изменении любого файла будет производится переустановка всех пакетов:

```
COPY ./ /src/app
WORKDIR /src/app
RUN NPM install
```

Воспользуемся кэшированием и разделим статические файлы и установку:

```
COPY ./package.json /src/app/package.json
WORKDIR /src/app
RUN npm install

COPY . /src/app
```

Использование шаблона базового образа node:7-onbuild:

```
$ cat Dockerfile
FROM node:7-onbuild
EXPOSE 3000
$ docker build .
```

В таком случае, файлы, которые не нужно включать в образ, такие как системные файлы, например, Dockerfile, .git, .node_modules, файлы с ключами, их нужно внести в node_modules, файлы с ключами, их нужно внести в .dockerignore.
–v /config

docker cp config.conf name_container:/config/

Статистика использованных ресурсов в реальном времени:

essh@kubernetes-master:~/mongo-rs$ docker ps -q | docker stats

CONTAINER ID NAME CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O PIDS

c8222b91737e mongo-rs_slave_1 19.83% 44.12MiB / 15.55GiB 0.28% 54.5kB / 78.8kB 12.7MB / 5.42MB 31

aa12810d16f5 mongo-rs_backup_1 0.81% 44.64MiB / 15.55GiB 0.28% 12.7kB / 0B 24.6kB / 4.83MB 26

7537c906a7ef mongo-rs_master_1 20.09% 47.67MiB / 15.55GiB 0.30% 140kB / 70.7kB 19.2MB / 7.5MB 57

f3916da35b6b redis 0.15% 3.043MiB / 15.55GiB 0.02% 13.2kB / 0B 2.97MB / 0B 4

f97e0697db61 node_api 0.00% 65.52MiB / 15.55GiB 0.41% 862kB / 8.23kB 137MB / 24.6kB 20

8c0d1adc9b9c portainer 0.00% 8.859MiB / 15.55GiB 0.06% 102kB / 3.87MB 57.8MB / 122MB 20

6018b7e3d9cd node_payin 0.00% 9.297MiB / 15.55GiB 0.06% 222kB / 3.04kB 82.4MB / 24.6kB 11

^C

При создании образов нужно учитывать:

** изменение большого слоя он будет пересоздан, поэтому его, часто лучше разделить, например, создать один слой с 'NPM i' и уже на втором скопировать код ;

* если файл в образе большой и контейнер иго изменят, то из слоя образа доступного только для чтения файл будет целиком скопирован в слой для редактирования, поэтому, контейнера предполагаются быть легковесными, а контент принято располагать в специальном хранилище. code-as-a-service: 12 факторов (12factor.net)

* Codebase – один сервис – они репозиторий;

* Dependeces – все зависимые сервисы в конфиге;

* Config – конфиги доступны через среду;

* BackEnd – обмениваются данными с другими сервисами через сеть на основе API;

* Processes – один сервис – одни процесс, что позволяет в случае падения однозначно отслеживать (завершается сам контейнер) и перезапускать его;

* Независимость о окружения и не влияние на него.

* CI/CD – code control (git) – build (jenkins, GitLab) – relies (Docker, jenkins) – deploy (helm, Kubernetes). Поддержание легковесности сервиса важно, но есть программы, не предназначенные для запуска в контейнерах, таки как базы данных. Из-за своей особенности к их запуску предъявляются определённые требования, а профит ограничен. Так, из-за больших данных они не просто медленно масштабируется, а ролинг-абдейт маловероятен, при этом перезапуск необходимо производить на тех же нодах, что и их данные из соображений производительности доступа к ним.

* Config – взаимоотношения сервисов определённо в конфигурации, например, docker-compose.yml;

* Port bindign – общение сервисов происходит через порты, при этом порт может выбираться автоматически, например, если в Dockerfile указан EXPOSE PORT, то при вызове контейнера с флагом -P он будет прикончен к свободному автоматически.

* Env – настройки среды предаются через переменные окружения, а не через конфиги, что позволяет их вносить в конфигурацию конфига сервисов, например, docker-compose.yml

* Logs – логи передаются потоком по сети, например, ELK, или выводится в вывод, который уже Docker передаёт потоком.

Внутренности Dockerd:

```
essh@kubernetes-master:~/mongo-rs$ ps aux | grep dockerd
root 6345 1.1 0.7 3257968 123640 ? Ssl июл05 76:11 /usr/bin/dockerd -H fd:// –containerd=/run/containerd/containerd.sock
essh 16650 0.0 0.0 21536 1036 pts/6 S+ 23:37 0:00 grep –color=auto dockerd
essh@kubernetes-master:~/mongo-rs$ pgrep dockerd
6345
essh@kubernetes-master:~/mongo-rs$ pstree -c -p -A $(pgrep dockerd)
dockerd(6345)-+-docker-proxy(720)-+-{docker-proxy}(721)
| |-{docker-proxy}(722)
| |-{docker-proxy}(723)
| |-{docker-proxy}(724)
| |-{docker-proxy}(725)
| |-{docker-proxy}(726)
| |-{docker-proxy}(727)
| `-{docker-proxy}(728)
|-docker-proxy(7794)-+-{docker-proxy}(7808)
```

Docker-File:

* чистка кэши от пакетных менеджеров: apt-get, pip и других, этот кэш не нужен на продакшне, лишь занимает место и нагружает сеть, но ныне не зачастую не актуально, так как есть многоэтапные сборки, но об этом ниже.

* группируйте команды одних сущностей, например, получение кэша APT, установку программ и удаление кэша: в одной инструкции – код только программ, при разнесённом варианте – код программ и кэш, так как если не удалить кэш в одной инструкции, то он будет сохранён в слое, не зависимо от последующих действий.

* разделяйте инструкции по частоте изменения, так, например, если не разделить установку программного обеспечения и код, то при изменении чего-либо в коде, то вместо использования готового слоя с программами они будут переустановлены заново, что повлечёт существенное время на подготовку образа, которое критично для разработчиков:

```
ADD ./app/package.json /app
RUN npm install
ADD ./app /app
```

Альтернативы Docker

** Rocket или rkt – контейнеры для операционной среды CoreOS от RedHut, специально созданной на использование контейнеров.

** Hyper-V – среда для запуска Docker в операционной системе Windows, представляющая из себя обертку (легковесную виртуальную машину) контейнера.

От Docker ответвились его базовые компоненты, которые используются им как примитивы, ставшие стандартными компонентами для реализации контейнеров, таких как RKT, объединенных в проект containerd:

* CRI-O – OpanSource проект, с самого начала нацеленный на полную поддержку стандартов CRI (Container Runtime Interface), github.com/opencontainers/runtime-spec/">Runtime Specification и github.com/opencontainers/image-spec">Image Specification как общего интерфейса взаимодействия системы оркестровки с контейнерами. Наряду с Docker, добавлена поддержка CRI-O 1.0 в Kubernetes (речь пойдёт дальше) с версии 1.7 в 2007, а также в MiniKube и Kubic. Имеет реализацию CLI (Common Line Interface) в проекте Pandom, практически полностью повторяющий команды Docker, но без оркестровки (Docker Swarm), который по умолчанию является инструментом в Linux Fedora.

* CRI (Kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-Kubernetes/">Container Runtime Interface) – среда для запуска контейнеров, универсально предоставляющие примитивы (Executor, Supervisor, Metadata, Content, Snapshot, Events и Metrics) для работы с Linux контейнерами (пространствами процессов, групп и т.д.).

** CNI (Container Networking Interface) – работа с сетью.

Portainer

Простейшим вариантом мониторинга будет Portainer:

```
essh@kubernetes-master:~/microKubernetes$ cat << EOF > docker-compose.monitoring.yml
version: '2'
>
services:
portainer:
image: portainer/portainer
command: -H unix:///var/run/Docker.sock
restart: always
ports:
– 9000:9000
volumes:
– /var/run/Docker.sock:/var/run/Docker.sock
– ./portainer_data:/data
>
EOF
```

```
essh@kubernetes-master:~/microKubernetes$ docker-compose -f docker-compose.monitoring.yml up -d
```

Мониторинг с помощью Prometheus

Мониторинг – поддержка непрерывности работы, отслеживание текущей ситуации (выявление, локализация и отправка об инциденте, например, в SaaS PagerDuty), прогнозирование возможных ситуаций, визуализация, построение моделей нормально работы IAOps (Artificial Intelligence For It Operations, https://www.gartner.com/en/information-technology/glossary/aiops-artificial-intelligence-operations).

Мониторинг содержит следующие этапы:

* выявление инцидента;
* уведомление о инциденте;
* локализация;
* решение.

Мониторинг можно классифицировать по уровню на следующие типы:

* инфраструктурный (операционная система, сервера, Kubernetes, СУБД), ;
* прикладной (логи приложений, трейсы, события приложений), ;
* бизнес-процессов (точки в транзакциях, трейсы транзакциях).

Мониторинг можно классифицировать по принципу:

* распределённый (трейсы), ;
* синтетический (доступность), ;
* IAOps (прогнозирование, аномалии).

Мониторинг делится на две части по степени анализа на системы логирования и системы расследования инцидентов. Примером логирования

служит ELK стек, а расследования инцидентов – Sentry (SaaS). Для микро-сервисов добавляется ещё и система трассировки

запросов, такие как Jeger или Zipkin. Система логирования просто пишет все логи, которые доступны.

Система расследования инцидентов пишет гораздо больше информации, но пишет её только в случае ошибок в приложении, например,

параметры окружения, версии установленных пакетов, стек трейс и так далее, что позволяет при просмотре получить максимальную информацию

по ошибке, а не собрать её по кусочкам с сервера и GIT репозитория. Но набор и формат информации зависит от окружения, поэтому

системе инцидентов нужно иметь интеграцию с различными языковыми платформами, а ещё лучше с конкретными фреймворками. Так Sentry

отравляет переменные окружения, участок кода и указание в каком месте произошла ошибка, параметры программного и платформенного

окружения, вызовы методов.

Мониторинг по экосистеме можно разделить на:

* Встроенный в облако Cloud: Azure Monitoring, Amazon CloudWatch, Google Cloud Monitoring
* Предоставляющийся как сервис с поддержкой различных интеграций SaaS: DataDog, NewRelic
* CloudNative: Prometheus
* Для выделенных серверов OnPremis: Zabbix

Zabbix разработан в 1998 год и выведен в OpenSource под лицензией в GPL в 2001. Для того времени, традиционный интерфейс:

без какого-либо дизайна, с большим числом вкладок, селекторов и тому подобного. Так как он разрабатывался для

собственных нужд, то он содержит конкретные решения. Ориентирован он

на мониторинг устройств и их компонентов, таких как диски, сеть, принтеры, роутеры и тому подобного. Для

взаимодействия можно использовать:

Агенты – устанавливаются на сервера, собирают множество метрик и отравляют Zabbix серверу

HTTP – Zabbix делает запросы по http, например, принтеров

SNMP— сетевой протокол для взаимодействия с сетевых устройств

IPMI – протокол для взаимодействия с серверными устройствами, такими, как роутеры

В 2019 году Gratner представил рейтинг систем мониторинга в своём квадрате:

** Dynatrace;

** Cisco (AppDynamics);
** New Relic;
** Broadcom (CA Technologies);
** Riverbed и Microsoft;
** IBM;
** Oracle;
** SolarWinds;
** Micro Focus;
** ManageEngine и Tingyun.
Не вошли в квадрат:
** Correlsense;
** Datadog;
** Elastic;
** Honeycomb;
** Instant;
** Jennifer Soft;
** Light Step;
** Nastel Technologies;
** SignalFx;
** Splunk;
** Sysdig.

Когда мы запускаем приложение в Docker контейнере, весь стандартный вывод (то, что выводится в консоли) запущенной программы (процесса) помещается в буфер. Этот буфер мы можем просмотреть программой docker logs name_container. Если мы следуем идеологии Docker – "один процесс – один контейнер" – мы можем просматривать логи отдельной программы. Для просмотра логов удобно пользоваться командами less и tail. Первая программа позволяет удобно прокручивать лога стрелками клавиатуры, искать нужное с на основе совпадений и по шаблону регулярных выражений, на подобие текстового редактора vi. Вторая программа выводит нужно нам количество

Важным критерием обеспечения бесперебойной работы является контроль свободного места. Так, если места не останется, то база данных не сможет записывать данные, с другими компонентами ситуация может быть более плачевная, чем потеря новых данных. В Docker есть настройки лимитов не только для отдельных контейнеров, минимум 10%. Во время создания образа или запуска контейнера может быть выброшена ошибка о том, что заданные пределы превышены. Для изменения настроек по умолчанию нужно указать серверу Dockerd настройки, предварительно его остановив service docker stop (все контейнера будут остановлены) и после возобновив его работу service docker start (работа контейнеров будет возобновлена). Настройки можно задать как параметры /bin/dockerd –storange-opt dm.basesize=50G –stirange-opt

В Container мы имеем авторизацию, контроль наши контейнеров, с возможностью для теста их создавать и видеть графики по процессору и памяти. Для большего потребуется система мониторинга. Систем мониторинга довольно много, например, Zabbix, Graphite, Prometheus, Nagios, InfluxData, OkMeter, DataDog, Bosum, Sensu и другие, из которых наиболее популярны Zabbix и Prometheus (промисиус). Первый, традиционно применяется, так как является лидирующим средством деплоя, который полюбился админам за простоту использования (достаточно только иметь SSH доступ к серверу), низкоуровненность, позволяющий работать не только с серверами, но и другим железом, таким как роутеры. Второй же является противоположностью первого: он заточен исключительно на сбор метрик и мониторинг, ориентирован как готовое решение, а не фреймворк и полюбился программистам, по прин-

ципу поставил, выбрал метрики и получил графики. Ключевой особенностью между Zabbix и Prometheus заключается не в предпочтениях одних детально настраивать под себя и других затрачивать намного меньше времени, а в сфере применения. Zabbix ориентирован на настройку работы с конкретным железом, которым может быть что угодно, и часто весьма экзотическим в корпоративной среде, и для этой сущности пишется сбор метрик вручную, вручную настраивается график. Для динамически меняющуюся среды облачных решений, даже если это просто контейнера Docker, и тем более, если это Kubernetes, в которой постоянно создаются огромное количество сущностей, а сами сущности в отрыве от общей среды не имеют особого интереса он не подходит, для этого в Prometheus встроен Service Discovery и для Kubernetes поддерживается навигация через название области видимости (namespace), балансера (service) и группы контейнеров (POD), которую можно настроить в Grafana виде таблиц. В Kubernetes, по данным The News Stack 2017 Kubernetes UseriExperience, используется в 63% случаев, в остальных более редкие облачные средства мониторинга.

Метрики бывают системные (например, CRU, RAM, ROM) и прикладные (метрики сервисов и приложений). Системные метрики – метрики ядра, которые используются Kubernetes для масштабирования и тому подобного и метрики non-core, который не используются Kubernetes. Приведу пример связок сбора метрик:

* cAdvisor + Heapster + InfluxDB
* cAdvisor + collectd + Heapster
* cAdvisor + Prometheus
* snapd + Heapster
* snapd + SNAP cluster-level agent
* Sysdig

На рынке много систем мониторинга и сервисов. Мы рассмотрим именно OpenSource, которые можно установить в свой кластер. Их разделить можно по модели получения метрик: на тех, которые забирают логи опрашивая, и на тех, кто ожидает, что в них отравят метрики. Вторые более просты, как по структуре, так и по использования в малых масштабах. Примером может быть InfluxDB, представляющий из себя базу данных, в которую можно писать. Минусом подобного решения является сложность масштабирования как по поддержки, так и по нагрузке. Если все сервисы будут одновременно писать, то они могут перегрузить систему мониторинга тем более, что её сложно масштабировать, так эндпойтн прописан в каждом сервисе. Представителем первой группы, исповедующей pull-модель взаимодействия, является Prometheus. Он также представляет из себя базу данных с демоном, который опрашивает сервисы на основе их регистраций в файле конфигураций и стягивает метки в определённом формате, например:

cpu_usage: 2

cpu_usage{app: myapp} : 2

Prometheus – зрелый продукт, он разработан в 2012, а в 2016 включён в составе консорта CNCF (Cloud Native Computing Foundation). Prometheus состоит из:

* TSDB (Time Series Satabase) базы данных, которая больше напоминает очередь хранения метрик, с заданным периодом накопления, например, недели, позволяющая обрабатывать сотни тысяч метрик в секунду. Данная база локальна для Prometheus, не поддерживает горизонтального масштабирование, в случае с Prometheus оно достигается с помощью поднятием нескольких его инстансев и шардированием их. Prometheus поддерживает агрегацию данных, что полезно для снижения объёма накопленных данных, а также архивирование базы данных из памяти на диск.

* Service Discovery поддерживать Kubernetes в коробке через публичное API через опрашивание POD, отфильтрованных в соответствии с конфигом по 9121 порту TPC.

* Grafana (отдельный продукт, по умолчанию добавляемый) – универсальное UI с дашбордами и графиками, поддерживающее Prometheus через PromQL.

Для отдачи метрик можно воспользоваться готовыми решениями или разработать свои. Для подавляющего большинства системных метрик существуют exporter, а для прикладных, часто приходится отдавать свои метрики. Экспортёры бывают общие и специализированные. Например, NodeExporter предоставляет большинство метрик, в том числе и по процессам, но их два, а специализированный на них – больше. Если запустить Prometheus без экспортёров, то он выдаст почти тысячу метрик, но это метрики самого Prometheus, и там не будет приставок в них node_*. Чтобы появились эти метрики, нужно включить NodeExporter и прописать в конфигурации Prometheus URL к нему, для сбора предоставляемых им метрикам. Для NodeExporter это может быть localhost или адрес ноды и порт 9256. Обычно, экспортёры специализируются на метриках конкретных продуктов, например:

** node_exporter – метрики нод (CRU, Memory, Network);
** snmp_exporter – метрики протокола SNMP;
** mysqld_exporter – метрики базы данных MySQL;
** consul_exporter – метрики базы данных Consul;
** graphite_exporter – метрики базы данных Graphite;
** memcached_exporter – метрики базы данных Memcached;
** haproxy_exporter – метрики балансировщика HAProxy;
** CAdvisor – метрики контейнеров;
** process-exporter – детальные метрики процессов ;
** metrics-server – CRU, Memory, File-descriptors, Disks;
** cAdvisor – a Docker daemon metrics – containers monitoring;
** kube-state-metrics – deployments, PODs, nodes.

Prometheus поддерживает удалённую запись данных (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#remote_write), например в распределённое хранилище TSDB для Prometheus – Weave Works Cortex, используя настройку в конфигурации, что позволяет анализировать данные с нескольких Prometheus:

remote_write:
– url: "http://localhost:9000/receive"

Рассмотрим его работу на готовом инстансе. Я возьму для этого www.katacoda.com/courses/istio/deploy-istio-on-kubernetes и пройду его. Наш Prometheus располагается на стандартном для него порту 9090:

controlplane $ kubectl -n istio-system get svc prometheus
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
prometheus ClusterIP 10.99.70.170 < none> 9090/TCP 6m59s

Чтобы открыть его UI, я перейду на WEB-вкладку и изменю в адресе 80 на 9090: https://2886795314-9090-ollie08.environments.katacoda.com/graph. В строке ввода нужно вводить желаемую метрику на языке PromQL (Prometheus query language), также как и InfluxQL для InfluxDB и SQL для TimescaleDB. Для примера я введу «CRU», и он мне отобразит список его содержащий. Под строкой содержатся две вкладки: вкладка с графиком и вкладка для отображения в табличном виде. Я буду смотреть на табличное представление. Я выбрал machine_cru_cores и нажал Execute. Распространённые метрики, обычно имеют схожие названия, например, machine_cru_cores и node_cru_cores. Сами метрики состоят из названия, тегов в скобочках и значения метрики, в таком же виде их и нужно запрашивать, в таком же виде они и отображаются в таблице.

machine_cpu_cores{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",instance="co
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux"
2

machine_cpu_cores{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",instance="no
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="node01",kubernetes_io_os="linux"}
2

Если в сети MEMORY – то можно выбрать machine_memory_bytes – размер оперативной памяти на машине (сервере или виртуальной):

machine_memory_bytes{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",instance=
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux"
2096992256

machine_memory_bytes{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",instance=
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="node01",kubernetes_io_os="linux"}
4092948480

Но в байтах ненаглядно, поэтому воспользуемся PromQL для перевода в Gb: machine_memory_bytes / 1000 / 1000 / 1000

{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",instance="controlplane",job="ku
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux"
2.096992256

{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",instance="node01",job="kuberne
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="node01",kubernetes_io_os="linux"}
4.09294848

Введём для memory_bytes для поиска container_memory_usage_bytes – использованной памяти. Список содержит все контейнера и текущее потребление ими памяти, я приведу только три:

container_memory_usage_bytes{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",c
kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-
pod0e619e5dc53ed9efcef63f5fe1d7ee71.slice/docker-
b6549e892baa8687e4e98a106024b5c31a4af077d7c5544af03a3c72ec8997e0.scope",image="k8s.gcr.io/
pause:3.1",instance="controlplane",job="kubernetes-
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux",
controlplane_kube-system_0e619e5dc53ed9efcef63f5fe1d7ee71_0",namespace="kube-
system",pod="etcd-controlplane",pod_name="etcd-controlplane"} 45056

container_memory_usage_bytes{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",c
kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-
pod5a815a40_f2de_11ea_88d2_0242ac110032.slice/
docker-76711789af076c8f2331d8212dad4c044d263c5cc3fa333347921bd6de7950a4.scope",image="k8s.g
pause:3.1",instance="controlplane",job="kubernetes-
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux",
proxy-nhzhn_kube-system_5a815a40-f2de-11ea-88d2-0242ac110032_0",namespace="kube-
system",pod="kube-proxy-nhzhn",pod_name="kube-proxy-nhzhn"} 45056

container_memory_usage_bytes{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",c
kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-
pod6473aeea_f2de_11ea_88d2_0242ac110032.slice/
docker-24ef0e898e1bb7dec9854b67291171aa9c5715d7683f53bdfc2cef49a19744fe.scope",image="k8s.gcr
pause:3.1",instance="node01",job="kubernetes-
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="node01",kubernetes_io_os="linux",name
proxy-6v49x_kube-system_6473aeea-f2de-11ea-88d2-0242ac110032_0",namespace="kube-
system",pod="kube-proxy-6v49x",pod_name="kube-proxy-6v49x"} 835584

Выставим метку, которая содержится в метриках, чтобы отфильтровать один: container_memory_usage_bytes{container_name="prometheus"}

container_MEMORY_usage_bytes{beta_Kubernetes_io_arch="amd64",beta_Kubernetes_io_os="linu
kubePODs.slice/kubePODs-burstable.slice/kubePODs-burstable-
PODeaf4e833_f2de_11ea_88d2_0242ac110032.slice/Docker-
b314fb5c4ce8894f872f05bdd524b4b7d6ce5415aeb3fb91d6048441c47584a6.scope",image="sha256:b82ef
cadvisor",Kubernetes_io_arch="amd64",Kubernetes_io_hostname="node01",Kubernetes_io_os="linux",nam
knf44_istio-system_eaf4e833-f2de-11ea-88d2-0242ac110032_0",namespace="istio-
system",POD="prometheus-5b77b7d695-knf44",POD_name="prometheus-5b77b7d695-knf44"}

283443200

Приведём в Mb: container_memory_usage_bytes {container_name="prometheus"} / 1000 / 1000

{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",container="prometheus",containe
kubepods.slice/kubepods-burstable.slice/kubepods-burstable-
podeaf4e833_f2de_11ea_88d2_0242ac110032.slice/docker-
b314fb5c4ce8894f872f05bdd524b4b7d6ce5415aeb3fb91d6048441c47584a6.scope",image="sha256:b82ef
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="node01",kubernetes_io_os="linux",name
knf44_istio-system_eaf4e833-f2de-11ea-88d2-0242ac110032_0",namespace="istio-
system",pod="prometheus-5b77b7d695-knf44",pod_name="prometheus-5b77b7d695-knf44"}

286.18752

Отфильтруем по container_memory_usage_bytes{container_name="prometheus", instance="node01"}

beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",container="prometheus",container
kubepods.slice/kubepods-burstable.slice/kubepods-burstable-
podeaf4e833_f2de_11ea_88d2_0242ac110032.slice/docker-
b314fb5c4ce8894f872f05bdd524b4b7d6ce5415aeb3fb91d6048441c47584a6.scope",image="sha256:b82ef
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="node01",kubernetes_io_os="linux",name
knf44_istio-system_eaf4e833-f2de-11ea-88d2-0242ac110032_0",namespace="istio-
system",pod="prometheus-5b77b7d695-knf44",pod_name="prometheus-5b77b7d695-knf44"}

289.890304

А на второй его нет: container_memory_usage_bytes{container_name="prometheus", instance="node02"}

no data

Есть и агрегатные функции sum(container_memory_usage_bytes) / 1000 / 1000 / 1000

{} 22.812798976

max(container_memory_usage_bytes) / 1000 / 1000 / 1000

{} 3.6422983679999996

min(container_memory_usage_bytes) / 1000 / 1000 / 1000

{} 0

Можно и сгруппировать по меткам instance: max(container_memory_usage_bytes) by (instance) / 1000 / 1000 / 1000

{instance="controlplane"} 1.641836544

{instance="node01"} 3.6622745599999997

Можно производить действия с однотипными метками и отфильтровывать: container_memory_mapped_file / container_memory_usage_bytes * 100 > 80

{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",container="POD",container_nam
kubepods.slice/kubepods-burstable.slice/kubepods-burstable-
pode45f10af1ae684722cbd74cb11807900.slice/

docker-5cb2f2083fbc467b8b394b27b69686d309f951450bcb910d509572aea9922806.scope",image="k8s.g
pause:3.1",instance="controlplane",job="kubernetes-
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux",
controller-manager-controlplane_kube-
system_e45f10af1ae684722cbd74cb11807900_0",namespace="kube-system",pod="kube-
controller-manager-controlplane",pod_name="kube-controller-manager-controlplane"}

80.52631578947368

Посмотреть на метрики файловой системы можно с помощью container_fs_limit_bytes, который выдаёт большой список – приведу несколько из него:

container_fs_limit_bytes{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",containe
dev/vda1",id="/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-
pod0e619e5dc53ed9efcef63f5fe1d7ee71.slice/docker-
b6549e892baa8687e4e98a106024b5c31a4af077d7c5544af03a3c72ec8997e0.scope",image="k8s.gcr.io/
pause:3.1",instance="controlplane",job="kubernetes-
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux",
controlplane_kube-system_0e619e5dc53ed9efcef63f5fe1d7ee71_0",namespace="kube-
system",pod="etcd-controlplane",pod_name="etcd-controlplane"}

253741748224

container_fs_limit_bytes{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",containe
dev/vda1",id="/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-
pod5a815a40_f2de_11ea_88d2_0242ac110032.slice/
docker-76711789af076c8f2331d8212dad4c044d263c5cc3fa333347921bd6de7950a4.scope",image="k8s.g
pause:3.1",instance="controlplane",job="kubernetes-
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux",
proxy-nhzhn_kube-system_5a815a40-f2de-11ea-88d2-0242ac110032_0",namespace="kube-
system",pod="kube-proxy-nhzhn",pod_name="kube-proxy-nhzhn"}

253741748224

В нём присутствую метрики оперативной памяти через его устройство: "container_fs_limit_bytes{device="tmpfs"} / 1000 / 1000 / 1000"

{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",device="tmpfs",id="/",instance="
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="controlplane",kubernetes_io_os="linux"
0.209702912

{beta_kubernetes_io_arch="amd64",beta_kubernetes_io_os="linux",device="tmpfs",id="/",instance="
cadvisor",kubernetes_io_arch="amd64",kubernetes_io_hostname="node01",kubernetes_io_os="linux"}
0.409296896

Если мы хотим получить минимальный диск, то нам нужно из списка убрать устройство оперативной памяти: "min(container_fs_limit_bytes{device!="tmpfs"} / 1000 / 1000 / 1000)"

{} 253.74174822400002

Кроме метрик, указывающие само значение метрики, есть метрики счётчики. Их название, обычно, заканчиваются на "_total". Если их посмотреть, то мы увидим возрастающую линию. Чтобы получить значение, нам нужно получить разницу (с помощью функции rate) за период времени (указывается в квадратных скобках), примерно так rate(name_metric_total) [time]. Время, обычно ведётся в секундах или минутах. Для обозначения секунд используются приставка "s", например, 40s, 60s. Для минут – "m", например, 2m, 5m. Важно заметить, что

нельзя устанавливать время, меньшее времени опроса exporter, иначе метрика не будет отображаться.

А посмотреть имена метрик, которые смог записать можно по пути /metrics:

controlplane $ curl https://2886795314-9090-ollie08.environments.katacoda.com/metrics 2>/dev/null | head

HELP go_gc_duration_seconds A summary of the GC invocation durations.
TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 3.536e-05
go_gc_duration_seconds{quantile="0.25"} 7.5348e-05
go_gc_duration_seconds{quantile="0.5"} 0.000163193
go_gc_duration_seconds{quantile="0.75"} 0.001391603
go_gc_duration_seconds{quantile="1"} 0.246707852
go_gc_duration_seconds_sum 0.388611299
go_gc_duration_seconds_count 74
HELP go_goroutines Number of goroutines that currently exist.

Поднятие связки Prometheus и Graphana

Мы рассмотрели метрики в уже настроенном Prometheus, теперь поднимем Prometheus и настроим его сами:

essh@kubernetes-master:~$ docker run -d –net=host –name prometheus prom/prometheus
09416fc74bf8b54a35609a1954236e686f8f6dfc598f7e05fa12234f287070ab

essh@kubernetes-master:~$ docker ps -f name=prometheus
CONTAINER ID IMAGE NAMES
09416fc74bf8 prom/prometheus prometheus

UI с графиками по отображению метрик:

essh@kubernetes-master:~$ firefox localhost:9090

Добавим метрику go_gc_duration_seconds{quantile="0"} из списка:

essh@kubernetes-master:~$ curl localhost:9090/metrics 2>/dev/null | head -n 4
HELP go_gc_duration_seconds A summary of the GC invocation durations.
TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 1.0097e-05
go_gc_duration_seconds{quantile="0.25"} 1.7841e-05

Зайдя в UI по адресу localhost:9090 в меню выберем Graph. Добавим в дашборд с графиком: выбираем метрику с помощью списка – insert metrics at cursor. Здесь мы видим те же метрики, что и в списке localhost:9090/metrics, но агрегированные по параметрам, например, просто go_gc_duration_seconds. Мы выбираем метрику go_gc_duration_seconds и покажем её по кнопке Execute. Во вкладке console дашборда видим метрики:

go_gc_duration_seconds{instance="localhost:9090",JOB="prometheus",quantile="0"}
0.000009186
go_gc_duration_seconds{instance="localhost:9090",JOB="prometheus",quantile="0.25"}
0.000012056
go_gc_duration_seconds{instance="localhost:9090",JOB="prometheus",quantile="0.5"}
0.000023256
go_gc_duration_seconds{instance="localhost:9090",JOB="prometheus",quantile="0.75"}
0.000068848
go_gc_duration_seconds{instance="localhost:9090",JOB="prometheus",quantile="1"}
0.00021869

, перейдя во кладку Graph – графическое их представление.

Сейчас Prometheus собирает метрики с текущей ноды: go_*, net_*, process_*, prometheus_*, promhttp_*, scrape_* и up. Для сбора метрик с Docker кажем ему писать его метрики в Prometheus по порту 9323:

```
eSSH@Kubernetes-master:~$ curl http://localhost:9323/metrics 2>/dev/null | head -n 20
# HELP builder_builds_failed_total Number of failed image builds
# TYPE builder_builds_failed_total counter
builder_builds_failed_total{reason="build_canceled"} 0
builder_builds_failed_total{reason="build_target_not_reachable_error"} 0
builder_builds_failed_total{reason="command_not_supported_error"} 0
builder_builds_failed_total{reason="Dockerfile_empty_error"} 0
builder_builds_failed_total{reason="Dockerfile_syntax_error"} 0
builder_builds_failed_total{reason="error_processing_commands_error"} 0
builder_builds_failed_total{reason="missing_onbuild_arguments_error"} 0
builder_builds_failed_total{reason="unknown_instruction_error"} 0
# HELP builder_builds_triggered_total Number of triggered image builds
# TYPE builder_builds_triggered_total counter
builder_builds_triggered_total 0
# HELP engine_daemon_container_actions_seconds The number of seconds it takes to process each container action
# TYPE engine_daemon_container_actions_seconds histogram
engine_daemon_container_actions_seconds_bucket{action="changes",le="0.005"} 1
engine_daemon_container_actions_seconds_bucket{action="changes",le="0.01"} 1
engine_daemon_container_actions_seconds_bucket{action="changes",le="0.025"} 1
engine_daemon_container_actions_seconds_bucket{action="changes",le="0.05"} 1
engine_daemon_container_actions_seconds_bucket{action="changes",le="0.1"} 1
```

Чтобы демон докера применил параметры, его нужно перезапустить, что приведёт к падению всех контейнеров, а при старте демона контейнера будут подняты в соответствии с их политикой:

```
essh@kubernetes-master:~$ sudo chmod a+w /etc/docker/daemon.json
essh@kubernetes-master:~$ echo '{ "metrics-addr" : "127.0.0.1:9323", "experimental" : true }' | jq -M -f /dev/null > /etc/docker/daemon.json
essh@kubernetes-master:~$ cat /etc/docker/daemon.json
{
"metrics-addr": "127.0.0.1:9323",
"experimental": true
}

essh@kubernetes-master:~$ systemctl restart docker
```

Prometheus только отреагирует метрики на одном сервере от разных источников. Для того, чтобы мы могли собирать метрики с разных нод и видеть агрегированный результат, на каждую ноду нужно поставить агента, собирающего метрики:

```
essh@kubernetes-master:~$ docker run -d \
–v "/proc:/host/proc" \
–v "/sys:/host/sys" \
–v "/:/rootfs" \
–-net="host" \
–-name=explorer \
quay.io/prometheus/node-exporter:v0.13.0 \
```

```
–collector.procfs /host/proc \
–collector.sysfs /host/sys \
–collector.filesystem.ignored-mount-points "^/(sys|proc|dev|host|etc)($|/)"
1faf800c878447e6110f26aa3c61718f5e7276f93023ab4ed5bc1e782bf39d56
```

и прописать слушать адрес ноды, а пока у нас всё локально, localhost:9100. Теперь сообщим Prometheus слушать агента и докера:

```
essh@kubernetes-master:~$ mkdir prometheus && cd $_

essh@kubernetes-master:~/prometheus$ cat << EOF > ./prometheus.yml
global:
scrape_interval: 1s
evaluation_interval: 1s

scrape_configs:
– job_name: 'prometheus'

static_configs:
– targets: ['127.0.0.1:9090', '127.0.0.1:9100', '127.0.0.1:9323']
labels:
group: 'prometheus'
EOF

essh@kubernetes-master:~/prometheus$ docker rm -f prometheus
prometheus

essh@kubernetes-master:~/prometheus$ docker run \
–d \
–-net=host \
–-restart always \
–-name prometheus \
–v $(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml
prom/prometheus
7dd991397d43597ded6be388f73583386dab3d527f5278b7e16403e7ea633eef

essh@kubernetes-master:~/prometheus$ docker ps \
–f name=prometheus
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
7dd991397d43 prom/prometheus "/bin/prometheus –c…" 53 seconds ago Up 53 seconds prometheus
```

Теперь доступно 1702 метрики хоста:

```
essh@kubernetes-master:~/prometheus$ curl http://localhost:9100/metrics | grep -v '#' | wc -l
1702
```

из всего разнообразия сложно искать нужные для повседневных задач, например, используемое количество памяти node_memory_Active. Для этого есть агрегаторы метрик:

```
http://localhost:9090/consoles/node.html
http://localhost:9090/consoles/node-cpu.html
```

Но лучше использовать Grafana. Установим и её, пример можно посмотреть:

```
essh@kubernetes-master:~/prometheus$ docker run \
```

```
–d \
–-name=grafana \
–-net=host
grafana/grafana
Unable to find image 'grafana/grafana:latest' locally
latest: Pulling from grafana/grafana
9d48c3bd43c5: Already exists
df58635243b1: Pull complete
09b2e1de003c: Pull complete
f21b6d64aaf0: Pull complete
719d3f6b4656: Pull complete
d18fca935678: Pull complete
7c7f1ccbce63: Pull complete
Digest:
sha256:a10521576058f40427306fcb5be48138c77ea7c55ede24327381211e653f478a
Status: Downloaded newer image for grafana/grafana:latest
6f9ca05c7efb2f5cd8437ddcb4c708515707dbed12eaa417c2dca111d7cb17dc

essh@kubernetes-master:~/prometheus$ firefox localhost:3000
```

Введем логин admin и пароль admin, после чего нам предложат изменить пароль. Далее нужно выполнить последующую настройку.

В Grafana первоначальный вход по логину admin и такому паролю. Сперва на предлагается выбрать источник – выбираем Prometheus, вводим localhost:9090, выбираем подключение не как к серверу, а как к браузеру (то есть по сети) и выбираем, что аутентификация у нас базовая – все – нажимаем Save and Test и Prometheus подключен.

Понятно, что всем раздавать пароль и логин от админских прав не стоит. Для этого нужно будет завести пользователей или интегрировать их внешней базой данных пользователей, такой как Microsoft Active Directory.

Я выберу во вкладке Dashboard активирую все три перенастроенных дашборда. Из списка New Dashboard верхнего меню выберу дашборд Prometheus 2.0 Stats. Но, данных нет:

Кликну на пункт меню "+" и выберу "Dashboard", предлагается создать дашборд. Дашборд может содержать несколько виджетов, например графики, которые можно располагать и настраивать, поэтому нажимаем на кнопку добавления графика и выбираем его тип. На самом графике выбираем редактировать, выбрав размер, нажимаем редактировать, и тут самое главное – выбор демонстрируемой метрики. Выбираем Prometheus

Полная сборка доступна:

```
essh@kubernetes-master:~/prometheus$ wget \
https://raw.githubusercontent.com/grafana/grafana/master/devenv/docker/ha_test/docker-compose.yaml

--2019-10-30 07:29:52– https://raw.githubusercontent.com/grafana/grafana/master/devenv/docker/ha_test/docker-compose.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)… 151.101.112.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.112.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 2996 (2,9K) [text/plain]
Saving to: ‘docker-compose.yaml’
```

docker-compose.yaml 100%[==========>] 2,93K –.-KB/s in 0s

2019-10-30 07:29:52 (23,4 MB/s) – ‘docker-compose.yaml’ saved [2996/2996]

Получение прикладных метрик приложения

До этого момента мы рассматривали случай, когда Prometheus опрашивал стандартный накопитель метрик, получая стандартные метрики. Теперь попробуем создать приложение и отдавать свои метрики. Для начала возьмём сервер NodeJS и напишем под него приложение. Для этого, создадим проект NodeJS:

vagrant@ubuntu:~$ mkdir nodejs && cd $_

vagrant@ubuntu:~/nodejs$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install < pkg> –save` afterwards to install a package and
save it as a dependency in the package.json file.

name: (nodejs)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: ESSch
license: (ISC)
About to write to /home/vagrant/nodejs/package.json:

{
"name": "nodejs",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1"
},
"author": "ESSch",
"license": "ISC"
}

Is this ok? (yes) yes

Для начала создадим WEB-сервер. Я воспользуюсь библиотекой для его создания:

vagrant@ubuntu:~/nodejs$ npm install Express –save

npm WARN deprecated Express@3.0.1: Package unsupported. Please use the express package (all lowercase) instead.

nodejs@1.0.0 /home/vagrant/nodejs

```
└─── Express@3.0.1
npm WARN nodejs@1.0.0 No description
npm WARN nodejs@1.0.0 No repository field.

vagrant@ubuntu:~/nodejs$ cat << EOF > index.js
const express = require('express');
const app = express();
app.get('/healt', function (req, res) {
res.send({status: "Healt"});
});
app.listen(9999, () => {
console.log({status: "start"});
});
EOF

vagrant@ubuntu:~/nodejs$ node index.js &
[1] 18963
vagrant@ubuntu:~/nodejs$ { status: 'start' }

vagrant@ubuntu:~/nodejs$ curl localhost:9999/healt
{"status":"Healt"}
```

Наш сервер готов к работе с Prometheus. Нам нужно настроить Prometheus на него.

Проблема масштабирования Prometheus возникает, когда данные не помещаются на один сервер, точнее, когда один сервер не успевает записывать данные и когда обработка данных одним сервером не устраивает по перформансу. Thanos решает эту проблему, не требуя настройки федерации, предоставляя пользователю интерфейс и API, которые он транслирует на инстансы Prometheus. Пользователем доступен веб-интерфейс, аналогичный Prometheus. Сам он взаимодействует с агентами, которые установлены на инстансах как side-car, как это делает Istio. Он и агенты доступны как контейнера и как Helm-чарт. Например, агент может быть поднят как контейнер, настроенный на Prometheus, а Prometheus настраивается конфигом с последующей перезагрузкой.

```
docker run –rm quay.io/thanos/thanos:v0.7.0 –help
docker run -d –net=host –rm \
–v $(pwd)/prometheus0_eu1.yml:/etc/prometheus/prometheus.yml \
–-name prometheus-0-sidecar-eu1 \
–u root \
quay.io/thanos/thanos:v0.7.0 \
sidecar \
–-http-address 0.0.0.0:19090 \
–-grpc-address 0.0.0.0:19190 \
–-reloader.config-file /etc/prometheus/prometheus.yml \
–-prometheus.url http://127.0.0.1:9090
```

Важной составляющей мониторинга являются уведомления. Уведомления состоят из триггеров срабатывания и провайдера. Триггер срабатывания пишется на PromQL как правило с условием в Prometheus. Когда сработал триггер (условие по метрике), Prometheus сигнализирует провайдеру отправить уведомление. Стандартным провайдером является Alertmanager и он способен отравлять сообщения в различные приёмники, такие как электронная почта и Slack.

, Например, метрика "up", принимающая значения 0 или 1, может быть использована, чтобы отравлять сообщение, если сервер выключен более 1 минуты. Для этого записывается правило:

groups:
– name: example
rules:
– alert: Instance Down
expr: up == 0
for: 1m

Когда метрика более 1 минуты равняется 0, то срабатывает этот триггер и Prometheus отравляет запрос на Alertmanager. В Alertmanager прописано, что делать с этим событием. Мы можем прописать, что при получении события InstanceDown нужно отравлять сообщение на почту. Для этого сконфигурируем Alertmanager это сделать:

global:
smtp_smarthost: 'localhost:25'
smtp_from: 'youraddress@example.org'
route:
receiver: example-email
receivers:
– name: example-email
email_configs:
– to: 'youraddress@example.org'

Сам Alertmanager воспользуется установленным протоколом на этом компьютере. Для того, чтобы он смог это сделать, его нужно установить. Возьмём, к примеру, протокол SMTP (Simple Mail Transfer Protocol). Для проверки, установим консольный почтовый сервер в параллель с Alert Manager – sendmail.

Быстрый и наглядный анализ логов системы

Для быстрого поиска по логам используется OpenSource движок полнотекстового поиска Lucene. На его основе были построены два низкоуровневых продукта: Sold и Elasticsearch, довольно сходных по возможностям, но отличающихся по удобству использования и лицензии. На их построены многие популярные сборки, например, просто набор поставки с ElasticSearch: ELK (Elasticsearch(Apache Lucene), Logstash, Kibana), EFK (Elasticsearch, Fluentd, Kibana), так и продукты, например, GrayLog2. Как GrayLog2, так и сборки (ELK/EFK) активно используются из-за меньшей необходимости настраивать не тестовых стендах, так, например, поставить EFK в кластере Kubernetes можно практически одной командой

helm install efk-stack stable/elastic-stack –set logstash.enabled=false –set fluentd.enabled=true –set fluentd-elastics

Альтернативой, пока не получившей большого рассмотрения являются системы, построенные на рассмотренном ранее Prometheus, например, PLG (Promtail(агент) – Loki(Prometheus) – Grafana).

Сравнение ElasticSearch и Sold (системы сопоставимы):

Elastic:

** Коммерческий с открытым кодом и возможность коммитить (через апрув);

** Поддерживает более сложные запросы, больше аналитики, поддержка из коробки распределенных запросов, более полный REST-full JSON-BASH, чейнинг, машинное обучение, SQL (платный);

*** Full-text search;

*** Real-time index;

*** Мониторинг (платный);
*** Мониторинг через Elastic FQ;
*** Машинное обучение (платно);
*** Простая индексация;
*** Больше типов данных и структур;
** Движок Lucene;
** Parent-child (JOIN);
** Scalable native;
** Документация с 2010;
Solr:
** OpenSource;
** Большая скорость при JOIN;
*** Full-text search;
*** Real-time index;
*** Мониторинг в админке;
*** Машинное обучение через модули;
*** Входные данные: Work, PDF и другие;
*** Необходима схемы для индексации;
*** Данные: вложенные объекты;
** Движок Lucene;
** JSON join;
** Scalable: Solar Cloud (настройка) && ZooKeeper (настройка);
** Документация с 2004.

В нынешнее время всё чаще применяется микро сервисная архитектура, которая позволяет за счёт слабой

связанности между своими компонентами и их простоты упростить их разработку, тестирование и отладку.

Но в целом систему из-за её распределённости становится сложнее анализировать. Для анализа состояния

в целом применяются логи, собираемые в централизованное место и преобразуемые в понятный вид. Также возникает

необходимость в анализе других данных, например, access_log NGINX, для сбора метрик о посещаемости, лога почты,

почтового сервера для выявления попытки подбора пароля и т.д. Примером такого решения возьмём ELK. Под ELK понимается

связка трёх продуктов: Logstash, Elasticsearch и Kubana, первый и последний из который сильно заточены на центральный и

обеспечивают удобство работы. Более обобщённо ELK называют Elastic Stack, так как инструмент подготовки логов Logstash

может быть заменён аналогом, например Fluentd или Rsyslog, а средство визуализации Kibana – на Grafana. Например, хоть и

Kibana предоставляет большие возможности для анализа, Grafana предоставляет отправку уведомлений при возникновениях событий, и

может использоваться совместно с другими продуктами, например, CAdVisor – анализа состояния системы и отдельных контейнеров.

Продукты EKL могут быть установлены самостоятельно, скачаны в виде самодостаточных контейнеров, для которых нужно настроить

связь или в виде одного контейнера.

Для нормальной работы Elasticsearch нужно, чтобы данные приходили в формате JSON. Если данные предавать в

текстовом формате (лог пишется одной строкой, отделяется от предыдущий переносом строки), то он сможет

предоставить только полнотекстовый поиск, так как они будут восприниматься одной строкой. Для передачи

логов в формате JSON имеется два варианта: или настроить исследуемый продукт выдавать в этом формате,

например, для NGINX имеется такая возможность. Но, зачастую это невозможно, так как имеется уже

накопленная база логов, а традиционно они пишутся в текстовом формате. Для таких случаев необходима

пост обработка логов из текстового формата в JSON, которой занимается Logstash. Важно заметить, что если

есть возможность сразу же передавать данные в структурированном виде (JSON, XML и других), то следует это

сделать, так как если сделать детальный парсинг, то любое отклонение одностороннее отклонение от формата

приведёт с неработоспособности, а при поверхностном – теряем ценную информацию. В любом случае парсинг в

этот системе является узким горлышком, хотя может ограниченно масштабироваться до сервиса или лога

файла. К счастью, всё больше и больше продуктов начинают поддерживать структурированные логи, например,

последние версии NGINX поддерживает логи в JSON формате.

Для систем, не поддерживающих данный формат можно использовать преобразование к нему с помощью таких

программ, как Logstash, File bear и Fluentd. Первый из них входит в стандартную поставку Elastic Stack от вендора

и может быть установлен одним образом ELK в Docker – контейнер. Он поддерживает получение данных от файлов, сети и

стандартного потока как на входе, так и на выходе, а главное нативно внутренний протокол Elastic Search.

Logstash мониторит log-файлы на основе даты изменения или получает по сети данные по telnet от распределённой

системы, например, контейнеров и после преобразования отравляет на выход, обычно, в Elastic Search. Он прост и

входит в стандартную поставку с Elastic Stack, благодаря чему просто и беспроблемно настраивается. Но благодаря

Java машине внутри тяжёл и не сильно функционален, хотя и поддерживает плагины, например, синхронизации с MySQL

для отправки новых данных. Чуть больше возможностей предоставляет Filebeat. Энтерпрайзным инструментом на все

случаи жизни может служить Fluentd благодаря высокой функциональности (чтение логов, системных журналов и т.д.),

масштабируемости и возможности раскатки по кластерам Kubernetes с помощью Helm чарта, и мониторинг всего

дата-центра в стандартной поставке, но об этом соответствующем разделе.

Для управления логов можно воспользоваться Curator, который сможет архивировать из ElasticSearch старые
логи или их удалять, повышая эффективность его работы.

Процесс получения логов логичен осуществляется специальными сборщиками: logstash, fluentd, filebeat или
другими.
fluentd наименее требовательный и более простой аналог Logstash. Настройка
производится в /etc/td-agent/td-agent.conf, который содержит четыре блока:
** match – содержит настройки передачи полученных данных;
** include – содержит информацию о типах файлов;
** system – содержит настройки системы.
Logstash представляет гораздо более функциональный язык конфигураций. Logstash демон агента – logstash мониторит
изменения в файлах. Если же логи находятся не локально, а на распределённой системе, то устанавливается logstash на каждый сервер и
запускается в режиме агента bin/logstash agent -f /env/conf/my.conf. Поскольку запускать
logstash только в качестве агента для пересылки логов расточительно, то можно использовать продукт от тех
же разработчиков Logstash Forwarder (ранее Lumberjack) пересылающий логи по протоколу lumberjack к
logstash серверу. Для отслеживания и получения данных с MySQL можно использовать агент Packetbeat
(https://www.8host.com/blog/sbor-metrik-infrastruktury-s-pomoshhyu-packetbeat-i-elk-v-ubuntu-14-04/).
Также logstash позволяет преобразовать данные разного типа:
** grok – задать регулярные выражения выдирания полей из строки, часто для логов из текстового формата в JSON;
** date – в случае архивных логов проставить дату создания лога не текущей датой, а взять её из самого лога;
** kv – для логов типа key=value;
** mutate – выбрать только нужные поля и изменить данные в полях, например, произвести замену символа "/" на "_";
** multiline – для многострочных логов с разделителями.
Пример, можно лог в формате "дата тип число" разложить на составляющие, например "01.01.2021 INFO 1" разложить в хэш "message":

```
filter {
grok {
type => "my_log"
match => ["message", "%{MYDATE:date} %{WORD:loglevel} ${ID.id.int}"]
}
}
```

Шаблон ${ID.id.int} берёт класс – шаблон ID, полученное значение будет подставлено в поле id и строковое значение будет преобразовано к типу int.
В блоке «Output» мы можем указать: выводить данные в консоль с помощью блока "Stdout", в файл – "File", передавать по http через JSON REST API – "Elasticsearch" или отрав-

лять по почте – "Email". Также можно заказать условия по полям, полученным в блоке filter. Например,:

```
output {
if [type] == "Info" {
elasticsearch {
host => localhost
index => "log-%{+YYYY.MM.dd}"
}
}
}
```

Здесь индекс Elasticsearch (база данных, если проводить аналогию с SQL) меняется каждый день. Для создания нового индекса не нужно его специально создавать – так поступают БД NoSQL, так как нет жёсткого требования описывать структуру – свойство и тип. Но всё же описать рекомендуется, иначе все поля будут со строковыми значениями, если не задано число. Для отображения данных Elasticsearch используется плагин WEB-ui интерфейса на AngularJS – Kibana. Для отображения временной шкалы в её графиках нужно описать хотя бы одно поле с типом дата, а для агрегатных функция – числовое, будь то целое или с плавающей точкой. Также, если добавляются новые поля, для их индексации и отображения требуется произвести пере индексацию всего индекса, поэтому наиболее полное описание структуры поможет избежать очень трудозатратные операции – переиндексации.

Разделение индекса по дням сделано для ускорения работы Elasticsearch, а в Kibana можно выбрать несколько по шаблону, здесь log-*, также снимается ограничение в один миллион документов на индекс.

Рассмотрим более подробный плагин вывода у Logstash:

```
output {
if [type] == "Info" {
elasticsearch {
claster => elasticsearch
action => "create"
hosts => ["localhost:9200"]
index => "log-%{+YYYY.MM.dd}"
document_type => ....
document_id => "%{id}"
}
}
}
```

Взаимодействии с ElasticSearch осуществляется через JSON REST API, драйвера для которых есть для большинства современных языков. Но для того, чтобы не писать код, мы воспользуемся утилитой Logstash, которая также он умеет преобразовывать текстовые данные в JSON на основе регулярных выражений. Также есть заготовленные шаблоны, наподобие классов в регулярных выражениях, таких как %{IP:client } и других, которые можно посмотреть по https://github.com/elastic/logstash/tree/v1.1.9/patterns. Для стандартных сервисов со стандартными настройками в интернете есть множество готовых конфигов, например, для NGINX – https://github.com/zooniverse/static/blob/master/logstash— Nginx.conf. Более подобно описано в статье https://habr.com/post/165059/.

ElasticSearch – NoSQL база данных, поэтому не нужно задавать формат (набор полей и его типы). Для поиска ему он всё же нужен, поэтому он сам его определяет, и при каждом

смене формата происходит пере индексации, при котором работа невозможно. Для поддержания унифицированной структуры в логгере Serilog (DOT Net) есть поле EventType в которое можно зашифровать набор полей и их типы, для остальных придётся реализовывать отдельно. Для анализа логов от приложения микро сервисной архитектуры важно задать ID пока выполнения, то есть ID запроса, который будет неизменен и передаваться от микросервиса к микросервису, чтобы можно было проследить весь путь выполнения запроса.

Установим ElasticSearch (https://habr.com/post/280488/) и проверим работу curl -X GET localhost:9200

sudo sysctl -w vm.max_map_count=262144

$ curl 'localhost:9200/_cat/indices?v'

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

green open graylog_0 h2NICPMTQlqQRZhfkvsXRw 4 0 0 0 1kb 1kb

green open .kibana_1 iMJl7vyOTuu1eG8DlWl1OQ 1 0 3 0 11.9kb 11.9kb

yellow open indexname le87KQZwT22lFll8LSRdjw 5 1 1 0 4.5kb 4.5kb

yellow open db i6I2DmplQ7O40AUzyA-a6A 5 1 0 0 1.2kb 1.2kb

Создадим запись в базе данных blog и таблице post curl -X PUT "$ES_URL/blog/post/1?pretty" -d'

Поисковой движок ElasticSearch

В предыдущем разделе мы рассмотрели стек ELK, который составляют ElasticSearch, Logstash и Kibana. В полном наборе, а часто его ещё расширяют Filebeat – более заточенный на работу с расширение Logstash, для работы с текстовыми логами. Несмотря на то, что Logstash выполняет быстро свою задачу без необходимости, не используют, а логи в формате JSON отравляют через API загрузки дампа прямо в Logstash.

Если же у нас приложение, то применяется чистый ElasticSearch, который используется как поисковой движок, а Kibana используется как средство написания и отладки запросов – блок Dev Tools. Хотя и базы реляционные данных имеют долгую историю развития, всё же остаётся принцип, что чем более данные деморализованы, тем они медленнее, ведь их приходится при каждом запросе объединять. Данная проблема решается созданием View, в которой хранится результирующая выборка. Но хоть современные базы данных обзавелись внушительным функционалом, вплоть до полнотекстового поиска, но всё же им не сравниться в эффективности и функциональности поиска с поисковыми движками. Приведу пример с работы: несколько таблиц с метриками, который объединяются в запросе в одну, и производится поиск по выбранным параметрам в админке, таким как диапазон дат, страница в пагинации и содержания в сроке столбца чата. Данный не много, на выходе получаем таблицу в пол миллиона строк, да и поиск по дате и части строки укладывается в миллисекунды. А вот пагинации тормозит, в начальных страницах её запрос выполняется около двух минут, в конечных – более четырёх. При этом объединить запрос на логичен данных и на получения пагинации в лоб не получится. А тот же зарос, при этом он не оптимизирован, в ElasticSearch выполняется за 22 миллисекунды и содержит как данные и, так и число всех данных для пагинации.

Стоит предостеречь читателя от отказа от необдуманной реляционной базы данных, хотя и ElasticSearch содержит в себе NoSQL базу данных, но она предназначена исключительно для поиска и не содержит полноценных средств для нормализации и восстановления.

ElasticSearch не имеет в стандартной поставке консольного клиента – всё взаимодействие осуществляется через http вызовы GET, PUT и DELETE. Приведём пример использования с использованием программы (команды) Curl из программной оболочки BASH ОС linux:

#Создание записей (таблица и база данных создаются автоматически)

curl -XPUT mydb/mytable/1 -d'{

....

}'

```
#Получен значения по id
curl -XGET mydb/mytable/1
curl -XGET mydb/mytable/1

#Простой поиск
curl -XGET mydb -d'{
"search": {
"match": {
"name": "my"
}
}
}'

#Удаление базы
curl -XDELETE mydb
```

Облачные системы, как источник непрерывного масштабирования: Google Cloud и Amazon AWS

Кроме хостинга и аренды сервера, в частности виртуального VPS, можно воспользоваться облачными решениями (SAS, Service As Software) решениями, то есть осуществлять работу нашего WEB приложения (ий) только через панель управления используя готовую инфраструктуру. Этот подход имеет как плюсы, так и минусы, которые зависит от бизнеса заказчика. Если с технической стороны сам сервер удалён, но мы можем к нему подключиться, и как бонус получаем панель администрирования, то для разработчика различия более существенны. Разделим проекты на три группы по месту развёртывания: на хостинге, в своём дата центре или использующие VPS и в облаке. Компании использующие хостинг в силу существенных ограничений, налагаемых на разработку – невозможность установить своё программное обеспечение и нестабильность и размер предоставляемой мощности – в основном специализируются на заказной (потоковой) разработке сайтов и магазинов, которая в силу малых требований к квалификации разработчиков и нетребовательности к знаниям инфраструктуры рынок готов оплачивать их труд по минимум. Ко второй группе относятся компании реализующие состоявшиеся проекты, но разработчики отстранены от работы с инфраструктурой наличием системных администраторов, build инженеров, DevOps и других специалистов по инфраструктуре. Компании, выбирающие облачные решения, в основном оправдывают переплату за готовую инфраструктуру и мощности их расширяемостью (актуально для стартапов, когда рост нагрузки не предсказуем). Для реализации подобных проектов в основном берут высококвалифицированных специалистов широкого круга для реализации нестандартных решений, где инфраструктура уже является просто инструментом, а специалисты по ней просто отсутствуют. На разработчиков возлагаются функции по проектированию проекта в целом, как единого целого, а не программы в отрыве от инфраструктуры. В основном это зарубежные компании, готовые хорошо оплачивать труд ценных сотрудников.

Для развёртывания будем использовать Kubernetes для противодействия vender lock, когда инфраструктура проекта завязана на API конкретного облачного провайдера и не позволят перейти на другие или собственные облака без существенных изменений в самом приложении. Kubernetes поддерживается Amazon AWS, Google Cloud, Microsoft Azure, локальной установкой одного инстанса с помощью MiniKube.

Воспользуемся Google Cloud, на текущий 2018 год он предоставляет бесплатное использование на один год ограниченных ресурсов (300 долларов США), при этом существуют

лимиты, которые можно посмотреть в меню IAM и администрирование –> Квоты. Важно заметить, облачные провайдеры не предоставляют тарифов в современном диапазоне, а предоставляют тарифы на использовании определённых мощностей, то есть сайт мало посещаем – платим мало, сложно обрабатываем много данных – платим много. По этой причине, когда потребности в вычислительных мощностях у компании предсказуемы (не стартап) может быть целесообразно использовать собственные возможности для постоянной нагрузки, что может быть экономически целесообразно, не рискую ограниченностью вычислительными мощностями.

И так заходим на cloud.google.com регистрируется, привязываем дебетовую карту с минимальным балансом и переходим в консоль console.cloud.google.com, в котором можно пройти обучение по интерфейсу для общего ознакомления. В меню нажимаем пункт Оплата: у меня нетронутые демо-деньги 300 долларов США и осталось 356 дней (средства списываются не в режиме реального времени).

Если смотреть на как основу для Back-End для мобильной разработки (MBasS, Mobile backend as a service), то его предоставляют разные провайдеры: Google Firebase, AWS Mobile, Azure Mobile

Google App Engine

Создание кластера через WEB-интерфейс

Предварительно проверим ограничения (квоты) Меню –> Продукты –> IAM и администрирование –> Квоты, а если вы находитесь на тестовом аккаунте, то Static IP addresses будет равен 1, то не сможет создаться балансировщик и придётся удалять кластер. Создадим кластер в Меню – Ресурсы – Kubernetes Engine в тремя репликами микромашины и последней версией Kubernetes. В левом нижнем углу в пункте Marketplace создадим 2 инстанса NGINX. После создания кластера кликнем по вкладке Сервисы и перейдём по IP-адресу.

Marketplace: Сеть, Бесплатные, Приложения Kubernetes: NGINX Создадим кастомный кластер standard-cluster- NGINX, выбрав минимум CPU и RAM, 2 ноды вместо 3 и последнюю вер сию Kubernetes (я выбрал 1.11.3, а мой код будет совместим с – не ниже 1.10). В Меню – Ресурсы – Kubernetes Engine во кладке Кластера нажмём кнопку Подключиться. Управлением кластером в командной строке осуществляется с помощью команды cubectl, о ней можно почитать в документации: https://kubernetes.io/docs/reference/kubectl/overview/ и список по https://gist.github.com/ipedrazas/95391ffd88190bea94ca188d3d2c1cbe.

Создание виртуальной машины:

Можно создать программный проект, но пользоваться им можно будет только на платном аккаунте:

NAME_PROJECT=bitrix_12345;

NAME_CLUSTER=bitrix;

gcloud projects create $NAME_CLUSTER –name $NAME_CLUSTER;

gcloud config set project $NAME_CLUSTER;

gcloud projects list;

Несколько тонкостей: ключ –zone обязателен и ставится в конце, диска не должен быть меньше 10Gb, а тип машин можно взять из https://cloud.google.com/compute/docs/machine-types. Если реплика у нас одна, то по умолчанию создаётся минимальная конфигурация для тестирования:

gcloud container clusters create $NAME_CLUSTER –zone europe-north1-a

Вы можете увидеть в админке, развернув выпадающий список в шапке, и открыв вкладку Все проекты.

gcloud projects delete NAME_PROJECT;

, если больше – стандартная, параметры которой мы подредактируем:

```
$ gcloud container clusters create mycluster \
–-machine-type=n1-standard-1 –disk-size=10GB –image-type ubuntu \
–-scopes compute-rw,gke-default \
–-machine-type=custom-1-1024 \
–-cluster-version=1.11 –enable-autoupgrade \
–-num-nodes=1 –enable-autoscaling –min-nodes=1 –max-nodes=2 \
–-zone europe-north1-a
```

Ключ –enable-autorepair запускаем работу мониторинга доступности ноды и в случае её падения – она будет пересоздана. Ключ требует версию Kubernetes не менее 1.11, а на момент написания книги версия по умолчанию 1.10 и поэтому нужно её задать ключом, например, –cluster-version=1.11.4-gke.12. Но можно зафиксировать только мажорную версия –cluster-version=1.11 и установить автообновление версии –enable-autoupgrade. Также зададим авто уверение количества нод, если ресурсов не хватает: –num-nodes=1 –min-nodes=1 –max-nodes=2 –enable-autoscaling.

Теперь поговорим об виртуальный ядрах и оперативной памяти. По умолчанию поднимается машина n1-standart-1, имеющая одно виртуальное ядро и 3.5Gb оперативной памяти, в трёх экземплярах, что совокупно даёт три виртуальных ядра и 10.5Gb оперативной памяти. Важно, чтобы в кластере было всего не менее двух виртуальных ядер процессора, иначе их, формально по лимитам на системные контейнера Kubernetes, не хватит для полноценной работы (могут не подняться контейнера, например, системные). Я возьму две ноды по одному ядру и общее количество ядер будет два. Такая же ситуация и с оперативной памятью, для поднятия контейнера с NGINX мне вполне хватало 1Gb (1024Mb) оперативной памяти, а вот для поднятия контейнера с LAMP (Apache MySQL PHP) уже нет, у меня не поднялся системный сервис kube-dns-548976df6c-mlljx, который отвечает за DNS в поде. Не смотря не то, что он не является жизненно важным и нам не пригодится, в следующий раз может не подняться уж более важный вместо него. При этом важно заметить, что у меня нормально поднимался кластер с 1Gb и было всё нормально, я общий объём в 2Gb оказался пограничным значением. Я задал 1080Mb (1.25Gb), учтя, что минимальная планка оперативной памяти составляет 256Mb (0.25Gb) и мой объём должен быть кратен ей и быть не меньше, для одно ядра, 1Gb. В результате в кластера 2 ядра и 2.5Gb вместо 3 ядре и 10.5Gb, что является существенной оптимизацией ресурсов и цены на платном аккаунте.

Теперь нам нужно подключиться к серверу. Ключ у нас уже есть на сервере ${HOME}/.kube/config и теперь нам нужно просто авторизоваться:

```
$ gcloud container clusters get-credentials b –zone europe-north1-a –project essch
$ kubectl port-forward Nginxlamp-74c8b5b7f-d2rsg 8080:8080
Forwarding from 127.0.0.1:8080 –> 8080
Forwarding from [::1]:8080 –> 8080
$ google-chrome http://localhost:8080 # это не будет работать в Google Shell
$ kubectl expose Deployment Nginxlamp –type="LoadBalancer" –port=8080
```

Для локального использования kubectl Вам нужно установить gcloud и с помощью него установить kubectl используя команду gcloud components install kubectl, но пока не будем усложнять первые шаги.

В разделе Сервисы админки будет доступен POD не только через сервис балансировщик front-end, но и через внутренний балансировщик Deployment. Хоть и после пересоздания и сохранится, но конфиг более поддерживаем и очевиден.

Также есть возможность дать возможность регулировать количество нод в автоматическом режиме в зависимости от нагрузки, например, количества контейнеров с установленными требованиями к ресурсам, с помощью ключей –enable-autoscaling –min-nodes=1 –max-nodes=2.

Простой кластер в GCP

Для создания кластера можно пойти двумя путями: через графический интерфейс Google Cloud Platform или через его API командой gcloud. Посмотрим, как это можно сделать через UI. Рядом с меню кликнем на выпадающей список и создадим отдельный проект. В разделе Kubernetes Engine выбираем создать кластер. Дадим название, 2CPU, зону europe-north-1 (дата-центр в Финляндии ближе всего к СПб) и последнюю версию Kubernetes. После создания кластера кликаем на подключиться и выбираем Cloud Shell. Для создания через API по кнопке в правом верхнем углу выведем консольную панель и введём в ней:

gcloud container clusters create mycluster –zone europe-north1-a

Через некоторое время, у меня это заняло две с половиной минуты, будут подняты 3 виртуальные машины, на них установлена операционная система и примонтирован диск. Проверим:

esschtolts@cloudshell:~ (essch)$ gcloud container clusters list –filter=name=mycluster

NAME LOCATION MASTER_IP MACHINE_TYPE NODE_VERSION NUM_NODES STATUS

mycluster europe-north1-a 35.228.37.100 n1-standard-1 1.10.9-gke.5 3 RUNNING

esschtolts@cloudshell:~ (essch)$ gcloud compute instances list
NAME MACHINE_TYPE EXTERNAL_IP STATUS
gke-mycluster-default-pool-43710ef9-0168 n1-standard-1 35.228.73.217 RUNNING
gke-mycluster-default-pool-43710ef9-39ck n1-standard-1 35.228.75.47 RUNNING
gke-mycluster-default-pool-43710ef9-g76k n1-standard-1 35.228.117.209 RUNNING

Подключимся к виртуальной машине:

esschtolts@cloudshell:~ (essch)$ gcloud projects list
PROJECT_ID NAME PROJECT_NUMBER
agile-aleph-203917 My First Project 546748042692
essch app 283762935665

esschtolts@cloudshell:~ (essch)$ gcloud container clusters get-credentials mycluster \
–-zone europe-north1-a \
–-project essch
Fetching cluster endpoint and auth data.
kubeconfig entry generated for mycluster.

У нас пока нет кластера:

esschtolts@cloudshell:~ (essch)$ kubectl get pods
No resources found.

Создадим кластер:

esschtolts@cloudshell:~ (essch)$ kubectl run Nginx –image=Nginx –replicas=3
deployment.apps "Nginx" created

Проверим его состав:

esschtolts@cloudshell:~ (essch)$ kubectl get deployments –selector=run=Nginx
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
Nginx 3 3 3 3 14s

esschtolts@cloudshell:~ (essch)$ kubectl get pods –selector=run=Nginx
NAME READY STATUS RESTARTS AGE
Nginx-65899c769f-9whdx 1/1 Running 0 43s
Nginx-65899c769f-szwtd 1/1 Running 0 43s
Nginx-65899c769f-zs6g5 1/1 Running 0 43s

Удостоверимся, что все три реплики кластера распределились равномерно на все три ноды:

esschtolts@cloudshell:~ (essch)$ kubectl describe pod Nginx-65899c769f-9whdx | grep Node:
Node: gke-mycluster-default-pool-43710ef9-g76k/10.166.0.5
esschtolts@cloudshell:~ (essch)$ kubectl describe pod Nginx-65899c769f-szwtd | grep Node:
Node: gke-mycluster-default-pool-43710ef9-39ck/10.166.0.4
esschtolts@cloudshell:~ (essch)$ kubectl describe pod Nginx-65899c769f-zs6g5 | grep Node:
Node: gke-mycluster-default-pool-43710ef9-g76k/10.166.0.5

Теперь поставим балансировщик нагрузки:

esschtolts@cloudshell:~ (essch)$ kubectl expose Deployment Nginx –type="LoadBalancer" –port=80
service "Nginx" exposed

Проверим, что он создался:

esschtolts@cloudshell:~ (essch)$ kubectl expose Deployment Nginx –type="LoadBalancer" –port=80
service "Nginx" exposed

esschtolts@cloudshell:~ (essch)$ kubectl get svc –selector=run=Nginx
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
Nginx LoadBalancer 10.27.245.187 pending> 80:31621/TCP 11s

esschtolts@cloudshell:~ (essch)$ sleep 60;

esschtolts@cloudshell:~ (essch)$ kubectl get svc –selector=run=Nginx
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
Nginx LoadBalancer 10.27.245.187 35.228.212.163 80:31621/TCP 1m

Проверим его работу:

esschtolts@cloudshell:~ (essch)$ curl 35.228.212.163:80 2>\dev\null | grep h1
< h1>Welcome to Nginx!< /h1>

Чтобы каждый раз не копировать полные названия – сохраним их в переменных (подробнее о формате JSONpath в документации Go: https://golang.org/pkg/text/template/#pkg-overview):

esschtolts@cloudshell:~ (essch)$ pod1=$(kubectl get pods -o jsonpath={.items[0].metadata.name});
esschtolts@cloudshell:~ (essch)$ pod2=$(kubectl get pods -o jsonpath={.items[1].metadata.name});
esschtolts@cloudshell:~ (essch)$ pod3=$(kubectl get pods -o jsonpath={.items[2].metadata.name});
esschtolts@cloudshell:~ (essch)$ echo $pod1 $pod2 $pod3
Nginx-65899c769f-9whdx Nginx-65899c769f-szwtd Nginx-65899c769f-zs6g5

Изменим страницы в каждом POD, скопировав уникальные страницы в каждую реплику, и проверим балансировку, проверяя распределение запросов по POD:

esschtolts@cloudshell:~ (essch)$ echo 1 > test.html;
esschtolts@cloudshell:~ (essch)$ kubectl cp test.html ${pod1}:/usr/share/Nginx/html/index.html
esschtolts@cloudshell:~ (essch)$ echo 2 > test.html;
esschtolts@cloudshell:~ (essch)$ kubectl cp test.html ${pod2}:/usr/share/Nginx/html/index.html
esschtolts@cloudshell:~ (essch)$ echo 3 > test.html;

esschtolts@cloudshell:~ (essch)$ kubectl cp test.html ${pod3}:/usr/share/Nginx/html/index.html

esschtolts@cloudshell:~ (essch)$ curl 35.228.212.163:80 && curl 35.228.212.163:80 && curl 35.228.212.163:80
3
2
1

esschtolts@cloudshell:~ (essch)$ curl 35.228.212.163:80 && curl 35.228.212.163:80 && curl 35.228.212.163:80
3
1
1

Проверим отказоустойчивость кластера удалением одного POD:

esschtolts@cloudshell:~ (essch)$ kubectl delete pod ${pod1} && kubectl get pods && sleep 10 && kubectl get pods
pod "Nginx-65899c769f-9whdx" deleted
NAME READY STATUS RESTARTS AGE
Nginx-65899c769f-42rd5 0/1 ContainerCreating 0 1s
Nginx-65899c769f-9whdx 0/1 Terminating 0 54m
Nginx-65899c769f-szwtd 1/1 Running 0 54m
Nginx-65899c769f-zs6g5 1/1 Running 0 54m
NAME READY STATUS RESTARTS AGE
Nginx-65899c769f-42rd5 1/1 Running 0 12s
Nginx-65899c769f-szwtd 1/1 Running 0 55m
Nginx-65899c769f-zs6g5 1/1 Running 0 55m

Как мы видим, сразу после того, как POD стал недоступен (начался процесс его удаления) начала создаваться его замена. Вскоре, кластер полностью восстановит свою структуру. После того как мы закончили наши эксперименты, удалим виртуальные машины с кластером:

esschtolts@cloudshell:~ (essch)$ gcloud container clusters delete mycluster –zone europe-north1-a;
The following clusters will be deleted.
– [mycluster] in [europe-north1-a]
Do you want to continue (Y/n)? Y
Deleting cluster mycluster…done.
Deleted [https://container.googleapis.com/v1/projects/essch/zones/europe-north1-a/clusters/mycluster].

esschtolts@cloudshell:~ (essch)$ gcloud container clusters list –filter=name=mycluster

Итого. Мы создали кластер и создали балансировщик нагрузки всего двумя командами run и expose, теперь мы может заходить по IP-адресу балансировщика и наблюдать в браузере приветствующую страницу NGINX. При этом кластер само восстанавливается, для этого мы эмулировали отказ пода его удалением – он был создан снова.

Воспроизводимость создания кластера

Давайте разберём ситуацию из предыдущей главы, в которой мы создали кластер, удалили реплику, а она восстановилась. Дело в том, что мы не управляем командами на прямую, а с помощью команд создаём описания необходимой конфигурации кластера и помещаем его в распределённое хранилище, после чего состояние нод поддерживаются в соответствии с этим описанием в распределённом хранилище. Мы также можем получить и отредактировать эти

описании или же написать самим и потом загрузить их в распределённое хранилище. Это позволит нам сохранять состояние на диске в виде YAML файлов и восстанавливать его обратно, так часто поступают при переносе с рабочего сервера на тестовый. К тому же мы получаем возможность более гибко настраивать состояние, но, так как мы не ограниченны командами.

```
esschtolts@cloudshell:~ (essch)$ kubectl get deployment/Nginx –output=yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
annotations:
deployment.kubernetes.io/revision: "1"
creationTimestamp: 2018-12-16T10:23:26Z
generation: 1
labels:
run: Nginx
name: Nginx
namespace: default
resourceVersion: "1612985"
selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/Nginx
uid: 9fb3ad6a-011c-11e9-bfaa-42010aa60088
spec:
progressDeadlineSeconds: 600
replicas: 1
revisionHistoryLimit: 10
selector:
matchLabels:
run: Nginx
strategy:
rollingUpdate:
maxSurge: 1
maxUnavailable: 1
type: RollingUpdate
template:
metadata:
creationTimestamp: null
labels:
run: Nginx
spec:
containers:
– image: Nginx
imagePullPolicy: Always
name: Nginx
resources: {}
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
dnsPolicy: ClusterFirst
restartPolicy: Always
schedulerName: default-scheduler
securityContext: {}
terminationGracePeriodSeconds: 30
```

status:
availableReplicas: 1
conditions:
– lastTransitionTime: 2018-12-16T10:23:26Z
lastUpdateTime: 2018-12-16T10:23:26Z
message: Deployment has minimum availability.
reason: MinimumReplicasAvailable
status: "True"
type: Available
– lastTransitionTime: 2018-12-16T10:23:26Z
lastUpdateTime: 2018-12-16T10:23:28Z
message: ReplicaSet "Nginx-64f497f8fd" has successfully progressed.
reason: NewReplicaSetAvailable
status: "True"
type: Progressing
observedGeneration: 1
readyReplicas: 1
replicas: 1
updatedReplicas: 1

Для нас это будет излишним, поэтому удалю ненужное, ведь когда создавали, мы указали лишь имя и образ, остальное было заполнено значениями по умолчанию:

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
labels:
run: Nginx
name: Nginx
spec:
selector:
matchLabels:
run: Nginx
template:
metadata:
labels:
run: Nginx
spec:
containers:
– image: Nginx
name: Nginx

Также можно создать шаблон:

gcloud services enable compute.googleapis.com –project=${PROJECT}
gcloud beta compute instance-templates create-with-container ${TEMPLATE} \
–-machine-type=custom-1-4096 \
–-image-family=cos-stable \
–-image-project=cos-cloud \
–-container-image=gcr.io/kuar-demo/kuard-amd64:1 \
–-container-restart-policy=always \
–-preemptible \
–-region=${REGION} \

```
–-project=${PROJECT}
gcloud compute instance-groups managed create ${TEMPLATE} \
–-base-instance-name=${TEMPLATE} \
–-template=${TEMPLATE} \
–-size=${CLONES} \
–-region=${REGION} \
–-project=${PROJECT}
```

Высокая доступность сервиса

Чтобы обеспечить высокую доступность нужно в случае падения приложения перенаправлять трафик на запасной. Также, часто важно, чтобы нагрузка была распределена равномерно, так как приложение в единичном экземпляре не способно обрабатывать весь трафик. Для этого создаётся кластер, для примера возьмём более сложный образ, чтобы разобрать большее количество нюансов:

```
esschtolts@cloudshell:~/bitrix (essch)$ cat deploymnet.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: Nginxlamp
spec:
selector:
matchLabels:
app: lamp
replicas: 1
template:
metadata:
labels:
app: lamp
spec:
containers:
– name: lamp
image: mattrayner/lamp:latest-1604-php5
ports:
– containerPort: 80

esschtolts@cloudshell:~/bitrix (essch)$ cat loadbalancer.yaml
apiVersion: v1
kind: Service
metadata:
name: frontend
spec:
type: LoadBalancer
ports:
– name: front
port: 80
targetPort: 80
selector:
app: lamp

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods
```

NAME READY STATUS RESTARTS AGE
Nginxlamp-7fb6fdd47b-jttl8 2/2 Running 0 3m

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
frontend LoadBalancer 10.55.242.137 35.228.73.217 80:32701/TCP,8080:32568/TCP 4m
kubernetes ClusterIP 10.55.240.1 none> 443/TCP 48m

Теперь мы можем создать идентичные копии наших кластеров, например, для Production и Develop, но балансировка не будет работать должным образом. Балансировщик будет находить POD по метке, а этой метке соответствуют и POD в production, и в Developer кластере. Также не станет препятствием размещение кластеров в разных проектах. Хотя, для многих задач, это большой плюс, но не в случае кластера для разработчиков и продакшне. Для разграничения области видимости используются namespace. Мы незаметно их используем, когда мы выводим список POD без указания области видимости нам выводится область видимости по умолчанию default, но не выводятся POD из системной области видимости:

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get namespace
NAME STATUS AGE
default Active 5h
kube-public Active 5h
kube-system Active

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods –namespace=kube-system
NAME READY STATUS RESTARTS AGE
event-exporter-v0.2.3-85644fcdf-tdt7h 2/2 Running 0 5h
fluentd-gcp-scaler-697b966945-bkqrm 1/1 Running 0 5h
fluentd-gcp-v3.1.0-xgtw9 2/2 Running 0 5h
heapster-v1.6.0-beta.1-5649d6ddc6-p549d 3/3 Running 0 5h
kube-dns-548976df6c-8lvp6 4/4 Running 0 5h
kube-dns-548976df6c-mcctq 4/4 Running 0 5h
kube-dns-autoscaler-67c97c87fb-zzl9w 1/1 Running 0 5h
kube-proxy-gke-bitrix-default-pool-38fa77e9-0wdx 1/1 Running 0 5h
kube-proxy-gke-bitrix-default-pool-38fa77e9-wvrf 1/1 Running 0 5h
l7-default-backend-5bc54cfb57-6qk4l 1/1 Running 0 5h
metrics-server-v0.2.1-fd596d746-g452c 2/2 Running 0 5h

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods –namespace=default
NAMEREADY STATUS RESTARTS AGE
Nginxlamp-b5dcb7546-g8j5r 1/1 Running 0 4h
Создадим область видимости:
esschtolts@cloudshell:~/bitrix (essch)$ cat namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
name: development
labels:
name: development

esschtolts@cloudshell:~ (essch)$ kubectl create -f namespace.yaml
namespace "development" created

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get namespace –show-labels
NAME STATUS AGE LABELS
default Active 5h none>
development Active 16m name=development
kube-public Active 5h none>
kube-system Active 5h none>

Суть работы с областью видимости в том, что для конкретных кластеров мы задаём область видимость и можем выполнять команды указывая её, при этом они будут распространяться только на них. При этом, кроме ключей в командах, таких как kubectl get pods области видимости не фигурирую, поэтому конфигурационные файлы контроллеров (Deployment, DaemonSet и других) и сервисов (LoadBalancer, NodePort и других) не фигурируют, позволяя беспроблемно переносить их между областью видимости, что особенно актуально для pipeline разработки: сервер разработчика, тестовый сервер и продакшн сервер. Области видимости прописываются в файле контекстов кластеров $HOME/ .kube/config с помощью команды kubectl config view. Так, у меня в записи контекста нашего кластера не фигурирует запись об области видимости (по умолчанию default):

– context:
cluster: gke_essch_europe-north1-a_bitrix
user: gke_essch_europe-north1-a_bitrix
name: gke_essch_europe-north1-a_bitrix

Посмотреть можно примерно подобным образом:

esschtolts@cloudshell:~/bitrix (essch)$ kubectl config view -o jsonpath='{.contexts[4]}'
{gke_essch_europe-north1-a_bitrix {gke_essch_europe-north1-a_bitrix gke_essch_europe-north1-a_bitrix []}}

Создадим новый контекст для данного пользователя и кластера:

esschtolts@cloudshell:~ (essch)$ kubectl config set-context dev \
> –namespace=development \
> –cluster=gke_essch_europe-north1-a_bitrix \
> –user=gke_essch_europe-north1-a_bitrix
Context "dev" modified.

esschtolts@cloudshell:~/bitrix (essch)$ kubectl config set-context dev \
> –namespace=development \
> –cluster=gke_essch_europe-north1-a_bitrix \
> –user=gke_essch_europe-north1-a_bitrix
Context "dev" modified.

В результате был добавлен:

– context:
cluster: gke_essch_europe-north1-a_bitrix
namespace: development
user: gke_essch_europe-north1-a_bitrix
name: dev

Теперь осталось переключиться на него:

esschtolts@cloudshell:~ (essch)$ kubectl config use-context dev
Switched to context "dev".

esschtolts@cloudshell:~ (essch)$ kubectl config current-context
dev

esschtolts@cloudshell:~ (essch)$ kubectl get pods
No resources found.

esschtolts@cloudshell:~ (essch)$ kubectl get pods –namespace=default
NAMEREADY STATUS RESTARTS AGE
Nginxlamp-b5dcb7546-krkm2 1/1 Running 0 10h
Можно было добавить в существующий контекст пространство имён:
esschtolts@cloudshell:~/bitrix (essch)$ kubectl config set-context $(kubectl config current-context) –namespace=development
Context "gke_essch_europe-north1-a_bitrix" modified.
Теперь создадим кластер в новой области видимости dev(она теперь по умолчанию и её можно не указывать –namespace=dev) и удалим из области видимости по умолчанию default (она теперь не по умолчанию для нашего кластера и её нужно указывать –namespace =default):
esschtolts@cloudshell:~ (essch)$ cd bitrix/

esschtolts@cloudshell:~/bitrix (essch)$ kubectl create -f deploymnet.yaml -f loadbalancer.yaml
deployment.apps "Nginxlamp" created
service "frontend" created

esschtolts@cloudshell:~/bitrix (essch)$ kubectl delete -f deploymnet.yaml -f loadbalancer.yaml –namespace=default
deployment.apps "Nginxlamp" deleted
service "frontend" deleted

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods
NAMEREADY STATUS RESTARTS AGE
Nginxlamp-b5dcb7546-8sl2f 1/1 Running 0 1m
Теперь посмотрим внешний IP-адресс и откроем страницу:
esschtolts@cloudshell:~/bitrix (essch)$ curl $(kubectl get -f loadbalancer.yaml -o json
| jq -r .status.loadBalancer.ingress[0].ip) 2>/dev/null | grep '< h2 >'
< h2>Welcome to github.com/mattrayner/docker-lamp" target="_blank">Docker-Lamp a.k.a mattrayner/lamp< /h2>
Кастомизация
Теперь нам нужно изменить стандартное решение под наши нужды, а именно добавить конфиги и наше приложение. Для простоты мы пометим (изменим стандартный) в корень нашего приложения файл .htaccess, сведя к простому помещения нашего приложения в папку / app. Первое, что напрашивается сделать, это создать POD и потом скопировать с хоста в контейнер наше приложение (я взял Bitrix):
Хотя это решение и работает, оно имеет ряд существенных недостатков. Первое, что то, что нам нужно дожидаться из вне, постоянным опросом POD, когда он поднимет контейнер и мы в него скопируем приложение и не должен этого делать, если контейнер не поднялся, а также обрабатывать ситуацию когда он сломает наш POD, внешние сервисы, могут опираться на статус POD, хоты сам POD будет ещё не готов, пока не будет выполнен скрипт. Вторым недостатком является то, что у нас появляется какой то внешний скрипт, который нужно логически не отделим от POD, но при этом его нужно вручную запускать из вне, где хранить и где-то должна быть инструкция по его использованию. И напоследок, этих POD у нас может быть множество. На первый взгляд, логичным решением поместить код в Dockerfile:

```
esschtolts@cloudshell:~/bitrix (essch)$ cat Dockerfile
FROM mattrayner/lamp:latest-1604-php5
MAINTAINER ESSch ESSchtolts@yandex.ru>
RUN cd /app/ && ( \
wget https://www.1c-bitrix.ru/download/small_business_encode.tar.gz \
&& tar -xf small_business_encode.tar.gz \
&& sed -i '5i php_value short_open_tag 1' .htaccess \
&& chmod -R 0777 . \
&& sed -i 's/#php_value display_errors 1/php_value display_errors 1/' .htaccess \
&& sed -i '5i php_value opcache.revalidate_freq 0' .htaccess \
&& sed -i 's/#php_flag default_charset UTF-8/php_flag default_charset UTF-8/' .htaccess \
) && cd ..;
EXPOSE 80 3306
CMD ["/run.sh"]

esschtolts@cloudshell:~/bitrix (essch)$ docker build -t essch/app:0.12 . | grep Successfully
Successfully built f76e656dac53
Successfully tagged essch/app:0.12

esschtolts@cloudshell:~/bitrix (essch)$ docker image push essch/app | grep digest
0.12: digest: sha256:75c92396afacefdd5a3fb2024634a4c06e584e2a1674a866fa72f8430b19ff69 size: 11309

esschtolts@cloudshell:~/bitrix (essch)$ cat deploymnet.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: Nginxlamp
namespace: development
spec:
selector:
matchLabels:
app: lamp
replicas: 1
template:
metadata:
labels:
app: lamp
spec:
containers:
– name: lamp
image: essch/app:0.12
ports:
– containerPort: 80

esschtolts@cloudshell:~/bitrix (essch)$ IMAGE=essch/app:0.12 kubectl create -f deploymnet.yaml
deployment.apps "Nginxlamp" created
```

esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods -l app=lamp
NAME READY STATUS RESTARTS AGE
Nginxlamp-55f8cd8dbc-mk9nk 1/1 Running 0 5m

esschtolts@cloudshell:~/bitrix (essch)$ kubectl exec Nginxlamp-55f8cd8dbc-mk9nk – ls /app/
index.php

Это происходит потому, что разработчик образов, что правильно и написано в его документации, ожидал, что образ будет примонтирован к хосту и в скрипте запускаемого в сам конце удаляется папка app. Также в таком подходе мы столкнёмся с проблемой постоянных обновлений образов, конфигом (мы не сможем задать номер образа переменной, так как он будет исполняться на нодах кластера) и обновлений контейнеров, также мы не сможем обновлять папку, так как при пересоздании контейнера изменения будут возвращены в изначальное состояние.

Правильным решением будет примонтировать папку и включение в состав жизненно цикла POD запуск контейнера, который стартует перед основным контейнером и производит подготовительные операции окружения, часто это скачивание приложения с репозитория, сборка, прогон тестов, создания пользователей и выставление прав. Под каждую операцию правильно запускать отдельный init контейнер, в котором эта операция является базовым процессом, которые выполняются последовательно – цепочкой, которая будет разорвана, если одна из операций будет выполнена с ошибкой (вернёт не нулевой код завершения процесса). Для такого контейнера предусмотрено отдельное описание в POD – InitContainer, перечисляя их последовательно они будет выстраивать цепочку запусков init контейнеров в том же порядке. В нашем случае мы создали неименованный volume и с помощью InitContainer доставили в него установочные файлы. После успешного завершения InitContainer, которых может быть несколько, стартует основной. Основной контейнер уже монтируется в наш том, в котором уже есть установочные файлы, нам остаётся лишь перейти в браузер и выполнить установку:

esschtolts@cloudshell:~/bitrix (essch)$ cat deploymnet.yaml
kind: Deployment
metadata:
name: Nginxlamp
namespace: development
spec:
selector:
matchLabels:
app: lamp
replicas: 1
template:
metadata:
labels:
app: lamp
spec:
initContainers:
– name: init
image: ubuntu
command:
– /bin/bash
– -c
– |
cd /app

apt-get update && apt-get install -y wget
wget https://www.1c-bitrix.ru/download/small_business_encode.tar.gz
tar -xf small_business_encode.tar.gz
sed -i '5i php_value short_open_tag 1' .htaccess
chmod -R 0777 .
sed -i 's/#php_value display_errors 1/php_value display_errors 1/' .htaccess
sed -i '5i php_value opcache.revalidate_freq 0' .htaccess
sed -i 's/#php_flag default_charset UTF-8/php_flag default_charset UTF-8/' .htaccess
volumeMounts:
– name: app
mountPath: /app
containers:
– name: lamp
image: essch/app:0.12
ports:
– containerPort: 80
volumeMounts:
– name: app
mountPath: /app
volumes:
– name: app
emptyDir: {}

Посмотреть события во время создания POD можно командой watch kubectl get events, а логи kubectl logs {ID_CONTAINER} -c init или более универсально:

kubectl logs $(kubectl get PODs -l app=lamp -o JSON | jq ".items[0].metadata.name" | sed 's/"//g') -c init

Целесообразно выбирать для единичных задач маленькие образа, например, alpine:3.5:

esschtolts@cloudshell:~ (essch)$ docker pull alpine 1>\dev\null
esschtolts@cloudshell:~ (essch)$ docker pull ubuntu 1>\dev\null
esschtolts@cloudshell:~ (essch)$ docker images
REPOSITORY TAGIMAGE ID CREATED SIZE
ubuntu latest 93fd78260bd1 4 weeks ago 86.2MB
alpine latest 196d12cf6ab1 3 months ago 4.41MB

Немного изменив код существенно сэкономили на размере образа:

image: alpine:3.5
command:
– /bin/bash
– -c
– |
cd /app
apk –update add wget && rm -rf /var/cache/apk/*
tar -xf small_business_encode.tar.gz
rm -f small_business_encode.tar.gz
sed -i '5i php_value short_open_tag 1' .htaccess
sed -i 's/#php_value display_errors 1/php_value display_errors 1/' .htaccess
sed -i '5i php_value opcache.revalidate_freq 0' .htaccess
sed -i 's/#php_flag default_charset UTF-8/php_flag default_charset UTF-8/' .htaccess
chmod -R 0777 .
volumeMounts:

Существуют также минималистичные образа с предустановленными пакетами, такие как APIne с git: axeclbr/git и golang:1-alpine.

Способы обеспечения устойчивости к сбоям

Любой процесс может упасть. В случае с контейнером, если падает основной процесс, то падает и контейнер, содержащий его. Это нормально, если падение случилось в процессе корректного завершения работы. К примеру, наше приложение в контейнере делает бэкап базы данных, таком случае после выполнения контейнера мы получаем сделанную работу. Для демонстрации, возьмём команду sleep:

```
vagrant@ubuntu:~$ sudo docker pull ubuntu > /dev/null
vagrant@ubuntu:~$ sudo docker run -d ubuntu sleep 60
0bd80651c6f97167b27f4e8df675780a14bd9e0a5c3f8e5e8340a98fc351bc64

vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS NAMES
0bd80651c6f9 ubuntu "sleep 60" 15 seconds ago Up 12 seconds distracted_kalam

vagrant@ubuntu:~$ sleep 60

vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS NAMES

vagrant@ubuntu:~$ sudo docker ps -a | grep ubuntu
0bd80651c6f9 ubuntu "sleep 60" 4 minutes ago Exited (0) 3 minutes ago distracted_kalam
```

В случае с бэкапам – это норма, а в случае с приложениями, которые не должны завершаться – нет. Типичный прием – веб-сервер. Самое простое в таком случае заново рестартовать его:

```
vagrant@ubuntu:~$ sudo docker run -d –restart=always ubuntu sleep 10
c3bc2d2e37a68636080898417f5b7468adc73a022588ae073bdb3a5bba270165

vagrant@ubuntu:~$ sleep 30

vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
c3bc2d2e37a6 ubuntu sleep 10" 46 seconds ago Up 1 second
```

Мы видим, что, когда контейнер падает – он рестартует. Как результат – у нас всегда приложение в двух состояниях – поднимается или поднято. Если веб-сервер падает от какой-то редкой ошибки – это норма, но, скорее всего, ошибка в обработке запросов, и он будет падать на каждом таком запросе, а в мониторинге мы увидим поднятый контейнер. Такой веб-сервер лучше мёртвый, чем наполовину живой. Но, при этом нормальный веб-сервер может не стартануть из-за редких ошибок, например, из-за отсутствия подключения к базе данных из-за нестабильности сети. В таком случает, приложение должно уметь обрабатывать ошибки и завершаться. А в случае падения из-за ошибок кода – не перезапускать, чтобы увидеть неработоспособность и отправить на починку разработчикам. В случает же плавающей ошибки можно попробовать несколько раз:

```
vagrant@ubuntu:~$ sudo docker run -d –restart=on-failure:3 ubuntu sleep 10
056c4fc6986a13936e5270585e0dc1782a9246f01d6243dd247cb03b7789de1c
vagrant@ubuntu:~$ sleep 10
vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

c3bc2d2e37a6 ubuntu "sleep 10" 9 minutes ago Up 2 seconds keen_sinoussi
vagrant@ubuntu:~$ sleep 10
vagrant@ubuntu:~$ sleep 10
vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
c3bc2d2e37a6 ubuntu "sleep 10" 10 minutes ago Up 9 seconds keen_sinoussi
vagrant@ubuntu:~$ sleep 10
vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
c3bc2d2e37a6 ubuntu "sleep 10" 10 minutes ago Up 2 seconds keen_sinoussi

Другим аспектом является то, когда считать контейнер умершим. По умолчанию – это падание процесса. Но, далеко, не всегда приложение само падает в случае ошибки, чтобы дать контейнеру его перезапустить. Например, сервер может быть разработан неправильно и пытаться во время своего запуска скачать необходимые библиотеки, при этом этой возможности у него нет, например, из-за блокировки запросов файрволлом. В таком сценарии сервер может долго ожидать, если не прописан адекватный таймаут. В таком случае, нам нужно проверить работоспособность. Для веб-сервера это ответ на определённый url, например:

docker run –rm -d \
–-name=elasticsearch \
–-health-cmd="curl –silent –fail localhost:9200/_cluster/health || exit 1" \
–-health-interval=5s \
–-health-retries=12 \
–-health-timeout=20s \
{image}

Для демонстрации мы возьмём команду создания файла. Если приложение не достигло в отведённый лимит времени (поставим пок 0) рабочего состояния (к примеру, создания файла), то помечается как на рабочее, но до этого делается заданное кол-во проверок:

vagrant@ubuntu:~$ sudo docker run \
–d –name healt \
–-health-timeout=0s \
–-health-interval=5s \
–-health-retries=3 \
–-health-cmd="ls /halth" \
ubuntu bash -c 'sleep 1000'
c0041a8d973e74fe8c96a81b6f48f96756002485c74e51a1bd4b3bc9be0d9ec5

vagrant@ubuntu:~$ sudo docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
c0041a8d973e ubuntu "bash -c 'sleep 1000'" 4 seconds ago Up 3 seconds (health: starting) healt

vagrant@ubuntu:~$ sleep 20
vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
c0041a8d973e ubuntu "bash -c 'sleep 1000'" 38 seconds ago Up 37 seconds (unhealthy) healt

vagrant@ubuntu:~$ sudo docker rm -f healt
healt

Если же хотя бы одна из проверок сработала – то контейнер помечается как работоспособный (healthy) сразу:

```
vagrant@ubuntu:~$ sudo docker run \
–d –name healt \
–-health-timeout=0s \
–-health-interval=5s \
–-health-retries=3 \
–-health-cmd="ls /halth" \
ubuntu bash -c 'touch /health && sleep 1000'

vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
160820d11933 ubuntu "bash -c 'touch /hal…" 4 seconds ago Up 2 seconds (health: starting) healt
vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
160820d11933 ubuntu "bash -c 'touch /hal…" 6 seconds ago Up 5 seconds (healthy) healt

vagrant@ubuntu:~$ sudo docker rm -f healt
healt
```

При этом проверки повторяются всё время с заданным интервалом:

```
vagrant@ubuntu:~$ sudo docker run \
–d –name healt \
–-health-timeout=0s \
–-health-interval=5s \
–-health-retries=3 \
–-health-cmd="ls /halth" \
ubuntu bash -c 'touch /health && sleep 60 && rm -f /health && sleep 60'

vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8ec3a4abf74b ubuntu "bash -c 'touch /hal…" 7 seconds ago Up 5 seconds (health: starting) healt
vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8ec3a4abf74b ubuntu "bash -c 'touch /hal…" 24 seconds ago Up 22 seconds (healthy) healt
vagrant@ubuntu:~$ sleep 60
vagrant@ubuntu:~$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8ec3a4abf74b ubuntu "bash -c 'touch /hal…" About a minute ago Up About a minute (unhealthy) healt
```

Kubernetes предоставляет (kubernetes.io/docs/tasks/configure-POD-container/configure-liveness-readiness-probes/) три инструмента, которые проверяют состояние контейнера из вне. Они имеют больше значение, так они служат не только для информирования, но и для управления жизненным циклом приложения, накатом и откатом обновления. Их неправильная настройка может, и часто такое бывает, приводят к неработоспособности приложения. Так, к если liveness проба буте срабатывать до начала работы приложения – Kubernetes будет убивать контейнер, не дав ему подняться. Рассмотрим её более подробно. Проба liveness служит для определения работоспособности приложения и в случае, если приложение сломалось и не отвечает на пробу liveness – Kubernetes перезагружает контейнер. В качестве примера возьмём shell-пробу, из-за простоты демонстрации работы, но на практике её следует использовать

только в крайних случаях, например, если контейнер запускается не как долгоживущий сервер, а как JOB, выполняющий свою работу и завершающий своё существование, достигнув результата. Для проверок серверов лучше использовать HTTP-пробы, которые уже имеют встроенный выделенный проксировщик и не требуют наличия в контейнере curl и не зависящие от внешних настроек kube-proxy. При использовании баз данных нужно использовать TCP-пробу, так как HTTP—протокол они обычно не поддерживают. Создадим долгоживущий контейнер в www.katacoda.com/courses/kubernetes/playground:

```
controlplane $ cat << EOF > liveness.yaml
apiVersion: v1
kind: Pod
metadata:
name: liveness
spec:
containers:
– name: healtcheck
image: alpine:3.5
args:
– /bin/sh
– -c
– touch /tmp/healthy; sleep 10; rm -rf /tmp/healthy; sleep 60
livenessProbe:
exec:
command:
– cat
– /tmp/healthy
initialDelaySeconds: 15
periodSeconds: 5
EOF

controlplane $ kubectl create -f liveness.yaml
pod/liveness created

controlplane $ kubectl get pods
NAME READY STATUS RESTARTS AGE
liveness 1/1 Running 2 2m11s

controlplane $ kubectl describe pod/liveness | tail -n 10
Type Reason Age From Message
— – – – –
Normal Scheduled 2m37s default-scheduler Successfully assigned default/liveness to node01
Normal Pulling 2m33s kubelet, node01 Pulling image "alpine:3.5"
Normal Pulled 2m30s kubelet, node01 Successfully pulled image "alpine:3.5"
Normal Created 33s (x3 over 2m30s) kubelet, node01 Created container healtcheck
Normal Started 33s (x3 over 2m30s) kubelet, node01 Started container healtcheck
Normal Pulled 33s (x2 over 93s) kubelet, node01 Container image "alpine:3.5" already present on machine
Warning Unhealthy 3s (x9 over 2m13s) kubelet, node01 Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory
```

Normal Killing 3s (x3 over 2m3s) kubelet, node01 Container healtcheck failed liveness probe, will be restarted

Мы видим, что контейнер постоянно перезапускается.

```
controlplane $ cat << EOF > liveness.yaml
apiVersion: v1
kind: Pod
metadata:
name: liveness
spec:
containers:
– name: healtcheck
image: alpine:3.5
args:
– /bin/sh
– -c
– touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 60
livenessProbe:
exec:
command:
– cat
– /tmp/healthy
initialDelaySeconds: 15
periodSeconds: 5
EOF

controlplane $ kubectl create -f liveness.yaml
pod/liveness created

controlplane $ kubectl get pods
NAME READY STATUS RESTARTS AGE
liveness 1/1 Running 2 2m53s

controlplane $ kubectl describe pod/liveness | tail -n 15
SecretName: default-token-9v5mb
Optional: false
QoS Class: BestEffort
Node-Selectors: < none>
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
node.kubernetes.io/unreachable:NoExecute for 300s
Events:
Type Reason Age From Message
— – – – –
```

Normal Scheduled 3m44s default-scheduler Successfully assigned default/liveness to node01

Normal Pulled 68s (x3 over 3m35s) kubelet, node01 Container image "alpine:3.5" already present on machine

Normal Created 68s (x3 over 3m35s) kubelet, node01 Created container healtcheck

Normal Started 68s (x3 over 3m34s) kubelet, node01 Started container healtcheck

Warning Unhealthy 23s (x9 over 3m3s) kubelet, node01 Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory

Normal Killing 23s (x3 over 2m53s) kubelet, node01 Container healtcheck failed liveness probe, will be restarted

Также мы видим и на событиях кластера, что когда cat /tmp/health терпит неудачу – контейнере пересоздаётся:

controlplane $ kubectl get events

controlplane $ kubectl get events | grep pod/liveness

13m Normal Scheduled pod/liveness Successfully assigned default/liveness to node01

13m Normal Pulling pod/liveness Pulling image "alpine:3.5"

13m Normal Pulled pod/liveness Successfully pulled image "alpine:3.5"

10m Normal Created pod/liveness Created container healtcheck

10m Normal Started pod/liveness Started container healtcheck

10m Warning Unhealthy pod/liveness Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory

10m Normal Killing pod/liveness Container healtcheck failed liveness probe, will be restarted

10m Normal Pulled pod/liveness Container image "alpine:3.5" already present on machine

8m32s Normal Scheduled pod/liveness Successfully assigned default/liveness to node01

4m41s Normal Pulled pod/liveness Container image "alpine:3.5" already present on machine

4m41s Normal Created pod/liveness Created container healtcheck

4m41s Normal Started pod/liveness Started container healtcheck

2m51s Warning Unhealthy pod/liveness Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory

5m11s Normal Killing pod/liveness Container healtcheck failed liveness probe, will be restarted

Рассмотрим RadyNess пробу. Доступность этой пробы свидетельствует, что приложение готово к принятию запросов и можно на него сервис может переключать трафик:

controlplane $ cat << EOF > readiness.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: readiness

spec:

replicas: 2

selector:

matchLabels:

app: readiness

template:

metadata:

labels:

app: readiness

spec:

containers:

– name: readiness

image: python

args:

– /bin/sh

– -c

– sleep 15 && (hostname > health) && python -m http.server 9000

readinessProbe:

```
exec:
command:
– cat
– /tmp/healthy
initialDelaySeconds: 1
periodSeconds: 5
EOF

controlplane $ kubectl create -f readiness.yaml
deployment.apps/readiness created

controlplane $ kubectl get pods
NAME READY STATUS RESTARTS AGE
readiness-fd8d996dd-cfsdb 0/1 ContainerCreating 0 7s
readiness-fd8d996dd-sj8pl 0/1 ContainerCreating 0 7s

controlplane $ kubectl get pods
NAME READY STATUS RESTARTS AGE
readiness-fd8d996dd-cfsdb 0/1 Running 0 6m29s
readiness-fd8d996dd-sj8pl 0/1 Running 0 6m29s

controlplane $ kubectl exec -it readiness-fd8d996dd-cfsdb – curl localhost:9000/health
readiness-fd8d996dd-cfsdb
```

Наши контейнера отлично работают. Добавим в них трафик:

```
controlplane $ kubectl expose deploy readiness \
–-type=LoadBalancer \
–-name=readiness \
–-port=9000 \
–-target-port=9000
service/readiness exposed

controlplane $ kubectl get svc readiness
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
readiness LoadBalancer 10.98.36.51 < pending> 9000:32355/TCP 98s

controlplane $ curl localhost:9000

controlplane $ for i in {1..5}; do curl $IP:9000/health; done
1
2
3
4
5
```

Каждый контейнер имеет задержку. Проверим, что будет, если один из контейнеров перезапустить – будет ли на него перенаправляться трафик:

```
controlplane $ kubectl get pods
NAME READY STATUS RESTARTS AGE
readiness-5dd64c6c79-9vq62 0/1 CrashLoopBackOff 6 15m
readiness-5dd64c6c79-sblvl 0/1 CrashLoopBackOff 6 15m
```

kubectl exec -it .... -c .... bash -c "rm -f healt"

controlplane $ for i in {1..5}; do echo $i; done
1
2
3
4
5

controlplane $ kubectl delete deploy readiness
deployment.apps "readiness" deleted

Рассмотрим ситуацию, когда контейнер становится временно недоступен для работы:

(hostname > health) && (python -m http.server 9000 &) && sleep 60 && rm health && sleep 60 && (hostname > health) sleep 6000

/bin/sh -c sleep 60 && (python -m http.server 9000 &) && PID=$! && sleep 60 && kill -9 $PID

По умолчанию, в состояние Running контейнер переходит по завершения выполнения скриптов в Dockerfile и запуску скрипта, заданного в инструкции CMD, если он переопределён в конфигурации в разделе Command. Но, на практике, если у нас база данных, ей нужно ещё подняться (прочитать данные и перенести их оперативную память и другие действия), а это может занять значительно время, при этом она не будет отвечать на соединения, и другие приложения, хотя и прочитают в состоянии готовность принимать соединения не смогут этого сделать. Также, контейнер переходи в состояние Feils, когда падает главный процесс в контейнере. В случае с базой данных, она может бесконечно пытаться выполнить неправильный запрос и не сможет отвечать на приходящие запросы, при этом контейнер не буде перезапущен, так как формально демон (сервер) базы данных не упал. Для этих случаев и придуманы два идентификатора: readinessProbe и livenessProbe, проверяющих по кастомному скрипту или HTTP запросу переход контейнера в рабочее состояние или его неисправность.

esschtolts@cloudshell:~/bitrix (essch)$ cat health_check.yaml
apiVersion: v1
kind: Pod
metadata:
labels:
test: healtcheck
name: healtcheck
spec:
containers:
– name: healtcheck
image: alpine:3.5
args:
– /bin/sh
– -c
– sleep 12; touch /tmp/healthy; sleep 10; rm -rf /tmp/healthy; sleep 60
readinessProbe:
exec:
command:
– cat
– /tmp/healthy

initialDelaySeconds: 5
periodSeconds: 5
livenessProbe:
exec:
command:
– cat
– /tmp/healthy
initialDelaySeconds: 15
periodSeconds: 5

Контейнер стартует через 3 секунды и через 5 секунд начинается проверка на готовность каждые 5 секунд. На второй проверке (на 15 секунде жизни) проверка на готовность cat /tmp/healthy увенчается успехом. В это время начинает осуществляться проверка на работоспособность livenessProbe и на второй проверке (на 25 секунде) заканчивается ошибкой, после чего контейнер признаётся не рабочим и пересоздается.

esschtolts@cloudshell:~/bitrix (essch)$ kubectl create -f health_check.yaml && sleep 4 && kubectl get

pods && sleep 10 && kubectl get pods && sleep 10 && kubectl get pods
pod "liveness-exec" created
NAME READY STATUS RESTARTS AGE
liveness-exec 0/1 Running 0 5s
NAME READY STATUS RESTARTS AGE
liveness-exec 0/1 Running 0 15s
NAME READY STATUS RESTARTS AGE
liveness-exec 1/1 Running 0 26s
esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods
NAME READY STATUS RESTARTS AGE
liveness-exec 0/1 Running 0 53s
esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods
NAME READY STATUS RESTARTS AGE
liveness-exec 0/1 Running 0 1m
esschtolts@cloudshell:~/bitrix (essch)$ kubectl get pods
NAME READY STATUS RESTARTS AGE
liveness-exec 1/1 Running 1 1m

Kubernetes предоставляет ещё и startup, переделяющий момент, когда можно включить readiness и liveness пробы в работу. Это полезно в том случае, если, к примеру, мы скачиваем приложение. Рассмотрим более подробно. Для эксперимента возьмём www.katacoda.com/courses/Kubernetes/playground и Python. Существует TCP, EXEC и HTTP, но лучше использовать HTTP, так как EXEC порождает процессы и может оставлять их в виде "зомби процессов". К тому же, если сервер обеспечивает взаимодействие по HTTP, то именно по нему и нужно проверять (https://www.katacoda.com/courses/kubernetes/playground):

controlplane $ kubectl version –short
Client Version: v1.18.0
Server Version: v1.18.0

cat << EOF > job.yaml
apiVersion: v1
kind: Pod
metadata:
name: healt

```
spec:
containers:
– name: python
image: python
command: ['sh', '-c', 'sleep 60 && (echo "work" > health) && sleep 60 && python -m http.server 9000']
readinessProbe:
httpGet:
path: /health
port: 9000
initialDelaySeconds: 3
periodSeconds: 3
livenessProbe:
httpGet:
path: /health
port: 9000
initialDelaySeconds: 3
periodSeconds: 3
startupProbe:
exec:
command:
– cat
– /health
initialDelaySeconds: 3
periodSeconds: 3
restartPolicy: OnFailure
EOF

controlplane $ kubectl create -f job.yaml
pod/healt

controlplane $ kubectl get pods # ещё не загружен
NAME READY STATUS RESTARTS AGE
healt 0/1 Running 0 11s

controlplane $ sleep 30 && kubectl get pods # ещё не загружен, но образ уже стянут
NAME READY STATUS RESTARTS AGE
healt 0/1 Running 0 51s

controlplane $ sleep 60 && kubectl get pods
NAME READY STATUS RESTARTS AGE
healt 0/1 Running 1 116s

controlplane $ kubectl delete -f job.yaml
pod "healt" deleted
```

Самодиагностика микро сервисного приложения

Рассмотрим работу probe на примере микро сервисного приложения bookinfo, входящего в состав Istio как пример: https://github.com/istio/istio/tree/master/samples/bookinfo. Демонстра-

ция будет в www.katacoda.com/courses/istio/deploy-istio-on-kubernetes. После разворачивания будет доступны

Управление инфраструктурой

Хотя, и у Kubernetes есть свой графический интерфейс – UI-дашборд, но кроме мониторинга и простейших действий не предоставляет. Больше возможностей даёт OpenShift, предоставляя совмещения графического и текстового создания. Полноценный продукт с сформированной экосистемой Google в Kubernetes не предоставляет, но предоставляет облачное решение – Google Cloud Platform. Однако, существуют и сторонние решения, такие как Open Shift и Rancher, позволяющие пользоваться полноценно через графический интерфейс на своих мощностях. При желании, конечно, можно синхронизироваться с облаком.

Каждый продукт, зачастую, не совместим с друг другом по API, единственным известным исключением является Mail. Cloud, в котором заявляется поддержка Open Shift. Но, существует стороннее решение, реализующее подход "инфраструктура как код" и поддерживающее API большинства известных экосистем – Terraform. Он, так же как Kubernetes, применяет концепция инфраструктура как код, но только не к контейнеризации, а к виртуальным машинам (серверам, сетям, дискам). Принцип Инфраструктура как код подразумевает наличии декларативной конфигурации – то есть описания результата без явного указания самих действий. При активации конфигурация (в Kubernetes это kubectl apply -f name_config .yml, а в Hashicorp Terraform – terraform apply) системы приводится в соответствие с конфигурационными файлами, при изменении конфигурации или инфраструктуры, инфраструктура, в конфликтующих частях с её декларацией приводится в соответствие, при этом сама система решает, как этого достичь, причём поведение может быть различным, например, при изменении метаинформации в POD – она будет изменена, а при изменении образа – POD будет удалён и создан уже новым. Если, до этого мы создавали серверную инфраструктуру для контейнеров в императивном виде с помощью команды gcloud публичного облака Google Cloud Platform (GCP), то теперь рассмотрим, как создать аналогичное с помощью конфигурация в декларативном описании паттерна инфраструктура как код с помощью универсального инструмента Terraform, поддерживающего облако GCP.

Terraform не возник на пустом месте, а стал продолжением длинной истории появления программных продуктов конфигурирования и управления серверной инфраструктуры, перечислю в прядке появления и перехода:

** CFN;

** Pupet;

** Chef;

** Ansible;

** Облачные AWS API, Kubernetes API;

* IasC: Terraform не зависит от типа инфраструктуры (поддерживает более 120 провайдеров, среди которых не только облака), в отличии от ведровых аналогов, поддерживающих только себя: CloudFormation для Amazon WEB Service, Azure Resource Manager для Microsoft Azure, Google Cloud Deployment Manager от Google Cloud Engine.

CloudFormation создан Amazon и предназначен для негоже, также полностью интегрирован в CI/CD его инфраструктуры размещением на AWS S3, что усложняет версионирование через GIT. Мы рассмотрим платформой независимый Terraform: синтаксис базовой функциональности един, а специфичная подключается через сущности Провайдеры (https://www.terraform.io/docs/providers/index. html). Terraform – один бинарный файл, поддерживает огромное количество провайдеров, и, конечно же, таких как AWS и GCE. Terraform, как и большинство продуктов от Hashicorp написаны на Go и представляют из себя один бинарный исполняемый файл, не требует установки, достаточно просто скачать его в папку Linux:

(agile-aleph-203917)$ wget https://releases.hashicorp.com/terraform/0.11.13/terraform_0.11.13_linux_amd64.zip

(agile-aleph-203917)$ unzip terraform_0.11.13_linux_amd64.zip -d .

(agile-aleph-203917)$ rm -f terraform_0.11.13_linux_amd64.zip

(agile-aleph-203917)$ ./terraform version

Terraform v0.11.13

Он поддерживает разбиение на модули, которые можно написать самому или использовать готовые (https://registry.terraform.io/browse?offset=27&provider=google). Для оркестрации и поддержки изменений в зависимостях можно воспользоваться Terragrunt (https://davidbegin.github.io/terragrunt/), например:

```
terragrant = {
terraform {
source = "terraform-aws-modules/…"
}
dependencies {
path = ["..network"]
}
}
name = "…"
ami = "…"
instance_type = "t3.large"
```

Единая семантика для разных провайдеров (AWS, GCE, Яндекс. Облако и многих других) конфигураций, что позволяет создать трансцендентную инфраструктуру, например, постоянные нагруженные сервисы расположить для экономии на собственных мощностях, а переменно нагружены (например, в период акций) в публичных облаках. Благодаря тому, что управление декларативно и может быть описана файлами (IaC, инфраструктура как код), создание инфраструктуры можно добавить в pipeline CI/CD (разработка, тестирование, доставка, всё автоматически и с контролем версий). Без CI/CD поддерживается блокировка файла конфигурации для предотвращения конкурентного его редактирования при совместной работе. инфраструктура создаётся не скриптом, а приводится в соответствие с конфигурацией, которая декларативна и не может содержать логики, хотя, можно и внедрить в неё BASH-скрипты и использовать Conditions (термальный оператор) для разных окружений.

Terraform будет читать все файлы в текущем каталоге с расширением .tf в Hachicort Configuraiton Language (HCL) формате или .tf. json в JSON формате. Часто, вместо одного файл его разделяют на несколько, как минимум на два: первый содержащий конфигурацию, второй – приватные данные, вынесенные в переменными.

Для демонстрации возможностей Terraform мы создадим репозиторий GitHub из-за его простоты авторизации и API. Сперва получим токен, сгенерирована в WEB-интерфейсе: SettingsDeveloper sittings -> Personal access token –> Generate new token и установив разрешения. Ничего не будем создавать, просто проверим подключение:

```
(agile-aleph-203917)$ ls *.tf
main.tf variables.tf

$ cat variables.tf
variable "github_token" {
default = "630bc9696d0b2f4ce164b1cabb118eaaa1909838"
}
$ cat main.tf
provider "github" {
```

token = "${var.github_token}"
}

(agile-aleph-203917)$ ./terraform init
(agile-aleph-203917)$ ./terraform apply

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Теперь, создадим управляющий аккаунт Settings –> Organizations –> New organization –> Create organization.. Используя: API Terraform по создания репозитория www.terraform.io/docs/providers/github/r/repository. html добавим в конфиг описание репозитория:

(agile-aleph-203917)$ cat main.tf
provider "github" {
token = "${var.github_token}"
}
resource "github_repository" "terraform_repo" {
name = "terraform-repo"
description = "my terraform repo"
auto_init = true
}

Теперь осталось применить, посмотреть с планом создания репозитория, согласиться с ним:

(agile-aleph-203917)$ ./terraform apply
provider.github.organization
The GitHub organization name to manage.

Enter a value: essch2

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ github_repository.terraform_repo
id:<computed>
allow_merge_commit: "true"
allow_rebase_merge: "true"
allow_squash_merge: "true"
archived: "false"
auto_init: "true"
default_branch: <computed>
description: "my terraform repo"
etag: <computed>
full_name: <computed>
git_clone_url: <computed>
html _url: <computed>
http_clone_url: <computed>
name: "terraform-repo"
ssh_clone_url: <computed>

svn_url: <computed>

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

github_repository.terraform_repo: Creating…
allow_merge_commit: "" => "true"
allow_rebase_merge: "" => "true"
allow_squash_merge: "" => "true"
archived: "" => "false"
auto_init: "" => "true"
default_branch: "" => "<computed>"
description: "" => "my terraform repo"
etag: "" => "<computed>"
full_name: "" => "<computed>"
git_clone_url: "" => "<computed>"
html_url: "" => "<computed>"
http_clone_url: "" => "<computed>"
name: "" => "terraform-repo"
ssh_clone_url: "" => "<computed>"
svn_url: "" => "<computed>"
github_repository.terraform_repo: Creation complete after 4s (ID: terraform-repo)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed

Теперь можно наблюдать пустой репозиторий terraform-repo в WEB-интерфейсе. При повторном применении репозиторий не будет создан, так как Terraform применяет только изменения, который не было:

(agile-aleph-203917)$ ./terraform apply
provider.github.organization
The GITHub organization name to manage.
Enter a value: essch2
github_repository.terraform_repo: Refreshing state… (ID: terraform-repo)
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

А вот если я изменю название, то Terraform постарается применить изменения название через удаление и создание нового с актуальным названием. Важно заметить, что любые данные, которые мы бы запушили бы в этот репозиторий после смены названия были бы удалены. Для проверки, как будет производиться обновления можно предварительно спросить перечень производимых действий командой ./Terraform plane. И так, приступим:

(agile-aleph-203917)$ cat main.tf
provider "github" {
token = "${var.github_token}"
}
resource "github_repository" "terraform_repo" {
name = "terraform-repo2"

```
description = "my terraform repo"
auto_init = true
}

(agile-aleph-203917)$ ./terraform plan
provider.github.organization
The GITHub organization name to manage.

Enter a value: essch

Refreshing Terraform state in-memory prior to plan…
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

github_repository.terraform_repo: Refreshing state… (ID: terraform-repo)

--

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ github_repository.terraform_repo
id:<computed>
allow_merge_commit: "true"
allow_rebase_merge: "true"
allow_squash_merge: "true"
archived: "false"
auto_init: "true"
default_branch: <computed>
description: "my terraform repo"
etag: <computed>
full_name: <computed>
git_clone_url: <computed>
html_url: <computed>
http_clone_url: <computed>
name: "terraform-repo2"
ssh_clone_url: <computed>
svn_url: <computed>

"terraform apply" is subsequently run.
esschtolts@cloudshell:~/terraform (agile-aleph-203917)$ ./terraform apply
provider.github.organization
The GITHub organization name to manage.
Enter a value: essch2
github_repository.terraform_repo: Refreshing state… (ID: terraform-repo)
An execution plan has been generated and is shown below.
```

Resource actions are indicated with the following symbols:
–/+ destroy and then create replacement
Terraform will perform the following actions:
–/+ github_repository.terraform_repo (new resource required)
id:"terraform-repo" => <computed> (forces new resource)
allow_merge_commit: "true" => "true"
allow_rebase_merge: "true" => "true"
allow_squash_merge: "true" => "true"
archived: "false" => "false"
auto_init: "true" => "true"
default_branch: "master" => <computed>
description: "my terraform repo" => "my terraform repo"
etag: "W/\"a92e0b300d8c8d2c869e5f271da6c2ab\"" => <computed>
full_name: "essch2/terraform-repo" => <computed>
git_clone_url: "git://github.com/essch2/terraform-repo.git" => <computed>
html_url: "https://github.com/essch2/terraform-repo" => <computed>
http_clone_url: "https://github.com/essch2/terraform-repo.git" => <computed>
name: "terraform-repo" => "terraform-repo2" (forces new resource)
ssh_clone_url: "git@github.com:essch2/terraform-repo.git" => <computed>
svn_url: "https://github.com/essch2/terraform-repo" => <computed>
Plan: 1 to add, 0 to change, 1 to destroy.
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value: yes
github_repository.terraform_repo: Destroying… (ID: terraform-repo)
github_repository.terraform_repo: Destruction complete after 0s
github_repository.terraform_repo: Creating…
allow_merge_commit: "" => "true"
allow_rebase_merge: "" => "true"
allow_squash_merge: "" => "true"
archived: "" => "false"
auto_init: "" => "true"
default_branch: "" => "<computed>"
description: "" => "my terraform repo"
etag: "" => "<computed>"
full_name: "" => "<computed>"
git_clone_url: "" => "<computed>"
html_url: "" => "<computed>"
http_clone_url: "" => "<computed>"
name: "" => "terraform-repo2"
ssh_clone_url: "" => "<computed>"
svn_url: "" => "<computed>"
github_repository.terraform_repo: Creation complete after 5s (ID: terraform-repo2)
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.

Из соображений наглядности, я создал большую брешь в безопасности – я поместил токен в конфигурационный файл, а значит и в репозиторий, и теперь любой, кото может получить доступ к нему сможет удалить все репозитории. Terraform предоставляет несколько спо-

собов задания переменных, кроме использованного. Я же просто пересоздам токен и перекрою его переданным в командной строке:

(agile-aleph-203917)$ rm variables.tf

(agile-aleph-203917)$ sed -i 's/terraform-repo2/terraform-repo3/' main.tf

./terraform apply -var="github_token=f7602b82e02efcbae7fc915c16eeee518280cf2a"

Создание инфраструктуры в GCP с Terraform

Каждые облака имеют свои наборы сервисов, свои API для них. Чтобы упростить переход с одного облака как для сотрудников с точки зрения изучения, так и точки зрения переписывания – существуют универсальные библиотеки, реализующие паттерн Фасад. Под фасадом понимаются универсальный API, срывающий особенности систем, лежащих за ним.

Одним из представителем фасадов API облаков является KOPS. KOPS – тулза для деплоя Kubernetes в GCP, AWS и Azure. KOPS похож на Kubectl – представляет из себя бинарника, может создавать как командами, так и по YML-конфигу, имеет схожий синтаксис, но в отличии от Kubectl – создаёт не POD, а нод кластера. Другим примером, является Terraform, специализирующийся именно на развёртывании по конфигурации для придерживания концепции IasC.

Для создания инфраструктуры нам понадобится токен, его создаётся в GCP для сервисного аккаунта, которому выдаются доступы. Для этого я перешёл по пути: IAM и администрирование –> Сервисные аккаунты –> Создать сервисный аккаунт и при создании выбыла роль Владелец (полный доступ для тестовых целей), создал ключ кнопкой Создать ключ в JSON формате и скачанный ключ я переименовал в Key. JSON. Для описания инфраструктуры я воспользовался документацией www.terraform.io/docs/providers/google/index.html:

```
(agil7e-aleph-20391)$ cat main.tf
provider "google" {
credentials = "${file("key.json")}"
project = "agile-aleph-203917"
region = "us-central1"
}
resource "google_compute_instance" "terraform" {
name = "terraform"
machine_type = "n1-standard-1"
zone = "us-central1-a"
boot_disk {
initialize_params {
image = "debian-cloud/debian-9"
}
}
network_interface {
network = "default"
}
}
```

Проверим права пользователя:

```
(agile-aleph-203917)$ gcloud auth list
Credentialed Accounts
ACTIVE ACCOUNT
* esschtolts@gmail.com
To set the active account, run:
$ gcloud config set account `ACCOUNT`
```

Выберем проект в качестве текущего (можно составить текущий по умолчанию):

```
$ gcloud config set project agil7e-aleph-20391;
(agil7e-aleph-20391)$ ./terraform init | grep success
Terraform has been successfully initialized!
```

Теперь создам один инстанс в WEB-консоли, предварительно скопировав ключ в файл key.json в каталог с Terraform:

```
machine_type: "" => "n1-standard-1"
metadata_fingerprint: "" => "<computed>"
name: "" => "terraform"
network_interface.#: "" => "1"
network_interface.0.address: "" => "<computed>"
network_interface.0.name: "" => "<computed>"
network_interface.0.network: "" => "default"
network_interface.0.network_ip: "" => "<computed>"
network_interface.0.network: "" => "default"
project: "" => "<computed>"
scheduling.#: "" => "<computed>"
self_link: "" => "<computed>"
tags_fingerprint: "" => "<computed>"
zone: "" => "us-central1-a"
google_compute_instance.terraform: Still creating… (10s elapsed)
google_compute_instance.terraform: Still creating… (20s elapsed)
google_compute_instance.terraform: Still creating… (30s elapsed)
google_compute_instance.terraform: Still creating… (40s elapsed)
google_compute_instance.terraform: Creation complete after 40s (ID: terraform)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Все, мы создали инстанс сервера. Теперь удалим его:

```
~/terraform (agil7e-aleph-20391)$ ./terraform apply
google_compute_instance.terraform: Refreshing state… (ID: terraform)
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
– destroy
Terraform will perform the following actions:
– google_compute_instance.terraform
Plan: 0 to add, 0 to change, 1 to destroy.
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value: yes
google_compute_instance.terraform: Destroying… (ID: terraform)
google_compute_instance.terraform: Still destroying… (ID: terraform, 10s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 20s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 30s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 40s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 50s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 1m0s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 1m10s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 1m20s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 1m30s elapsed)
```

google_compute_instance.terraform: Still destroying… (ID: terraform, 1m40s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 1m50s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 2m0s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 2m10s elapsed)
google_compute_instance.terraform: Still destroying… (ID: terraform, 2m20s elapsed)
google_compute_instance.terraform: Destruction complete after 2m30s
Apply complete! Resources: 0 added, 0 changed, 1 destroyed.

Создание инфраструктуры в AWS

Для создания конфигурации AWS кластера, создадим под него отдельную папку, а предыдущую – в параллельную:

esschtolts@cloudshell:~/terraform (agil7e-aleph-20391)$ mkdir gcp
esschtolts@cloudshell:~/terraform (agil7e-aleph-20391)$ mv main.tf gcp/main.tf
esschtolts@cloudshell:~/terraform (agil7e-aleph-20391)$ mkdir aws
esschtolts@cloudshell:~/terraform (agil7e-aleph-20391)$ cd aws

Role – аналог пользователя, только не для людей, а для сервисов, таких как AWS, а в нашем случае – это сервера EKS. Но мне видится аналогом ролей не пользователи, а группы, например, группа создания кластера, группа работы с БД и т.д. К серверу можно прикрепить только одну роль, при этом роль может содержать множество прав (Polices). В результате нам не нужно работать ни с логинами и паролями, ни с токенами, ни с сертификатами: хранить, передавать, ограничивать доступ, передавать – мы только указываем в WEB панели инструментов (IMA) или с помощью API (а производно в конфигурации) права. Нашему кластеру необходимы эти права, чтобы он смог само настраиваться и реплицироваться, так как он состоит из стандартных сервисов AWS. Для управления самими компонентами кластера AWS EC2 (сервера), AWS ELB (Elastic Load Balancer, балансировщик) и AWS KMS (Key Management Service, менеджер и шифрование ключей) нужен доступ AmazonEKSClusterPolicy, для мониторинга AmazonEKSServicePolicy с помощью компонентов CloudWatch Logs (мониторинг по логам), Route 53 (создание сети в зоне), IAM (управления правами). Я не стал описывать роль в конфиге и создал её через IAM по документации: https://docs.aws.amazon.com/eks/latest/userguide/service_IAM_role. html #create-service-role.

Для большей надёжности ноды Kubernetes кластера должны располагаться в разных зонах, то есть дата центрах. В каждом регионе, содержится несколько зон для поддержания отказоустойчивости, при этот сохраняя минимальных летенси (время ответа сервера) для местного населения. Важно заметить, что некоторые регионы могут быть представлены в нескольких экземплярах в рамках одной страны, например, US-east-1 в US East (N. Virginia) и US-east-2 в US East (Ohio) – обозначение регионов ведётся цифрами. Пока создание кластера EKS доступно только US-east зоне.

VPC для разработчика, простейшем варианте, сводится к наименованию подсети как конкретный ресурс.

Напишем конфигурацию в соответствии с документацией www.terraform.io/docs/providers/aws/r/eks_cluster. html :

esschtolts@cloudshell:~/terraform/aws (agile-aleph-203917)$ cat main.tf

```
provider "aws" {
access_key = "${var.token}"
secret_key = "${var.key}"
region = "us-east-1"
}

# Params
```

```
variable "token" {
default = ""
}
variable "key" {
default = ""
}

# EKS

resource "aws_eks_cluster" "example" {
enabled_cluster_log_types = ["api", "audit"]
name = "exapmle"
role_arn = "arn:aws:iam::177510963163:role/ServiceRoleForAmazonEKS2"

vpc_config {
subnet_ids = ["${aws_subnet.subnet_1.id}", "${aws_subnet.subnet_2.id}"]
}
}

output "endpoint" {
value = "${aws_eks_cluster.example.endpoint}"
}

output "kubeconfig-certificate-authority-data" {
value = "${aws_eks_cluster.example.certificate_authority.0.data}"
}

# Role

data "aws_iam_policy_document" "eks-role-policy" {
statement {
actions = ["sts:AssumeRole"]

principals {
type = "Service"
identifiers = ["eks.amazonaws.com"]
}
}
}

resource "aws_iam_role" "tf_role" {
name = "tf_role"
assume_role_policy = "${data.aws_iam_policy_document.eks-role-policy.json}"
tags = {
tag-key = "tag-value"
}
}

resource "aws_iam_role_policy_attachment" "attach-cluster" {
```

```
role = "tf_role"
policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
}

resource "aws_iam_role_policy_attachment" "attach-service" {
role = "tf_role"
policy_arn = "arn:aws:iam::aws:policy/AmazonEKSServicePolicy"
}

# Subnet

resource "aws_subnet" "subnet_1" {
vpc_id = "${aws_vpc.main.id}"
cidr_block = "10.0.1.0/24"
availability_zone = "us-east-1a"

tags = {
Name = "Main"
}
}

resource "aws_subnet" "subnet_2" {
vpc_id = "${aws_vpc.main.id}"
cidr_block = "10.0.2.0/24"
availability_zone = "us-east-1b"

tags = {
Name = "Main"
}
}

resource "aws_vpc" "main" {
cidr_block = "10.0.0.0/16"
}
```

Через 9 минут 44 секунды я получил готовую самоподдерживающуюся инфраструктуру для кластера Kubernetes:

```
esschtolts@cloudshell:~/terraform/aws (agile-aleph-203917)$ ./../terraform apply -var="token=AKIAJ4SYCNH2XVSHNN3A" -var="key=huEWRslEluynCXBspsul3AkKlinAlR9+MoU1ViY7"
```

Теперь удалим (у меня заняло 10 минуты 23 секунды):

```
esschtolts@cloudshell:~/terraform/aws (agile-aleph-203917)$ ./../terraform destroy -var="token=AKIAJ4SYCNH2XVSHNN3A" -var="key=huEWRslEluynCXBspsul3AkKlinAlR9+MoU1ViY7"
```

```
Destroy complete! Resources: 7 destroyed.
```

Налаживание процесса CI/CD

Amazon предоставляет (aws.amazon.com/ru/devops/) большой спектр DevOps инструментов, оформленных в облачную инфраструктуру:

* AWS Code Pipeline – сервис позволяет создать из набора сервисов в визуальном редакторе цепочку этапов, через которые должен пройти код, прежде чем он попадёт на продакшн, например, сборку и тестирование.

* AWS Code Build – сервис предоставляет авто масштабирующую очередь сборки, что может потребоваться для компилируемых языков программирования, когда при добавлении фич или внесения изменений необходимо длительная пере компиляция всего приложения, при использовании одного сервера становится узким местом при выкатке изменений.

* AWS Code Deploy – автоматизирует развёртывание и возвращение к предыдущему состоянию продакшна в случаи ошибок.

* AWS CodeStar – сервис объединяет в себе основные возможности предыдущих сервисов.

Настраиваем удалённое управление

сервер артефактов

aws s3 ls s3://name_backet aws s3 sync s3://name_backet name_fonder –exclude *.tmp # в папку будет скачены файлы из бакета, например, сайт

Теперь, нам нужно скачать плагин работы с AWS:

esschtolts@cloudshell:~/terraform/aws (agile-aleph-203917)$ ./../terraform init | grep success

Terraform has been successfully initialized!

Теперь нам нужно получить доступы к AWS, для того кликаем по имени вашего пользователя шапки WEB-интерфейса, кроме My account, появится пункт My Security Credentials, выбрав который, переходим Access Key –> Create New Access Key. Создадим EKS (Elastic Kuberntes Service):

esschtolts@cloudshell:~/terraform/aws (agile-aleph-203917)$ ./../terraform apply

–var="token=AKIAJ4SYCNH2XVSHNN3A" -var="key=huEWRslEluynCXBspsul3AkKlinAlR9+MoU1ViY7"

Удаляем все:

$ ../terraform destroy

Создание кластера в GCP

node pool – объединение нод в кластер с

```
resource "google_container_cluster" "primary" {
name = "tf"
location = "us-central1"
$ cat main.tf # состояние конфигурации
terraform {
required_version = "> 0.10.0"
}
terraform {
backend "s3" {
bucket = "foo-terraform"
key = "bucket/terraform.tfstate"
region = "us-east-1"
encrypt = "true"
}
}
$ cat cloud.tf# конфигурации облака
provider "google" {
token = "${var.hcloud_token}"
}
```

```
$ cat variables.tf# переменные и получение токенов
variable "hcloud_token" {}

$ cat instances.tf# создание ресурсов
resource "hcloud_server" "server" { ....

$ terraform import aws_acm_certificate.cert arn:aws:acm:eu-central-1:123456789012:certificate/7e7a28d2-163f-4b8f-b9cd-822f96c08d6a
$ terraform init # Инициализация конфигов
$ terraform plan # Проверка действий
$ terraform apply # Запуск действий
Отладка:
essh@kubernetes-master:~/graylog$ sudo docker run –name graylog –link graylog_mongo:mongo –link graylog_elasticsearch:elasticsearch \
–p 9000:9000 -p 12201:12201 -p 1514:1514 \
–e GRAYLOG_HTTP_EXTERNAL_URI="http://127.0.0.1:9000/" \
–d graylog/graylog:3.0
0f21f39192440d9a8be96890f624c1d409883f2e350ead58a5c4ce0e91e54c9d
docker: Error response from daemon: driver failed programming external connectivity on endpoint graylog (714a6083b878e2737bd4d4577d1157504e261c03cb503b6394cb844466fb4781): Bind for 0.0.0.0:9000 failed: port is already allocated.
essh@kubernetes-master:~/graylog$ sudo netstat -nlp | grep 9000
tcp6 0 0 :::9000:::* LISTEN 2505/docker-proxy
essh@kubernetes-master:~/graylog$ docker rm graylog
graylog
essh@kubernetes-master:~/graylog$ sudo docker run –name graylog –link graylog_mongo:mongo –link graylog_elasticsearch:elasticsearch \
–p 9001:9000 -p 12201:12201 -p 1514:1514 \
–e GRAYLOG_HTTP_EXTERNAL_URI="http://127.0.0.1:9001/" \
–d graylog/graylog:3.0
e5aefd6d630a935887f494550513d46e54947f897e4a64b0703d8f7094562875
https://blog.maddevs.io/terrafom-hetzner-a2f22534514b
Для примера, создадим один инстанс:
$ cat aws/provider.tf
provider "aws" {
region = "us-west-1"
}
resource "aws_instance" "my_ec2" {
ami = "${data.aws_ami.ubuntu.id}"
instance_type = "t2.micro"
}

$ cd aws
$ aws configure
$ terraform init
$ terraform apply –auto-approve
$ cd ..
provider "aws" {
```

```
region = "us-west-1"
}
resource "aws_sqs_queue" "terraform_queue" {
name = "terraform-queue"
delay_seconds = 90
max_message_size = 2048
message_retention_seconds = 86400
receive_wait_time_seconds = 10
}
data "aws_route53_zone" "vuejs_phalcon" {
name = "test.com."
private_zone = true
}
resource "aws_route53_record" "www" {
zone_id = "${data.aws_route53_zone.vuejs_phalcon.zone_id}"
name = "www.${data.aws_route53_zone.selected.name}"
type = "A"
ttl = "300"
records = ["10.0.0.1"]
}
resource "aws_elasticsearch_domain" "example" {
domain_name = "example"
elasticsearch_version = "1.5"
cluster_config {
instance_type = "r4.large.elasticsearch"
}
snapshot_options {
automated_snapshot_start_hour = 23
}
}
resource "aws_eks_cluster" "eks_vuejs_phalcon" {
name = "eks_vuejs_phalcon"
role_arn = "${aws_iam_role.eks_vuejs_phalcon.arn}"

vpc_config {
subnet_ids = ["${aws_subnet.eks_vuejs_phalcon.id}", "${aws_subnet.example2.id}"]
}
}
output "endpoint" {
value = "${aws_eks_cluster.eks_vuejs_phalcon.endpoint}"
}
output "kubeconfig-certificate-authority-data" {
value = "${aws_eks_cluster.eks_vuejs_phalcon.certificate_authority.0.data}"
}
provider "google" {
credentials = "${file("account.json")}"
project = "my-project-id"
region = "us-central1"
}
```

```
resource "google_container_cluster" "primary" {
name = "my-gke-cluster"
location = "us-central1"
remove_default_node_pool = true
initial_node_count = 1
master_auth {
username = ""
password = ""
}
}
output "client_certificate" {
value = "${google_container_cluster.primary.master_auth.0.client_certificate}"
}
output "client_key" {
value = "${google_container_cluster.primary.master_auth.0.client_key}"
}
output "cluster_ca_certificate" {
value = "${google_container_cluster.primary.master_auth.0.cluster_ca_certificate}"
}
$ cat deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
name: phalcon_vuejs
namespace: development
spec:
selector:
matchLabels:
app: vuejs
replicas: 1
template:
metadata:
labels:
app: vuejs
spec:
initContainers:
– name: vuejs_build
image: vuejs/ci
volumeMounts:
– name: app
mountPath: /app/public
command:
– /bin/bash
– -c
– |
cd /app/public
git clone essch/vuejs_phalcon:1.0 .
npm test
npm build
```

containers:
– name: healtcheck
image: mileschou/phalcon:7.2-cli
args:
– /bin/sh
– -c
– cd /usr/src/app && git clone essch/app_phalcon:1.0 && touch /tmp/healthy && sleep 10 && php script.php
readinessProbe:
exec:
command:
– cat
– /tmp/healthy
initialDelaySeconds: 5
periodSeconds: 5
livenessProbe:
exec:
command:
– cat
– /tmp/healthy
initialDelaySeconds: 15
periodSeconds: 5
voumes:
– name: app
emptyDir: {}

Вот и мы создали AWS EC2 инстанс. Мы опустили указание ключей, так как AWS API уже авторизован и эта авторизация будет использоваться Terraform.

Также, для использования кода, Terraform поддерживает переменные, данные и модули. Создадим отдельную сеть:

resource "aws_vpc" "my_vpc" {
cidr_block = "190.160.0.0/16"
instance_target = "default"
}
resource "aws_subnet" "my_subnet" {
vpc_id = "${aws_vpc.my_vpc.id}"
cidr_block = "190.160.1.0/24"
}
$ cat gce/provider.tf
provider "google" {
credentials = "${file("account.json")}"
project = "my-project-id"
region = "us-central1"
}
resource "google_compute_instance" "default" {
name = "test"
machine_type = "n1-standard-1"
zone = "us-central1-a"
}
$ cd gce

```
$ terraform init
$ terraform apply
$ cd ..
```

Для распределенной работы поместим состояние в AWS S3 состояние инфраструктуры (так же можно помещать другие данные), но для безопасности в другой регион:

```
terraform {
backend "s3" {
bucket = "tfstate"
key = "terraform.tfstate"
region = "us-state-2"
}
}
provider "kubernetes" {
host = "https://104.196.242.174"
username = "ClusterMaster"
password = "MindTheGap"
}

resource "kubernetes_pod" "my_pod" {
spec {
container {
image = "Nginx:1.7.9"
name = "Nginx"
port {
container_port = 80
}
}
}
}
```

Команды:

terraform init # скачивание зависимостей в соответствии с конфигами, проверка их

terraform validate # проверка синтаксиса

terraform plan # детально посмотреть, как будет изменена инфраструктура и почему именно так, например,

будет ли изменена только мета информация у сервиса или будет пересоздан сам сервис, что часто недопустимо для баз данных.

terraform apply # применение изменений

Общая часть для всех провайдеров – ядро.

```
$ which aws
$ aws fonfigure # https://www.youtube.com/watch?v=IxA1IPypzHs

$ cat aws.tf
# https://www.terraform.io/docs/providers/aws/r/instance.html
resource "aws_instance" "ec2instance" {
ami = "${var.ami}"
instance_type = "t2.micro"
}

resource "aws_security_group" "instance_gc" {
```

```
…
}

$cat run.js
export AWS_ACCESS_KEY_ID="anaccesskey"
export AWS_SECRET_ACCESS_KEY="asecretkey"
export AWS_DEFAULT_REGION="us-west-2"
terraform plan
terraform apply

$ cat gce.tf # https://www.terraform.io/docs/providers/google/index.html#
# Google Cloud Platform Provider

provider "google" {
credentials = "${file("account.json")}"
project = "phalcon"
region = "us-central1"
}

#https://www.terraform.io/docs/providers/google/r/app_engine_application.html
resource "google_project" "my_project" {
name = "My Project"
project_id = "your-project-id"
org_id = "1234567"
}

resource "google_app_engine_application" "app" {
project = "${google_project.my_project.project_id}"
location_id = "us-central"
}

# google_compute_instance
resource "google_compute_instance" "default" {
name = "test"
machine_type = "n1-standard-1"
zone = "us-central1-a"

tags = ["foo", "bar"]

boot_disk {
initialize_params {
image = "debian-cloud/debian-9"
}
}

// Local SSD disk
scratch_disk {
}
```

```
network_interface {
network = "default"

access_config {
// Ephemeral IP
}
}

metadata = {
foo = "bar"
}

metadata_startup_script = "echo hi > /test.txt"

service_account {
scopes = ["userinfo-email", "compute-ro", "storage-ro"]
}
}
```

Расширяемость с помощью external resource, в качестве которого может быть скрипт на BASH:

```
data "external" "python3" {
program = ["Python3"]
}
```

Создание кластера машин с помощью Terraform

Создание кластера с помощью Terraform рассматривается в Создание инфраструктуры в GCP. Сейчас уделим больше внимания самому кластеру, а не инструментам по его созданию. Создам через панель администратора GCE проект (отображается в шапке интерфейса) node-cluster. Ключ для Kubernetes я скачал IAM и администрирование –> Сервисные аккаунты –> Создать сервисный аккаунт и при создании выбрал роль Владелец и положил в проект под названием kubernetes_key.JSON:

```
eSSH@Kubernetes-master:~/node-cluster$ cp ~/Downloads/node-cluster-243923-bbec410e0a83.JSON ./kubernetes_key.JSON
```

Скачал terraform:

```
essh@kubernetes-master:~/node-cluster$ wget https://releases.hashicorp.com/terraform/0.12.2/terraform_0.12.2_linux_amd64.zip >/dev/null 2>/dev/null
essh@kubernetes-master:~/node-cluster$ unzip terraform_0.12.2_linux_amd64.zip && rm -f terraform_0.12.2_linux_amd64.zip
Archive: terraform_0.12.2_linux_amd64.zip
inflating: terraform
essh@kubernetes-master:~/node-cluster$ ./terraform version
Terraform v0.12.2
```

Добавили провайдера GCE и запустил скачивание "драйверов" к нему:

```
essh@kubernetes-master:~/node-cluster$ cat main.tf
provider "google" {
credentials = "${file("kubernetes_key.json")}"
project = "node-cluster"
region = "us-central1"
```

}essh@kubernetes-master:~/node-cluster$ ./terraform init

Initializing the backend…

Initializing provider plugins…
– Checking for available provider plugins…
– Downloading plugin for provider "google" (terraform-providers/google) 2.8.0…

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "…" constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.google: version = "~> 2.8"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
Добавлю виртуальную машину:
essh@kubernetes-master:~/node-cluster$ cat main.tf
provider "google" {
credentials = "${file("kubernetes_key.json")}"
project = "node-cluster-243923"
region = "europe-north1"
}
resource "google_compute_instance" "cluster" {
name = "cluster"
zone = "europe-north1-a"
machine_type = "f1-micro"

boot_disk {
initialize_params {
image = "debian-cloud/debian-9"
}
}

network_interface {
network = "default"
access_config {}
}

```
essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# google_compute_instance.cluster will be created
+ resource "google_compute_instance" "cluster" {
+ can_ip_forward = false
+ cpu_platform = (known after apply)
+ deletion_protection = false
+ guest_accelerator = (known after apply)
+ id = (known after apply)
+ instance_id = (known after apply)
+ label_fingerprint = (known after apply)
+ machine_type = "f1-micro"
+ metadata_fingerprint = (known after apply)
+ name= "cluster"
+ project = (known after apply)
+ self_link = (known after apply)
+ tags_fingerprint = (known after apply)
+ zone= "europe-north1-a"

+ boot_disk {
+ auto_delete = true
+ device_name = (known after apply)
+ disk_encryption_key_sha256 = (known after apply)
+ source = (known after apply)

+ initialize_params {
+ image = "debian-cloud/debian-9"
+ size = (known after apply)
+ type = (known after apply)
}
}

+ network_interface {
+ address = (known after apply)
+ name = (known after apply)
+ network = "default"
+ network_ip = (known after apply)
+ subnetwork = (known after apply)
+ subnetwork_project = (known after apply)

+ access_config {
+ assigned_nat_ip = (known after apply)
```

```
+ nat_ip = (known after apply)
+ network_tier = (known after apply)
}
}

+ scheduling {
+ automatic_restart = (known after apply)
+ on_host_maintenance = (known after apply)
+ preemptible = (known after apply)

+ node_affinities {
+ key = (known after apply)
+ operator = (known after apply)
+ values = (known after apply)
}
}
}

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

google_compute_instance.cluster: Creating…
google_compute_instance.cluster: Still creating… [10s elapsed]
google_compute_instance.cluster: Creation complete after 11s [id=cluster]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Добавлю к ноде публичный статический IP-адрес и SSH-ключ:
essh@kubernetes-master:~/node-cluster$ ssh-keygen -f node-cluster
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in node-cluster.
Your public key has been saved in node-cluster.pub.
The key fingerprint is:
SHA256:vUhDe7FOzykE5BSLOIhE7Xt9o+AwgM4ZKOCW4nsLG58 essh@kubernetes-master
The key's randomart image is:
+–[RSA 2048]–+
|.o. +. |
|o. o . = . |
|* + o . = . |
|=* . . . + o |
|B + . . S * |
| = + o o X + . |
```

```
| o. = . + = + |
| .=… . . |
| ..E. |
+–[SHA256]–+
essh@kubernetes-master:~/node-cluster$ ls node-cluster.pub
node-cluster.pub
essh@kubernetes-master:~/node-cluster$ cat main.tf
provider "google" {
credentials = "${file("kubernetes_key.json")}"
project = "node-cluster-243923"
region = "europe-north1"
}

resource "google_compute_address" "static-ip-address" {
name = "static-ip-address"
}

resource "google_compute_instance" "cluster" {
name = "cluster"
zone = "europe-north1-a"
machine_type = "f1-micro"

boot_disk {
initialize_params {
image = "debian-cloud/debian-9"
}
}

metadata = {
ssh-keys = "essh:${file("./node-cluster.pub")}"
}

network_interface {
network = "default"
access_config {
nat_ip = "${google_compute_address.static-ip-address.address}"
}
}
}essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply
```

Проверим подключение SSH к серверу:

```
essh@kubernetes-master:~/node-cluster$ ssh -i ./node-cluster essh@35.228.82.222
The authenticity of host '35.228.82.222 (35.228.82.222)' can't be established.
ECDSA key fingerprint is SHA256:o7ykujZp46IF
+eu7SaIwXOlRRApiTY1YtXQzsGwO18A.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.228.82.222' (ECDSA) to the list of known hosts.
Linux cluster 4.9.0-9-amd64 #1 SMP Debian 4.9.168-1+deb9u2 (2019-05-13) x86_64

The programs included with the Debian GNU/Linux system are free software;
```

the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
essh@cluster:~$ ls
essh@cluster:~$ exit
logout
Connection to 35.228.82.222 closed.
Установим пакеты:
essh@kubernetes-master:~/node-cluster$ curl https://sdk.cloud.google.com | bash
essh@kubernetes-master:~/node-cluster$ exec -l $SHELL
essh@kubernetes-master:~/node-cluster$ gcloud init
Выберем проект:
You are logged in as: [esschtolts@gmail.com].

Pick cloud project to use:
[1] agile-aleph-203917
[2] node-cluster-243923
[3] essch
[4] Create a new project
Please enter numeric choice or text value (must exactly match list
item):
Please enter a value between 1 and 4, or a value present in the list: 2

Your current project has been set to: [node-cluster-243923].
Выберем зону:
[50] europe-north1-a
Did not print [12] options.
Too many options [62]. Enter "list" at prompt to print choices fully.
Please enter numeric choice or text value (must exactly match list
item):
Please enter a value between 1 and 62, or a value present in the list: 50
essh@kubernetes-master:~/node-cluster$ PROJECT_I="node-cluster-243923"
essh@kubernetes-master:~/node-cluster$ echo $PROJECT_I
node-cluster-243923
essh@kubernetes-master:~/node-cluster$ export GOOGLE_APPLICATION_CREDENTIALS=$HOME/node-cluster/kubernetes_key.json
essh@kubernetes-master:~/node-cluster$ sudo docker-machine create –driver google –google-project $PROJECT_ID vm01
sudo export GOOGLE_APPLICATION_CREDENTIALS=$HOME/node-cluster/kubernetes_key.json docker-machine create –driver google –google-project $PROJECT_ID vm01
// https://docs.docker.com/machine/drivers/gce/
// https://github.com/docker/machine/issues/4722
essh@kubernetes-master:~/node-cluster$ gcloud config list
[compute]
region = europe-north1
zone = europe-north1-a
[core]

```
account = esschtolts@gmail.com
disable_usage_reporting = False
project = node-cluster-243923

Your active configuration is: [default]
```

Добавим копирование файла и выполнение скрипта:

```
essh@kubernetes-master:~/node-cluster$ cat main.tf
provider "google" {
credentials = "${file("kubernetes_key.json")}"
project = "node-cluster-243923"
region = "europe-north1"
}

resource "google_compute_address" "static-ip-address" {
name = "static-ip-address"
}

resource "google_compute_instance" "cluster" {
name = "cluster"
zone = "europe-north1-a"
machine_type = "f1-micro"

boot_disk {
initialize_params {
image = "debian-cloud/debian-9"
}
}

metadata = {
ssh-keys = "essh:${file("./node-cluster.pub")}"
}

network_interface {
network = "default"
access_config {
nat_ip = "${google_compute_address.static-ip-address.address}"
}
}
}

resource "null_resource" "cluster" {

triggers = {
cluster_instance_ids = "${join(",", google_compute_instance.cluster.*.id)}"
}

connection {
host = "${google_compute_address.static-ip-address.address}"
```

```
type = "ssh"
user = "essh"
timeout = "2m"
private_key = "${file("~/node-cluster/node-cluster")}"
# agent = "false"
}

provisioner "file" {
source = "client.js"
destination = "~/client.js"
}

provisioner "remote-exec" {
inline = [
"cd ~ && echo 1 > test.txt"
]
}

essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply
google_compute_address.static-ip-address: Creating…
google_compute_address.static-ip-address: Creation complete after 5s [id=node-cluster-243923/europe-north1/static-ip-address]
google_compute_instance.cluster: Creating…
google_compute_instance.cluster: Still creating… [10s elapsed]
google_compute_instance.cluster: Creation complete after 12s [id=cluster]
null_resource.cluster: Creating…
null_resource.cluster: Provisioning with 'file'…
null_resource.cluster: Provisioning with 'remote-exec'…
null_resource.cluster (remote-exec): Connecting to remote host via SSH…
null_resource.cluster (remote-exec): Host: 35.228.82.222
null_resource.cluster (remote-exec): User: essh
null_resource.cluster (remote-exec): Password: false
null_resource.cluster (remote-exec): Private key: true
null_resource.cluster (remote-exec): Certificate: false
null_resource.cluster (remote-exec): SSH Agent: false
null_resource.cluster (remote-exec): Checking Host Key: false
null_resource.cluster (remote-exec): Connected!
null_resource.cluster: Creation complete after 7s [id=816586071607403364]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

esschtolts@cluster:~$ ls /home/essh/
client.js test.txt

[sudo] password for essh:
google_compute_address.static-ip-address: Refreshing state… [id=node-cluster-243923/europe-north1/static-ip-address]
google_compute_instance.cluster: Refreshing state… [id=cluster]
null_resource.cluster: Refreshing state… [id=816586071607403364]
```

Enter a value: yes

null_resource.cluster: Destroying… [id=8165860716074033 64]
null_resource.cluster: Destruction complete after 0s
google_compute_instance.cluster: Destroying… [id=cluster]
google_compute_instance.cluster: Still destroying… [id=cluster, 10s elapsed]
google_compute_instance.cluster: Still destroying… [id=cluster, 20s elapsed]
google_compute_instance.cluster: Destruction complete after 27s
google_compute_address.static-ip-address: Destroying… [id=node-cluster-243923/europe-north1/static-ip-address]
google_compute_address.static-ip-address: Destruction complete after 8s

Для деплоя всего проекта можно добавить его в репозиторий, а загружать его в виртуальную машину будем через копирование установочного скрипта на эту виртуальную машину с последующим его запуском:

Переходим к Kubernetes

В минимальном варианте создание кластера из трёх нод выглядит примерно так:

```
essh@kubernetes-master:~/node-cluster/Kubernetes$ cat main.tf
provider "google" {
credentials = "${file("../kubernetes_key.json")}"
project = "node-cluster-243923"
region = "europe-north1"
}

resource "google_container_cluster" "node-ks" {
name = "node-ks"
location = "europe-north1-a"
initial_node_count = 3
}
essh@kubernetes-master:~/node-cluster/Kubernetes$ sudo ../terraform init
essh@kubernetes-master:~/node-cluster/Kubernetes$ sudo ../terraform apply
```

Кластер создался за 2:15, а после того, как я добавил europe-north1-a две дополнительные зоны europe-north1 -b, europe-north1-c и установили количество создаваемых инстансев в зоне в один, кластер создался за 3:13 секунде, потому что для более высокой доступности ноды были созданы в разных дата-центрах: europe-north1-a, europe-north1-b, europe-north1-c:

```
provider "google" {
credentials = "${file("../kubernetes_key.json")}"
project = "node-cluster-243923"
region = "europe-north1"
}

resource "google_container_cluster" "node-ks" {
name = "node-ks"
location = "europe-north1-a"
node_locations = ["europe-north1-b", "europe-north1-c"]
initial_node_count = 1
}
```

Теперь разделим наш кластер на два: управляющий кластер с Kubernetes и кластер для наших POD. Все кластера будет распределены по трём дата – центрам. Кластер для наших POD может авто масштабироваться под нагрузкой до 2 на каждой зоне (с трёх до шести в общем):

```
essh@kubernetes-master:~/node-cluster/Kubernetes$ cat main.tf
provider "google" {
credentials = "${file("../kubernetes_key.json")}"
project = "node-cluster-243923"
region = "europe-north1"
}

resource "google_container_cluster" "node-ks" {
name = "node-ks"
location = "europe-north1-a"
node_locations = ["europe-north1-b", "europe-north1-c"]
initial_node_count = 1
}

resource "google_container_node_pool" "node-ks-pool" {
name = "node-ks-pool"
cluster = "${google_container_cluster.node-ks.name}"
location = "europe-north1-a"
node_count = "1"

node_config {
machine_type = "n1-standard-1"
}

autoscaling {
min_node_count = 1
max_node_count = 2
}
}
```

Посмотрим, что получалось и поищем IP- адрес точки входа в кластер:

```
essh@kubernetes-master:~/node-cluster/Kubernetes$ gcloud container clusters list
NAME LOCATION MASTER_VERSION MASTER_IP MACHINE_TYPE NODE_VERSION NUM_NODES STATUS
node-ks europe-north1-a 1.12.8-gke.6 35.228.20.35 n1-standard-1 1.12.8-gke.6 6 RECONCILING

essh@kubernetes-master:~/node-cluster/Kubernetes$ gcloud container clusters describe node-ks | grep '^endpoint'
endpoint: 35.228.20.35

essh@kubernetes-master:~/node-cluster/Kubernetes$ ping 35.228.20.35 -c 2
PING 35.228.20.35 (35.228.20.35) 56(84) bytes of data.
64 bytes from 35.228.20.35: icmp_seq=1 ttl=59 time=8.33 ms
64 bytes from 35.228.20.35: icmp_seq=2 ttl=59 time=7.09 ms

-– 35.228.20.35 ping statistics –
```

2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 7.094/7.714/8.334/0.620 ms

Добавив переменные, которые я выделил в отдельный файл только для наглядности, которые параметризуют наш конфиг под разное использование, мы сможем его использовать, например, для создания тестового и рабочего кластеров. Добавлять переменные можно как var.name_value, а вставлять в текст аналогично JS: ${var.name_value}, а также path.root.

```
essh@kubernetes-master:~/node-cluster/Kubernetes$ cat variables.tf
variable "region" {
default = "europe-north1"
}

variable "project_name" {
type = string
default = ""
}

variable "gce_key" {
default = "./kubernetes_key.json"
}

variable "node_count_zone" {
default = 1
}
```

Передать их можно через ключ -var, например: sudo ./terraform apply -var="project_name=node-cluster-243923".

```
essh@kubernetes-master:~/node-cluster/Kubernetes$ cp ../kubernetes_key.json .
essh@kubernetes-master:~/node-cluster/Kubernetes$ sudo ../terraform apply -var="project_name=node-cluster-243923"
```

Наш проект в папке является не только проектом, но и модулем, готовым к использованию:

```
essh@kubernetes-master:~/node-cluster/Kubernetes$ cd ..
essh@kubernetes-master:~/node-cluster$ cat main.tf
module "Kubernetes" {
source = "./Kubernetes"
project_name = "node-cluster-243923"
}
essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply
```

Или же загрузить в публичный репозиторий:

```
essh@kubernetes-master:~/node-cluster/Kubernetes$ git init
Initialized empty GIT repository in /home/essh/node-cluster/Kubernetes/.git/
essh@kubernetes-master:~/node-cluster/Kubernetes$ echo "terraform.tfstate" >> .gitignore
essh@kubernetes-master:~/node-cluster/Kubernetes$ echo "terraform.tfstate.backup" >> .gitignore
essh@kubernetes-master:~/node-cluster/Kubernetes$ echo ".terraform/" >> .gitignore
essh@kubernetes-master:~/node-cluster/Kubernetes$ rm -f kubernetes_key.json
essh@kubernetes-master:~/node-cluster/Kubernetes$ git remote add origin https://github.com/ESSch/terraform-google-kubernetes.git
essh@kubernetes-master:~/node-cluster/Kubernetes$ git add .
```

essh@kubernetes-master:~/node-cluster/Kubernetes$ git commit -m 'create a k8s Terraform module'

[master (root-commit) 4f73c64] create a Kubernetes Terraform module

3 files changed, 48 insertions(+)

create mode 100644 .gitignore

create mode 100644 main.tf

create mode 100644 variables.tf

essh@kubernetes-master:~/node-cluster/Kubernetes$ git push -u origin master

essh@kubernetes-master:~/node-cluster/Kubernetes$ git tag -a v0.0.2 -m 'publish'

essh@kubernetes-master:~/node-cluster/Kubernetes$ git push origin v0.0.2

После публикации в реестре модулей https://registry.terraform.io/, выполнив требования, такие как наличие описания, мы можем его использовать:

essh@kubernetes-master:~/node-cluster$ cat main.tf

module "kubernetes" {

source = "./Kubernetes"

source = "ESSch/kubernetes/google"

version = "0.0.2"

project_name = "node-cluster-243923"

}

essh@kubernetes-master:~/node-cluster$ sudo ./terraform init

essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply

При очередном создании кластера у меня возникла ошибка ZONE_RESOURCE_POOL_EXHAUSTED "does not have enough resources available to fulfill the request. Try a different zone, or try again later", говорящая о том, что в данном регионе нет нужных серверов. Для меня это не проблема и мне не нужно править код модуля, ведь я параметризовал модуль регионом, и если я просто передам модулю в качестве параметра регионregion = "europe-west2", terraform после команды обновления и инициализации ./terrafrom init и команды применения ./terraform apply перенесёт мой кластер в указный регион. Ещё немного улучшим наш модуль тем, что вынесем провайдер (provider) из дочернего модуля Kubernetes в главный модуль (основной скрипт также является модулем). Вынеся в главный модуль, мы сможем использовать ещё один модуль, в противном случае провайдер в одном модуле будет конфликтовать с провайдером в другом. Наследование из главного модуля в дочерние и их прозрачность относится только к провайдерам. Остальные же данные для передачи из дочернего в родительский придётся использовать внешние переменные, а из родительского в дочерний – параметризовать родительский, но это позже, когда мы будем создавать ещё один модель. Также, вынесение провайдера в родительский модуль, будет полезно при создании следующего модуля, который мы создадим, так как он будет создавать элементы Kubernetes, которые не зависят от провайдера, и мы сможем тем самым отвязать провайдера Google от нашего модуля и его можно будет использовать с другими провайдерами, поддерживающим Kubernetes. Теперь нам не нужна передавать в переменной название проекта – он задан в провайдере. Для удобства разработки я буду пока использовать локальное подключение модуля. Я создал папку и файл нового модуля:

essh@kubernetes-master:~/node-cluster$ ls nodejs/

main.tf

essh@kubernetes-master:~/node-cluster$ cat main.tf

//module "kubernetes" {

// source = "ESSch/kubernetes/google"

```
// version = "0.0.2"
//
// project_name = "node-cluster-243923"
// region = "europe-west2"
//}

provider "google" {
credentials = "${file("./kubernetes_key.json")}"
project = "node-cluster-243923"
region = "europe-west2"
}

module "Kubernetes" {
source = "./Kubernetes"
project_name = "node-cluster-243923"
region = "europe-west2"
}

module "nodejs" {
source = "./nodejs"
}
essh@kubernetes-master:~/node-cluster$ sudo ./terraform init
essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply
```

Теперь передадим данные из инфраструктурного модуля Kubernetes в модуль приложений:

```
essh@kubernetes-master:~/node-cluster$ cat Kubernetes/outputs.tf
output "endpoint" {
value = google_container_cluster.node-ks.endpoint
sensitive = true
}

output "name" {
value = google_container_cluster.node-ks.name
sensitive = true
}

output "cluster_ca_certificate" {
value = base64decode(google_container_cluster.node-ks.master_auth.0.cluster_ca_certificate)
}
essh@kubernetes-master:~/node-cluster$ cat main.tf
//module "kubernetes" {
// source = "ESSch/kubernetes/google"
// version = "0.0.2"
//
// project_name = "node-cluster-243923"
// region = "europe-west2"
//}
```

```
provider "google" {
credentials = file("./kubernetes_key.json")
project = "node-cluster-243923"
region = "europe-west2"
}

module "Kubernetes" {
source = "./Kubernetes"
project_name = "node-cluster-243923"
region = "europe-west2"
}

module "nodejs" {
source = "./nodejs"
endpoint = module.Kubernetes.endpoint
cluster_ca_certificate = module.Kubernetes.cluster_ca_certificate
}
essh@kubernetes-master:~/node-cluster$ cat nodejs/variable.tf
variable "endpoint" {}

variable "cluster_ca_certificate" {}
```

Для проверки балансировки трафика со всех нод запустим NGINX, заменив стандартную страницу на имя хоста. Заменим мы простым вызовом команды и возобновим работу сервера. Для запуска сервера посмотрим в Dockerfile его вызов: CMD ["Nginx", "-g", "daemon off;"], что равносильно вызову Nginx -g 'daemon off;' в командной строке . Как видим, в Dockerfile не используется BASH в качестве среды для запуска, а запускается непосредственно сам сервер, что позволяет в случае в случае падения процесса оболочка будет жить, не давая контейнеру упасть и создаться заново. Но для наших экспериментов вполне подойдёт BASH:

```
essh@kubernetes-master:~/node-cluster$ sudo docker run -it Nginx:1.17.0 which Nginx
/usr/sbin/Nginx
sudo docker run -it –rm -p 8333:80 Nginx:1.17.0 /bin/bash -c "echo \$HOSTNAME > /usr/share/Nginx/html/index2.html && /usr/sbin/Nginx -g 'daemon off;'"
```

Теперь создадим наши POD в трёх экземплярах с NGINX, которые Kubernetes будут стараться распределить на разные сервера по умолчанию. Также добавим сервис в качестве балансировщика:

```
essh@kubernetes-master:~/node-cluster$ cat nodejs/main.tf
terraform {
required_version = ">= 0.12.0"
}

data "google_client_config" "default" {}

provider "kubernetes" {
host = var.endpoint

token = data.google_client_config.default.access_token
cluster_ca_certificate = var.cluster_ca_certificate

load_config_file = false
```

```
}

essh@kubernetes-master:~/node-cluster$ cat nodejs/main.tf
resource "kubernetes_deployment" "nodejs" {
metadata {
name = "terraform-nodejs"
labels = {
app = "NodeJS"
}
}
spec {
replicas = 3
selector {
match_labels = {
app = "NodeJS"
}
}
template {
metadata {
labels = {
app = "NodeJS"
}
}
spec {
container {
image = "Nginx:1.17.0"
name = "node-js"
command = ["/bin/bash"]
args = ["-c", "echo $HOSTNAME > /usr/share/Nginx/html/index.html && /usr/sbin/Nginx -
g 'daemon off;'"]
}
}
}
}
}

resource "kubernetes_service" "nodejs" {
metadata {
name = "terraform-nodejs"
}
spec {
selector = {
app = kubernetes_deployment.nodejs.metadata.0.labels.app
}
port {
port = 80
target_port = var.target_port
}
```

type = "LoadBalancer"
}
Проверим работу с помощью kubectl, для этого передадим секреты от gcloud к kubectl.
essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply

essh@kubernetes-master:~/node-cluster$ gcloud container clusters get-credentials node-ks –region=europe-west2-a
Fetching cluster endpoint and auth data.
kubeconfig entry generated for node-ks.

essh@kubernetes-master:~/node-cluster$ kubectl get deployments -o wide
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR
terraform-nodejs 3 3 3 3 25m node-js Nginx:1.17.0 app=NodeJS

essh@kubernetes-master:~/node-cluster$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE
terraform-nodejs-6bd565dc6c-8768b 1/1 Running 0 4m45s 10.4.3.15 gke-node-ks-node-ks-pool-07115c5b-bw15 none>
terraform-nodejs-6bd565dc6c-hr5vg 1/1 Running 0 4m42s 10.4.5.13 gke-node-ks-node-ks-pool-27e2e52c-9q5b none>
terraform-nodejs-6bd565dc6c-mm7lh 1/1 Running 0 4m43s 10.4.2.6 gke-node-ks-default-pool-2dc50760-757p none>

esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ docker ps | grep node-js_terraform
152e3c0ed940 719cd2e3ed04
"/bin/bash -c 'ech…" 8 minutes ago Up 8 minutes
Kubernetes_node-js_terraform-nodejs-6bd565dc6c-8768b_default_7a87ae4a-9379-11e9-a78e-42010a9a0114_0

esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ docker exec -it 152e3c0ed940 cat /usr/share/Nginx/html/index.html
terraform-nodejs-6bd565dc6c-8768b

esschtolts@gke-node-ks-node-ks-pool-27e2e52c-9q5b ~ $ docker exec -it c282135be446 cat /usr/share/Nginx/html/index.html
terraform-nodejs-6bd565dc6c-hr5vg

esschtolts@gke-node-ks-default-pool-2dc50760-757p ~ $ docker exec -it 8d1cf9ef44e6 cat /usr/share/Nginx/html/index.html
terraform-nodejs-6bd565dc6c-mm7lh

esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.4.2.6
terraform-nodejs-6bd565dc6c-mm7lh
esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.4.5.13
terraform-nodejs-6bd565dc6c-hr5vg
esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.4.3.15
terraform-nodejs-6bd565dc6c-8768b

Балансировщик балансируют нагрузку между POD, которые отфильтрованы по соответствию их селекторов в метаинформации и Селектора, указанного в описании балансировщика в секции spec. Все ноды связанны в одну общую сеть, поэтому можно подключиться к любой ноде (я это сделал через SSH WEB- интерфейса GCP в разделе с виртуальными машинами Compute Engine). Обратиться можно как по IP-адресу в контейнере или хосте ноды, так и по хосту сервиса terraform-nodejs в контейнере curl terraform-NodeJS:80, который создаётся внутренним DNS по названию сервиса. Посмотреть внешний IP-адрес EXTERNAL -IP можно как с помощью kubectl у сервиса, так и с помощью веб-интерфейса: GCP –> Kubernetes Engine –> Сервисы:

essh@kubernetes-master:~/node-cluster$ kubectl get service
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.7.240.1 none> 443/TCP 6h58m
terraform-nodejs LoadBalancer 10.7.246.234 35.197.220.103 80:32085/TCP 5m27s

esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.7.246.234
terraform-nodejs-6bd565dc6c-mm7lh
esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.7.246.234
terraform-nodejs-6bd565dc6c-mm7lh
esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.7.246.234
terraform-nodejs-6bd565dc6c-hr5vg
esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.7.246.234
terraform-nodejs-6bd565dc6c-hr5vg
esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.7.246.234
terraform-nodejs-6bd565dc6c-8768b
esschtolts@gke-node-ks-node-ks-pool-07115c5b-bw15 ~ $ curl 10.7.246.234
terraform-nodejs-6bd565dc6c-mm7lh

essh@kubernetes-master:~/node-cluster$ curl 35.197.220.103
terraform-nodejs-6bd565dc6c-mm7lh
essh@kubernetes-master:~/node-cluster$ curl 35.197.220.103
terraform-nodejs-6bd565dc6c-mm7lh
essh@kubernetes-master:~/node-cluster$ curl 35.197.220.103
terraform-nodejs-6bd565dc6c-8768b
essh@kubernetes-master:~/node-cluster$ curl 35.197.220.103
terraform-nodejs-6bd565dc6c-hr5vg
essh@kubernetes-master:~/node-cluster$ curl 35.197.220.103
terraform-nodejs-6bd565dc6c-8768b
essh@kubernetes-master:~/node-cluster$ curl 35.197.220.103
terraform-nodejs-6bd565dc6c-mm7lh

Теперь перейдем к внедрению сервера NodeJS:

essh@kubernetes-master:~/node-cluster$ sudo ./terraform destroy
essh@kubernetes-master:~/node-cluster$ sudo ./terraform apply
essh@kubernetes-master:~/node-cluster$ sudo docker run -it –rm node:12 which node
/usr/local/bin/node

sudo docker run -it –rm -p 8222:80 node:12 /bin/bash -c 'cd /usr/src/ && git clone https://github.com/fhinkel/nodejs-hello-world.git &&
/usr/local/bin/node /usr/src/nodejs-hello-world/index.js'
firefox http://localhost:8222

Заменим блок у нашего контейнера на:

container {
image = "node:12"
name = "node-js"
command = ["/bin/bash"]
args = [
"-c",
"cd /usr/src/ && git clone https://github.com/fhinkel/nodejs-hello-world.git && /usr/local/bin/node /usr/src/nodejs-hello-world/index.js"
]
}

Если закомментировать модуль Kubernetes, а в кэше он остаётся, остаётся убрать из кэша лишнее:

essh@kubernetes-master:~/node-cluster$ ./terraform apply

Error: Provider configuration not present

essh@kubernetes-master:~/node-cluster$ ./terraform state list
data.google_client_config.default
module.Kubernetes.google_container_cluster.node-ks
module.Kubernetes.google_container_node_pool.node-ks-pool
module.nodejs.kubernetes_deployment.nodejs
module.nodejs.kubernetes_service.nodejs

essh@kubernetes-master:~/node-cluster$ ./terraform state rm module.nodejs.kubernetes_deployment.nodejs
Removed module.nodejs.kubernetes_deployment.nodejs
Successfully removed 1 resource instance(s).
essh@kubernetes-master:~/node-cluster$ ./terraform state rm module.nodejs.kubernetes_service.nodejs
Removed module.nodejs.kubernetes_service.nodejs
Successfully removed 1 resource instance(s).

essh@kubernetes-master:~/node-cluster$ ./terraform apply
module.Kubernetes.google_container_cluster.node-ks: Refreshing state… [id=node-ks]
module.Kubernetes.google_container_node_pool.node-ks-pool: Refreshing state… [id=europe-west2-a/node-ks/node-ks-pool]

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Надежность и автоматизация кластера на Terraform

В общих черта автоматизация рассмотрена в https://codelabs.developers.google.com/codelabs/cloud-builder-gke-continuous-deploy/index. html #0. Мы остановимся подробнее. Теперь, если мы запустим удаление ./terraform destroy и попытаемся воссоздать инфраструктуру целиком сначала, то у нас появятся ошибки. Ошибки получаются из-за того, что порядок создания сервисов не задан и terraform, по умолчанию, отравляет запросы к API в 10 параллельных потоков, хотя это можно изменить, указав при применении или удаление ключ -parallelism=1. В результате Terraform пытается создать Kubernetes сервисы (Deployment и service) на серверах (node-pull), которых пока ещё нет, та же ситуация и при создании сервиса, которому требуется , проксирующего ещё не созданный Deployment.

Указывая Terraform запрашивать API в один поток ./terraform apply -parallelism=1 мы снижаем возможные ограничения со стороны провайдера на частоту обращений к API, но не решаем проблему отсутствия порядка создания сущностей. Мы не будем комментировать зависимые блоки и постепенно раскомментировать и запускать ./terraform apply, также не будем запускать по частям нашу систему, указывая конкретные блоки ./terraform apply -target=module.nodejs.kubernetes_deployment.nodejs. Мы укажем в коде сами зависимости на инициализации переменной, первая из которой уже определена как внешняя var.endpoint, а вторую мы создадим локально:

```
locals {
app = kubernetes_deployment.nodejs.metadata.0.labels.app
}
```

Теперь мы можем добавить зависимости в код depends_on = [var.endpoint] и depends_on = [kubernetes_deployment .nodejs].

Таже может появляться ошибка недоступности сервиса: Error: Get https://35.197.228.3/API/v1…: dial tcp 35.197.228.3:443: connect: connection refused, то скорее превышено время подключения, которое составляет по умолчанию 6 минут (3600 секунд), но тут можно просто попробовать еще раз.

Теперь перейдём к решению проблемы надёжности работы контейнера, основной процесс которого мы запускаем в командной оболочке. Первое, что мы сделаем, отделим создание приложения от запуска контейнера. Для этого нужно весь процесс создания сервиса перенести в процесс создания образа, которых можно протестировать, и по которому можно создать контейнер сервиса. Итак, создадим образ:

```
essh@kubernetes-master:~/node-cluster$ cat app/server.js
const http = require('http');
const server = http.createServer(function(request, response) {
response.writeHead(200, {"Content-Type": "text/plain"});
response.end(`Nodejs_cluster is working! My host is ${process.env.HOSTNAME}`);
});

server.listen(80);

essh@kubernetes-master:~/node-cluster$ cat Dockerfile
FROM node:12
WORKDIR /usr/src/
ADD ./app /usr/src/

RUN npm install

EXPOSE 3000
ENTRYPOINT ["node", "server.js"]

essh@kubernetes-master:~/node-cluster$ sudo docker image build -t nodejs_cluster .
Sending build context to Docker daemon 257.4MB
Step 1/6 : FROM node:12
—> b074182f4154
Step 2/6 : WORKDIR /usr/src/
—> Using cache
—> 06666b54afba
Step 3/6 : ADD ./app /usr/src/
```

—> Using cache
—> 13fa01953b4a
Step 4/6 : RUN npm install
—> Using cache
—> dd074632659c
Step 5/6 : EXPOSE 3000
—> Using cache
—> ba3b7745b8e3
Step 6/6 : ENTRYPOINT ["node", "server.js"]
—> Using cache
—> a957fa7a1efa
Successfully built a957fa7a1efa
Successfully tagged nodejs_cluster:latest

essh@kubernetes-master:~/node-cluster$ sudo docker images | grep nodejs_cluster
nodejs_cluster latest a957fa7a1efa 26 minutes ago 906MB

Теперь положим наш образ в реестр GCP, а не Docker Hub, потому что мы получаем сразу приватный репозиторий с которому автоматом есть доступ у наших сервисов:

essh@kubernetes-master:~/node-cluster$ IMAGE_ID="nodejs_cluster"
essh@kubernetes-master:~/node-cluster$ sudo docker tag $IMAGE_ID:latest gcr.io/$PROJECT_ID/$IMAGE_ID:latest
essh@kubernetes-master:~/node-cluster$ sudo docker images | grep nodejs_cluster
nodejs_cluster latest a957fa7a1efa 26 minutes ago 906MB
gcr.io/node-cluster-243923/nodejs_cluster latest a957fa7a1efa 26 minutes ago 906MB

essh@kubernetes-master:~/node-cluster$ gcloud auth configure-docker
gcloud credential helpers already registered correctly.
essh@kubernetes-master:~/node-cluster$ docker push gcr.io/$PROJECT_ID/$IMAGE_ID:latest
The push refers to repository [gcr.io/node-cluster-243923/nodejs_cluster]
194f3d074f36: Pushed
b91e71cc9778: Pushed
640fdb25c9d7: Layer already exists
b0b300677afe: Layer already exists
5667af297e60: Layer already exists
84d0c4b192e8: Layer already exists
a637c551a0da: Layer already exists
2c8d31157b81: Layer already exists
7b76d801397d: Layer already exists
f32868cde90b: Layer already exists
0db06dff9d9a: Layer already exists
latest: digest: sha256:912938003a93c53b7c8f806cded3f9bffae7b5553b9350c75791ff7acd1dad0b size: 2629

essh@kubernetes-master:~/node-cluster$ gcloud container images list
NAME
gcr.io/node-cluster-243923/nodejs_cluster

Only listing images in gcr.io/node-cluster-243923. Use –repository to list images in other repositories.

Теперь мы можем посмотреть его в админке GCP: Container Registry –> Образы. Заменим код нашего контейнера на код с нашим образом. Если для продакшна нужно обязательно версионировать запускаемый образ во избежание автоматического их обновления при системных пересозданиях POD, например, при переносе POD с одной ноды на другую при выводе машины с нашей нодой на техническое обслуживание. Для разработки лучше использовать тег latest, который позволит при обновлении образа обновить сервис. При обновлении сервиса его нужно пересоздать, то есть удалить и заново создать, так как иначе terraform просто обновит параметры, а не пересоздаст контейнер с новым образом. Также, если обновим образ и пометим сервис как изменённый с помощью команды ./terraform taint ${NAME_SERVICE}, наш сервис будет просто обновлён, что можно увидеть с помощью команды ./terraform plan. Поэтому для обновления, пока, нужно пользоваться командами ./terraform destroy -target= ${NAME_SERVICE} и ./terraform apply, а название сервисов можно посмотреть в ./terraform state list:

essh@kubernetes-master:~/node-cluster$ ./terraform state list
data.google_client_config.default
module.kubernetes.google_container_cluster.node-ks
module.kubernetes.google_container_node_pool.node-ks-pool
module.Nginx.kubernetes_deployment.nodejs
module.Nginx.kubernetes_service.nodejs

essh@kubernetes-master:~/node-cluster$ ./terraform destroy -target=module.nodejs.kubernetes_deployment.nodejs

essh@kubernetes-master:~/node-cluster$ ./terraform apply
А теперь заменим код нашего контейнера:
container {
image = "gcr.io/node-cluster-243923/nodejs_cluster:latest"
name = "node-js"
}
Проверим результат балансировки на разные ноды(нет переноса строк в конце вывода):
essh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lqg48essh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lqg48essh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80
Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80

Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80

Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lhgsnessh@kubernetes-master:~/node-cluster$ curl http://35.246.85.138:80

Nodejs_cluster is working! My host is terraform-nodejs-997fd5c9c-lqg48essh@kubernetes-master:~/node-cluster$

Автоматизируем процесс создания образов, для этого воспользуемся сервисом Google Cloud Build (бесплатен для 5 пользователей и трафике до 50Гб) для создания нового образа при создании в репозитории Cloud Source Repositories (бесплатно на Google Cloud Platform Free Tier) новой версии (тэга). Google Cloud Platform –> Menu –> Инструменты –> Cloud Build –> триггеры –> Включить Cloud Build API –> Начать –> Создать хранилище, которое будет доступно по Google Cloud Platform –> Menu –> Инструменты —> Хранилища исходного кода (Cloud Source Repositories):

essh@kubernetes-master:~/node-cluster$ cd app/

essh@kubernetes-master:~/node-cluster/app$ ls

server.js

essh@kubernetes-master:~/node-cluster/app$ mv ./server.js ../

essh@kubernetes-master:~/node-cluster/app$ gcloud source repos clone nodejs –project=node-cluster-243923

Cloning into '/home/essh/node-cluster/app/nodejs'…

warning: You appear to have cloned an empty repository.

Project [node-cluster-243923] repository [nodejs] was cloned to [/home/essh/node-cluster/app/nodejs].

essh@kubernetes-master:~/node-cluster/app$ ls -a

. .. nodejs

essh@kubernetes-master:~/node-cluster/app$ ls nodejs/

essh@kubernetes-master:~/node-cluster/app$ ls -a nodejs/

. .. .git

essh@kubernetes-master:~/node-cluster/app$ cd nodejs/

essh@kubernetes-master:~/node-cluster/app/nodejs$ mv ../../server.js .

essh@kubernetes-master:~/node-cluster/app/nodejs$ git add server.js

essh@kubernetes-master:~/node-cluster/app/nodejs$ git commit -m 'test server'

[master (root-commit) 46dd957] test server

1 file changed, 7 insertions(+)

create mode 100644 server.js

essh@kubernetes-master:~/node-cluster/app/nodejs$ git push -u origin master

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 408 bytes | 408.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://source.developers.google.com/p/node-cluster-243923/r/nodejs

* [new branch] master –> master

Branch 'master' set up to track remote branch 'master' from 'origin'.

Теперь пора настроить создание образа при создании новой версии продукта: переходим в GCP –> Cloud Build –> триггеры –> Создать триггер –> Хранилище исходного кода Google Cloud –> NodeJS. Тип триггер тег, чтобы не создавался образ при обычных коммитах. Я поменяю название образа с gcr.io/node-cluster-243923/NodeJS:$SHORT_SHA на gcr.io/node-

cluster-243923/NodeJS:$SHORT_SHA, а время ожидания на 60 секунд. Теперь сделаю коммит и добавлю тег:

```
essh@kubernetes-master:~/node-cluster/app/nodejs$ cp ../../Dockerfile .
essh@kubernetes-master:~/node-cluster/app/nodejs$ git add Dockerfile
essh@kubernetes-master:~/node-cluster/app/nodejs$ git add Dockerfile
essh@kubernetes-master:~/node-cluster/app/nodejs$ cp ../../Dockerfile .
essh@kubernetes-master:~/node-cluster/app/nodejs$ git add Dockerfile
essh@kubernetes-master:~/node-cluster/app/nodejs$ git commit -m 'add Dockerfile'
essh@kubernetes-master:~/node-cluster/app/nodejs$ git remote -v
origin https://source.developers.google.com/p/node-cluster-243923/r/nodejs (fetch)
origin https://source.developers.google.com/p/node-cluster-243923/r/nodejs (push)
essh@kubernetes-master:~/node-cluster/app/nodejs$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 380 bytes | 380.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://source.developers.google.com/p/node-cluster-243923/r/nodejs
46dd957..b86c01d master –> master
essh@kubernetes-master:~/node-cluster/app/nodejs$ git tag
essh@kubernetes-master:~/node-cluster/app/nodejs$ git tag -a v0.0.1 -m 'test to run'
essh@kubernetes-master:~/node-cluster/app/nodejs$ git push origin v0.0.1
Counting objects: 1, done.
Writing objects: 100% (1/1), 161 bytes | 161.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://source.developers.google.com/p/node-cluster-243923/r/nodejs
* [new tag] v0.0.1 –> v0.0.1
```

Теперь, если мы нажмём кнопку запустить триггер, мы увидим образ в Container Registry с нашим тегом:

```
essh@kubernetes-master:~/node-cluster/app/nodejs$ gcloud container images list
NAME
gcr.io/node-cluster-243923/nodejs
gcr.io/node-cluster-243923/nodejs_cluster
```

Only listing images in gcr.io/node-cluster-243923. Use –repository to list images in other repositories.

Теперь если мы просто добавим изменения и тег, то образ будет создан автоматически:

```
essh@kubernetes-master:~/node-cluster/app/nodejs$ sed -i 's/HOSTNAME\}/HOSTNAME \}\n/' server.js
essh@kubernetes-master:~/node-cluster/app/nodejs$ git add server.js
essh@kubernetes-master:~/node-cluster/app/nodejs$ git commit -m 'fix'
[master 230d67e] fix
1 file changed, 2 insertions(+), 1 deletion(-)
essh@kubernetes-master:~/node-cluster/app/nodejs$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 304 bytes | 304.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1)
```

```
To https://source.developers.google.com/p/node-cluster-243923/r/nodejs
b86c01d..230d67e master –> master
essh@kubernetes-master:~/node-cluster/app/nodejs$ git tag -a v0.0.2 -m 'fix'
essh@kubernetes-master:~/node-cluster/app/nodejs$ git push origin v0.0.2
Counting objects: 1, done.
Writing objects: 100% (1/1), 158 bytes | 158.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://source.developers.google.com/p/node-cluster-243923/r/nodejs
* [new tag] v0.0.2 –> v0.0.2
essh@kubernetes-master:~/node-cluster/app/nodejs$ sleep 60

essh@kubernetes-master:~/node-cluster/app/nodejs$ gcloud builds list
ID CREATE_TIME DURATION SOURCE IMAGES STATUS
2b024d7e-87a9-4d2a-980b-4e7c108c5fad 2019-06-22T17:13:14+00:00 28S nodejs@v0.0.2
gcr.io/node-cluster-243923/nodejs:v0.0.2 SUCCESS
6b4ae6ff-2f4a-481b-9f4e-219fafb5d572 2019-06-22T16:57:11+00:00 29S nodejs@v0.0.1
gcr.io/node-cluster-243923/nodejs:v0.0.1 SUCCESS
e50df082-31a4-463b-abb2-d0f72fbf62cb 2019-06-22T16:56:48+00:00 29S nodejs@v0.0.1
gcr.io/node-cluster-243923/nodejs:v0.0.1 SUCCESS
essh@kubernetes-master:~/node-cluster/app/nodejs$ git tag -a latest -m 'fix'
essh@kubernetes-master:~/node-cluster/app/nodejs$ git push origin latest
Counting objects: 1, done.
Writing objects: 100% (1/1), 156 bytes | 156.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://source.developers.google.com/p/node-cluster-243923/r/nodejs
* [new tag] latest –> latest
essh@kubernetes-master:~/node-cluster/app/nodejs$ cd ../..
```

Создание нескольких окружений с кластерами Terraform

При попытке создать несколько кластеров из одной конфигурации мы столкнёмся с дублированием идентификаторов, которые должны быть уникальны, поэтому изолируем их друг от друга, создав и поместив в разные проекты. Для ручного создания проекта перейдём по GCP –> Продукты –> IAM и администрирование –> Управление ресурсами и создадим проект NodeJS-prod и переключившись в проект, дождемся его активации. Посмотрим на состояние текущего проекта:

```
essh@kubernetes-master:~/node-cluster$ cat main.tf
provider "google" {
credentials = file("./kubernetes_key.json")
project = "node-cluster-243923"
region = "europe-west2"
}

module "kubernetes" {
source = "./Kubernetes"
}

data "google_client_config" "default" {}

module "Nginx" {
```

```
source = "./nodejs"
image = "gcr.io/node-cluster-243923/nodejs_cluster:latest"
endpoint = module.kubernetes.endpoint
access_token = data.google_client_config.default.access_token
cluster_ca_certificate = module.kubernetes.cluster_ca_certificate
}

essh@kubernetes-master:~/node-cluster$ gcloud config list project
[core]
project = node-cluster-243923

Your active configuration is: [default]

essh@kubernetes-master:~/node-cluster$ gcloud config set project node-cluster-243923
Updated property [core/project].

essh@kubernetes-master:~/node-cluster$ gcloud compute instances list
NAME ZONE INTERNAL_IP EXTERNAL_IP STATUS
gke-node-ks-default-pool-2e5073d4-csmg europe-north1-a 10.166.0.2 35.228.96.97
RUNNING
gke-node-ks-node-ks-pool-ccbaf5c6-4xgc europe-north1-a 10.166.15.233 35.228.82.222
RUNNING
gke-node-ks-default-pool-72a6d4a3-ldzg europe-north1-b 10.166.15.231 35.228.143.7
RUNNING
gke-node-ks-node-ks-pool-9ee6a401-ngfn europe-north1-b 10.166.15.234 35.228.129.224
RUNNING
gke-node-ks-default-pool-d370036c-kbg6 europe-north1-c 10.166.15.232 35.228.117.98
RUNNING
gke-node-ks-node-ks-pool-d7b09e63-q8r2 europe-north1-c 10.166.15.235 35.228.85.157
RUNNING
```

Переключим gcloud и посмотри на пустой проект:

```
essh@kubernetes-master:~/node-cluster$ gcloud config set project node-cluster-prod-244519
Updated property [core/project].
essh@kubernetes-master:~/node-cluster$ gcloud config list project
[core]
project = node-cluster-prod-244519

Your active configuration is: [default]
essh@kubernetes-master:~/node-cluster$ gcloud compute instances list
Listed 0 items.
```

В предыдущий раз, для node-cluster-243923 мы создавали сервисный аккаунт, от имени которого мы создавали кластер. Для работы с несколькими аккаунтами из Terraform создадим и для нового проекта сервисный аккаунт через IAM и администрирование –> Сервисные аккаунты. Нам нужно будет сделать две отдельные папки для запуска Terraform по отдельности для того, чтобы разделить SSH- подключения, имеющие разные авторизационные ключи. Если мы поместим оба провайдера с разными ключами, то получим успешное соединение для первого проекта, позже, когда Terraform перейдёт к созданию кластера для следующего проекта он получит отказ из-за недействительности ключа от первого проекта ко второму. Есть и другая возможность – активировать аккаунт, как аккаунт компании (потребуется сайт и электронная

почта, и проверка их со стороны Google), тогда появится возможность создавать из кода проекты без использования админки. После dev- окружение:

```
essh@kubernetes-master:~/node-cluster$ ./terraform destroy
essh@kubernetes-master:~/node-cluster$ mkdir dev
essh@kubernetes-master:~/node-cluster$ cd dev/
essh@kubernetes-master:~/node-cluster/dev$ gcloud config set project node-cluster-243923
Updated property [core/project].
essh@kubernetes-master:~/node-cluster/dev$ gcloud config list project
[core]
project = node-cluster-243923

Your active configuration is: [default]
essh@kubernetes-master:~/node-cluster/dev$ ../kubernetes_key.json ../main.tf .
essh@kubernetes-master:~/node-cluster/dev$ cat main.tf
provider "google" {
alias = "dev"
credentials = file("./kubernetes_key.json")
project = "node-cluster-243923"
region = "europe-west2"
}

module "kubernetes_dev" {
source = "../Kubernetes"
node_pull = false
providers = {
google = google.dev
}
}

data "google_client_config" "default" {}

module "Nginx" {
source = "../nodejs"
providers = {
google = google.dev
}
image = "gcr.io/node-cluster-243923/nodejs_cluster:latest"
endpoint = module.kubernetes_dev.endpoint
access_token = data.google_client_config.default.access_token
cluster_ca_certificate = module.kubernetes_dev.cluster_ca_certificate
}

essh@kubernetes-master:~/node-cluster/dev$ ../terraform init
essh@kubernetes-master:~/node-cluster/dev$ ../terraform apply
essh@kubernetes-master:~/node-cluster/dev$ gcloud compute instances list
NAME ZONE MACHINE_TYPE PREEMPTIBLE INTERNAL_IP EXTERNAL_IP STATUS
gke-node-ks-default-pool-71afadb8-4t39 europe-north1-a n1-standard-1 10.166.0.60 35.228.96.97 RUNNING
```

gke-node-ks-node-ks-pool-134dada1-3cdf europe-north1-a n1-standard-1 10.166.0.61 35.228.117.98 RUNNING

gke-node-ks-node-ks-pool-134dada1-c476 europe-north1-a n1-standard-1 10.166.15.194 35.228.82.222 RUNNING

essh@kubernetes-master:~/node-cluster/dev$ gcloud container clusters get-credentials node-ks
Fetching cluster endpoint and auth data.
kubeconfig entry generated for node-ks.

essh@kubernetes-master:~/node-cluster/dev$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE
terraform-nodejs-6fd8498cb5-29dzx 1/1 Running 0 2m57s 10.12.3.2 gke-node-ks-node-ks-pool-134dada1-c476 none>
terraform-nodejs-6fd8498cb5-jcbj6 0/1 Pending 0 2m58s none> none> none>
terraform-nodejs-6fd8498cb5-lvfjf 1/1 Running 0 2m58s 10.12.1.3 gke-node-ks-node-ks-pool-134dada1-3cdf none>

Как видно POD распределились по пулу нод, при этом не попали на ноду с Kubernetes из-за отсутствия свободного места. Важно заметить, что количество нод в пуле было увеличено автоматически, и только заданное ограничение не позволило создать третью ноду в пуле. Если мы установим remove_default_node_pool в true, то объединим POD Kubernetes и наши POD. По запросам ресурсов, Kubernetes занимает чуть более одного ядра, а наш POD половину, поэтому остальные POD не были созданы, но зато мы сэкономили на ресурсах:

essh@kubernetes-master:~/node-cluster/Kubernetes$ gcloud compute instances list
NAME ZONE MACHINE_TYPE PREEMPTIBLE INTERNAL_IP EXTERNAL_IP STATUS
gke-node-ks-node-ks-pool-495b75fa-08q2 europe-north1-a n1-standard-1 10.166.0.57 35.228.117.98 RUNNING
gke-node-ks-node-ks-pool-495b75fa-wsf5 europe-north1-a n1-standard-1 10.166.0.59 35.228.96.97 RUNNING

essh@kubernetes-master:~/node-cluster/Kubernetes$ gcloud container clusters get-credentials node-ks
Fetching cluster endpoint and auth data.
kubeconfig entry generated for node-ks.

essh@kubernetes-master:~/node-cluster/Kubernetes$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE
terraform-nodejs-6fd8498cb5-97svs 1/1 Running 0 14m 10.12.2.2 gke-node-ks-node-ks-pool-495b75fa-wsf5 none>
terraform-nodejs-6fd8498cb5-d9zkr 0/1 Pending 0 14m none> none> none>
terraform-nodejs-6fd8498cb5-phk8x 0/1 Pending 0 14m none> none> none>

После создания сервисного аккаунту, добавим ключ и проверим его:

essh@kubernetes-master:~/node-cluster/dev$ gcloud auth login essh@kubernetes-master:~/node-cluster/dev$ gcloud projects create node-cluster-prod3 Create in progress for [https://cloudresourcemanager.googleapis.com/v1/projects/node-cluster-prod3]. Waiting for [operations/cp.7153345484959140898] to finish…done. https://medium.com/@pnatraj/how-to-run-gcloud-command-line-using-a-service-account-f39043d515b9

essh@kubernetes-master:~/node-cluster$ gcloud auth application-default login

essh@kubernetes-master:~/node-cluster$ cp ~/Downloads/node-cluster-prod-244519-6fd863dd4d38.json ./kubernetes_prod.json
essh@kubernetes-master:~/node-cluster$ echo "kubernetes_prod.json" >> .gitignore
essh@kubernetes-master:~/node-cluster$ gcloud iam service-accounts list
NAME EMAIL DISABLED
Compute Engine default service account 1008874319751-compute@developer.gserviceaccount.com False
terraform-prod terraform-prod@node-cluster-prod-244519.iam.gserviceaccount.com False

essh@kubernetes-master:~/node-cluster$ gcloud projects list | grep node-cluster
node-cluster-243923 node-cluster 26345118671
node-cluster-prod-244519 node-cluster-prod 1008874319751
Создадим prod-окружение:
essh@kubernetes-master:~/node-cluster$ mkdir prod
essh@kubernetes-master:~/node-cluster$ cd prod/
essh@kubernetes-master:~/node-cluster/prod$ cp ../main.tf ../kubernetes_prod_key.json .
essh@kubernetes-master:~/node-cluster/prod$ gcloud config set project node-cluster-prod-244519
Updated property [core/project].

Конец ознакомительного фрагмента.