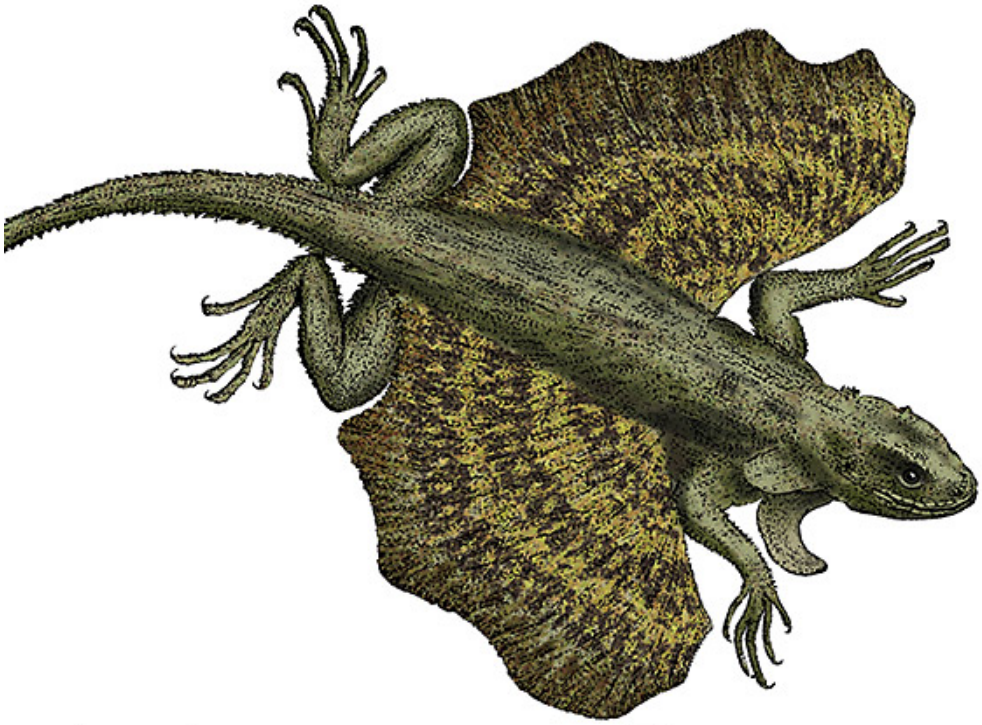
O'REILLY®

Второе Издание



Terraform

инфраструктура на уровне кода

Евгений Брикман

Евгений Брикман

Terraform: инфраструктура на уровне кода
2-е издание

2020
Научный редактор *К. Русецкий*
Переводчики *О. Сивченко, С. Черников*
Литературный редактор *В. Байдук*
Художник *В. Мостипан*
Корректоры *Е. Павлович, Е. Рафалюк-Бузовская*

Евгений Брикман
Terraform: инфраструктура на уровне кода. — СПб.: Питер, 2020.
ISBN 978-5-4461-1590-7
© ООО Издательство "Питер", 2020
Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Посвящается маме, папе, Лайле и Молли

Введение

Давным-давно в далеком-предалеком вычислительном центре древнее племя могущественных существ, известных как «сисадмины», вручную развертывало инфраструктуру. Каждый сервер, база данных (БД), балансировщик нагрузки и фрагмент сетевой конфигурации создавались и управлялись вручную. Это было мрачное и ужасное время: страх простоя, случайной ошибки в конфигурации, медленных и хрупких развертываний и того, что может произойти, если сисадмины перейдут на темную сторону (то есть возьмут отпуск). Но спешу вас обрадовать — благодаря движению DevOps у нас теперь есть замечательный инструмент: *Terraform*.

Terraform (https://www.terraform.io/) — это инструмент с открытым исходным кодом от компании HashiCorp. Он позволяет описывать инфраструктуру в виде кода на простом декларативном языке и развертывать ее/управлять ею в различных публичных облачных сервисах (скажем, Amazon Web Services, Microsoft Azure, Google Cloud Platform, DigitalOcean), а также частных облаках и платформах виртуализации (OpenStack, VMWare и др.) всего несколькими командами. Например, вместо того чтобы вручную щелкать кнопкой мыши на веб-странице или вводить десятки команд в консоль, вы можете воспользоваться следующим кодом и сконфигурировать сервер в AWS:

```
provider "aws" {
  region = "us-east-2"
}
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Чтобы его развернуть, введите следующее:

```
$ terraform init
$ terraform apply
```

Благодаря своей простоте и мощи Terraform стал ключевым игроком в мире DevOps. Он позволяет заменить громоздкие, хрупкие и неавтоматизированные средства управления инфраструктурой на надежный автоматизированный инструмент, поверх которого вы можете объединить все остальные элементы DevOps (автоматическое тестирование, непрерывную интеграцию и непрерывное развертывание) и сопутствующий инструментарий (например, Docker, Chef, Puppet).

Прочитайте эту книгу, и вы сможете сразу приступить к работе с Terraform.

Начав с простейшего примера Hello, World, вы научитесь работать с полным стеком технологий (кластером серверов, балансировщиком нагрузки, базой данных), рассчитанным на огромные объемы трафика и крупные команды разработчиков, уже после прочтения лишь нескольких глав. Это практическое руководство не только научит принципам DevOps и инфраструктуры как кода (infrastructure as code, или IaC), но и проведет вас через десятки примеров кода, которые можно попробовать выполнить дома. Поэтому держите компьютер под рукой.

Дочитав книгу, вы будете готовы к работе с Terraform в реальных условиях.

Целевая аудитория книги

Книга предназначена для всех, кто отвечает за уже написанный код. Это относится к сисадминам, специалистам по эксплуатации, релиз-, SR-, DevOps-инженерам, разработчикам инфраструктуры, разработчикам полного цикла, руководителям инженерной группы и техническим директорам. Какой бы ни была ваша должность, если вы занимаетесь инфраструктурой, развертываете код, конфигурируете серверы, масштабируете кластеры, выполняете резервное копирование данных, мониторите приложения и отвечаете на вызовы в три часа ночи, эта книга для вас.

В совокупности эти обязанности обычно называют операционной деятельностью (или системным администрированием). Раньше часто встречались разработчики, которые умели писать код, но не разбирались в системном администрировании; точно так же нередко попадались сисадмины без умения писать код. Когда-то такое разделение было приемлемым, но в современном мире, который уже нельзя представить без облачных вычислений и движения DevOps, практически любому разработчику необходимы навыки администрирования, а любой сисадмин должен уметь программировать.

Для чтения этой книги не обязательно быть специалистом в той или иной области — поверхностного знакомства с языками программирования, командной строкой и серверным программным обеспечением (сайтами) должно хватить. Всему остальному можно научиться в процессе. Таким образом, по окончании чтения вы будете уверенно разбираться в одном из важнейших аспектов современной разработки и системного администрирования — в управлении инфраструктурой как кодом.

Вы не только научитесь управлять инфраструктурой в виде кода, используя Terraform, но и узнаете, как это вписывается в общую концепцию DevOps. Вот несколько вопросов, на которые вы сможете ответить по прочтении этой книги.

• Зачем вообще использовать IaC?

• Какая разница между управлением конфигурацией, оркестрацией, инициализацией ресурсов и шаблонизацией серверов?

• Когда следует использовать Terraform, Chef, Ansible, Puppet, Salt, CloudFormation, Docker, Packer или Kubernetes?

• Как работает система Terraform и как с ее помощью управлять инфраструктурой?

• Как создавать модули Terraform, подходящие для повторного использования?

• Как писать код для Terraform, который будет достаточно надежным для практического применения?

• Как тестировать свой код для Terraform?

• Как внедрить Terraform в свой процесс автоматического развертывания?

• Как лучше всего использовать Terraform в командной работе?

Вам понадобятся лишь компьютер (Terraform поддерживает большинство операционных систем), интернет-соединение и желание учиться.

Почему я написал эту книгу

Terraform — мощный инструмент, совместимый со всеми популярными облачными провайдерами. Он основан на простом языке, позволяет повторно использовать код, выполнять тестирование и управлять версиями. Это открытый проект с дружелюбным и активным сообществом. Но, надо признать, он еще не до конца сформирован.

Terraform — относительно новая технология. Несмотря на ее популярность, по состоянию на январь 2020 года все еще не вышла версия 1.0.0 (стабильная версия — 0.12.21. — *Примеч. ред.*). По-прежнему сложно найти книги и статьи или встретить специалистов, которые бы помогли вам овладеть этим инструментом. Официальная документация Terraform хорошо подходит для знакомства с базовым синтаксисом и возможностями, но в ней мало информации об идиоматических шаблонах, рекомендуемых методиках, тестировании, повторном использовании кода и рабочих процессах в команде. Это как пытаться овладеть французским языком с помощью одного лишь словаря, игнорируя грамматику и идиомы.

Я написал эту книгу, чтобы помочь разработчикам изучить Terraform. Я пользуюсь этим инструментом четыре года из пяти с момента его создания — в основном в моей компании Gruntwork (http://www.gruntwork.io). Там он сыграл ключевую роль в создании библиотеки более чем из 300 000 строк проверенного временем инфраструктурного кода, готового к повторному использованию и уже применяемого сотнями компаний в промышленных условиях. Написание и поддержка такого большого объема инфраструктурного кода на таком длинном отрезке времени в таком огромном количестве разных компаний и сценариев применения позволили нам извлечь много непростых уроков. Я хочу поделиться ими с вами, чтобы вы могли сократить этот долгий процесс и овладеть Terraform в считаные дни.

Конечно, одним чтением этого не добьешься. Чтобы начать свободно разговаривать на французском, придется потратить какое-то время на общение с носителями языка, просмотр французских телепередач и прослушивание французской музыки. Чтобы овладеть Terraform, нужно написать для этой системы настоящий код, использовать его в реальном ПО и развернуть это ПО на настоящих серверах. Поэтому приготовьтесь к чтению, написанию и выполнению большого количества кода.

Структура издания

В книге освещается следующий список тем.

•*Глава 1 «Почему Terraform».* Как DevOps меняет наш подход к выполнению ПО; краткий обзор инструментов IaC, включая управление конфигурацией, шаблонизацию серверов, оркестрацию и инициализацию

ресурсов; преимущества IaC; сравнение Terraform, Chef, Puppet, Ansible, SaltStack, OpenStack Heat и CloudFormation; как сочетать такие инструменты, как Terraform, Packer, Docker, Ansible и Kubernetes.

• *Глава 2 «Приступаем к работе с Terraform».* Установка Terraform; краткий обзор синтаксиса Terraform; обзор утилиты командной строки Terraform; как развернуть один сервер; как развернуть веб-сервер; как развернуть кластер веб-серверов; как развернуть балансировщик нагрузки; как очистить созданные вами ресурсы.

• *Глава 3 «Как управлять состоянием Terraform».* Что такое состояние Terraform; как хранить состояние, чтобы к нему имели доступ разные члены команды; как блокировать файлы состояния, чтобы предотвратить конкуренцию; как управлять конфиденциальными данными в Terraform; как изолировать файлы состояния, чтобы смягчить последствия ошибок; как использовать рабочие области Terraform; рекомендуемая структура каталогов для проектов Terraform; как работать с состоянием, доступным только для чтения.

• *Глава 4 «Повторное использование инфраструктуры с помощью модулей Terraform».* Что такое модули; как создать простой модуль; как сделать модуль конфигурируемым с помощью входных и выходных значений; локальные переменные; версионные модули; потенциальные проблемы с модулями; использование модулей для описания настраиваемых элементов инфраструктуры с возможностью повторного применения.

• *Глава 5 «Работа с Terraform: циклы, условные выражения, развертывание и подводные камни».* Циклы с параметром count, выражения for_each и for, строковая директива for; условный оператор с параметром count, выражениями for_each и for, строковой директивой if; встроенные функции; развертывание с нулевым временем простоя; часто встречающиеся подводные камни, связанные с ограничениями count и for_each, развертываниями без простоя; как хорошие планы могут провалиться, проблемы с рефакторингом и отложенная согласованность.

• *Глава 6 «Код Terraform промышленного уровня».* Почему проекты DevOps всегда развертываются дольше, чем ожидается; что характеризует инфраструктуру, готовую к промышленному использованию; как создавать модули Terraform для промышленных условий; готовые к выпуску модули; реестр модулей Terraform; «аварийные люки» в Terraform.

• *Глава 7 «Как тестировать код Terraform».* Ручное тестирование кода Terraform; тестовые среды и очистка; автоматизированное тестирование кода Terraform; Terrarest; модульные тесты; интеграционные тесты; сквозные тесты; внедрение зависимостей; параллельное выполнение тестов; этапы

тестирования; пирамида тестирования; статический анализ; проверка свойств.

•*Глава 8 «Как использовать Terraform в команде».* Как внедрить Terraform в командную работу; как убедить начальство; рабочий процесс развертывания кода приложения; рабочий процесс развертывания инфраструктурного кода; управление версиями; золотое правило Terraform; разбор кода; рекомендации по оформлению кода; принятый в Terraform стиль; CI/CD для Terraform; процесс развертывания.

Эту книгу можно читать последовательно или сразу переходить к тем главам, которые вас больше всего интересуют. Имейте в виду, что все примеры последующих глав основаны на коде из предыдущих. Если вы листаете туда-сюда, используйте в качестве ориентира архив исходного кода (как описано в разделе «Примеры с открытым исходным кодом» далее). В приложении вы найдете список книг и статей о Terraform, системном администрировании, IaC и DevOps.

Что нового во втором издании

Первое издание вышло в 2017 году. В мае 2019-го я готовил второе издание и был очень удивлен тому, как все изменилось за пару лет! Эта книга по своему объему почти в два раза превосходит предыдущую и включает две полностью новые главы. Кроме того, существенно обновлены все оригинальные главы и примеры кода.

Если вы уже прочитали первое издание и хотите узнать, что изменилось, или вам просто интересно посмотреть, как эволюционировал проект Terraform, вот несколько основных моментов.

•*Четыре крупных обновления Terraform*. Когда вышла первая книга, стабильной версией Terraform была 0.8. Спустя четыре крупных обновления Terraform имеет версию 0.12. За это время появились некоторые поразительные новшества, о которых пойдет речь далее. Чтобы обновиться, пользователям придется попотеть!1

• *Улучшения в автоматическом тестировании*. Существенно эволюционировали методики и инструментарий написания автоматических тестов для кода Terraform. Тестированию посвящена новая глава, седьмая, которая затрагивает такие темы, как модульные, интеграционные и сквозные тесты, внедрение зависимостей, распараллеливание тестов, статический анализ и др.

• *Улучшения в модулях*. Инструментарий и методики создания модулей Terraform тоже заметно эволюционировали. В новой, шестой, главе вы найдете руководство по написанию испытанных модулей промышленного

уровня с возможностью повторного использования — таких, которым можно доверить благополучие своей компании.

• *Улучшения в рабочем процессе*. Глава 8 была полностью переписана согласно тем изменениям, которые произошли в процедуре интеграции Terraform в рабочий процесс команд. Там, помимо прочего, можно найти подробное руководство о том, как провести прикладной и инфраструктурный код через все основные этапы: разработку, тестирование и развертывание в промышленной среде.

• *HCL2*. В Terraform 0.12 внутренний язык HCL обновился до HCL2. Это включает в себя поддержку полноценных выражений (чтобы вам не приходилось заворачивать все в ${…}!), развитые ограничители типов, условные выражения с отложенным вычислением, поддержку выражений null, for_each и for, вложенные блоки и др. Все примеры кода в этой книге были адаптированы для HCL2, а новые возможности языка подробно рассматриваются в главах 5 и 6.

• *Переработанные механизмы хранения состояния*. В Terraform 0.9 появилась концепция внутренних хранилищ. Это полноценный механизм хранения и разделения состояния Terraform со встроенной поддержкой блокирования. В Terraform 0.9 также были представлены окружения состояния, которые позволяют управлять развертываниями в разных средах; но уже в версии 0.10 им на смену пришли рабочие области. Все эти темы рассматриваются в главе 3.

• *Вынос провайдеров из ядра Terraform*. В Terraform 0.10 из ядра был вынесен код для всех провайдеров (то есть код для AWS, GCP, Azure и т. д.). Благодаря этому разработка провайдеров теперь ведется в отдельных репозиториях, в своем собственном темпе и с выпуском независимых версий. Однако теперь придется загружать код провайдера с помощью команды terraforminit каждый раз, когда вы начинаете работать с новым модулем. Об этом пойдет речь в главах 2 и 7.

• *Большое количество новых провайдеров*. В 2016 году проект Terraform официально поддерживал лишь несколько основных облачных провайдеров (AWS, GCP и Azure). Сейчас же их количество превысило 100, а провайдеров, разрабатываемых сообществом, и того больше2. Благодаря этому вы можете использовать код для работы не только с множеством разных облаков (например, теперь существуют провайдеры для Alicloud, Oracle Cloud Infrastructure, VMware vSphere и др.), но и с другими аспектами окружающего мира, включая системы управления версиями (GitHub, GitLab или BitBucket), хранилища данных (MySQL, PostreSQL или InfluxDB), системы мониторинга и оповещения (включая DataDog, New Relic или Grafana), платформы наподобие Kubernetes, Helm, Heroku, Rundeck или

Rightscale и многое другое. Более того, сейчас у каждого провайдера намного лучше покрытие: скажем, провайдер для AWS охватывает большинство сервисов этой платформы, а поддержка новых сервисов часто появляется даже раньше, чем у CloudFormation!

• *Реестр модулей Terraform*. В 2017 году компания HashiCorp представила реестр модулей Terraform (registry.terraform.io) — пользовательский интерфейс, который облегчает просмотр и загрузку открытых универсальных модулей Terraform, разрабатываемых сообществом. В 2018 году была добавлена возможность запускать внутри своей организации закрытый реестр. В Terraform 0.11 появился полноценный синтаксис для загрузки модулей из реестра. Подробнее об этом читайте в разделе «Управление версиями» на с. 153.

• *Улучшенная обработка ошибок*. В Terraform 0.9 обновилась обработка ошибок состояния: если при записи состояния в удаленное хранилище обнаруживается ошибка, это состояние сохраняется локально, в файле errored.tfstate. В Terraform 0.12 механизм был полностью переработан. Теперь ошибки перехватываются раньше, а сообщения о них стали более понятными и содержат путь к файлу, номер строчки и фрагмент кода.

• *Много других мелких изменений*. Было сделано много менее значительных изменений, включая появление локальных переменных (см. раздел «Локальные переменные модулей» на с. 144), новые «аварийные люки» для взаимодействия с внешним миром с помощью скриптов (например, подраздел «Модули вне Terraform» на с. 242), выполнение plan в рамках команды apply (см. раздел «Развертывание одного сервера» на с. 64), исправление циклических проблем с create_before_destroy, значительное улучшение параметра count, которое позволяет ссылаться в нем на источники данных и ресурсы (см. раздел «Циклы» на с. 160), десятки новых встроенных функций, обновленное наследование provider и многое другое.

Чего нет в этой книге

Книга не задумывалась как исчерпывающее руководство по Terraform. Она не охватывает все облачные провайдеры, все ресурсы, которые поддерживаются каждым из них, или каждую команду, доступную в этой системе. За этими подробностями я отсылаю вас к документации по адресу https://www.terraform.io/docs/index.html.

Документация содержит множество полезной информации, но, если вы только знакомитесь с Terraform, концепцией «инфраструктура как код» или системным администрированием, вы попросту не знаете, какие вопросы задавать. Поэтому данная книга сосредоточена на том, чего *нет* в документации: как выйти за рамки вводных примеров и начать использовать

Terraform в реальных условиях. Моя цель — быстро подготовить вас к работе с данной системой. Для этого мы обсудим, зачем вообще может понадобиться Terraform, как внедрить этот инструмент в рабочий процесс и какие методики и шаблоны проектирования обычно работают лучше всего.

Чтобы это продемонстрировать, я включил в книгу ряд примеров кода. Я пытался сделать так, чтобы вам было просто работать с ними в домашних условиях. Для этого минимизировал количество сторонних зависимостей. Именно поэтому везде используется лишь один облачный провайдер, AWS. Таким образом, вам нужно будет зарегистрироваться только в одном стороннем сервисе (к тому же AWS предлагает хороший бесплатный тариф, поэтому не придется ничего платить за выполнение примеров).

Примеры с открытым исходным кодом

Все доступные в этой книге примеры кода можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

Перед чтением можете скопировать репозиторий, чтобы иметь возможность выполнять примеры на своем компьютере:

git clone https://github.com/brikis98/terraform-up-and-running-code.git

Примеры кода в этом репозитории разбиты по главам. Стоит отметить, что большинство из них демонстрирует состояние кода на момент *завершения* главы. Если вы хотите научиться как можно большему, весь код лучше писать самостоятельно, с нуля.

Программирование начинается в главе 2, где вы научитесь развертывать кластер веб-серверов с помощью Terraform от начала и до конца. После этого следуйте инструкциям в каждой последующей главе, развивая и улучшая этот пример.

Вносите изменения так, как указано в книге, пытайтесь писать весь код самостоятельно и используйте примеры из репозитория в GitHub только для того, чтобы свериться или прояснить непонятные моменты.

Версии

Все примеры в этой книге проверены на версии Terraform 0.12.x, которая на момент написания является последним крупным обновлением. Поскольку Terraform — относительно новый инструмент, который все еще не достиг версии 1.0.0, вполне вероятно, что будущие выпуски будут

содержать обратно несовместимые изменения и некоторые из рекомендуемых методик со временем поменяются и эволюционируют.

Я попытаюсь выпускать обновления как можно чаще, но проект Terraform движется очень быстро. Чтобы не отставать, вам самим придется прилагать определенные усилия. Чтобы не пропустить последние новости, статьи и обсуждения, связанные с Terraform и DevOps, посещайте сайт этой книги по адресу http://www.terraformupandrunning.com/ и подпишитесь на информационную рассылку (http://www.terraformupandrunning.com/#newsletter)!

Использование примеров кода

Эта книга предназначена для того, чтобы помочь вам решать ваши задачи. Вы можете свободно использовать примеры кода в своих программах и документации. Если вы не воспроизводите существенную часть кода, не нужно с нами связываться. Это, скажем, касается ситуаций, когда вы включаете в свою программу несколько фрагментов кода, которые приводятся в книге. Однако продажа или распространение CD с примерами из книг издательства O'Reilly требует отдельного разрешения. Если вы цитируете эту книгу с примерами кода при ответе на вопрос, разрешение не требуется. Но нужно связаться с нами, если хотите включить существенную часть приводимого здесь кода в документацию своего продукта.

Мы приветствуем, но не требуем отсылки на оригинал. Отсылка обычно состоит из названия, имени автора, издательства, ISBN. Например: «Terraform: инфраструктура на уровне кода», Евгений Брикман. Питер, 2020. 978-5-4461-1590-7.

Если вам кажется, что то, как вы обращаетесь с примерами кода, выходит за рамки добросовестного использования или условий, перечисленных выше, можете обратиться к нам по адресу permissions@oreilly.com.

Условные обозначения

В этой книге используются следующие типографические обозначения.

Курсив

Обозначает новые термины и важные моменты.

Рубленый

Обозначает URL, адреса электронной почты и элементы интерфейса.

Моноширинный шрифт

Используется в листингах кода, а также в тексте, обозначая такие программные элементы, как имена переменных и функций, базы данных,

типы данных, переменные среды, операторы и ключевые слова, названия папок и файлов, а также пути к ним.

Жирный моноширинный шрифт

Обозначает команды или другой текст, который должен быть введен пользователем.

Курсивный моноширинный шрифт

Обозначает текст, вместо которого следует подставить пользовательские значения или данные, зависящие от контекста.

Этот значок обозначает примечание общего характера.

Этот значок обозначает предупреждение или предостережение.

Благодарности

Джош Падник

Без тебя эта книга не появилась бы. Ты тот, кто познакомил меня с Terraform, научил основам и помог разобраться со всеми сложностями. Спасибо, что поддерживал меня, пока я воплощал наши коллективные знания в книгу. Спасибо за то, что ты такой классный соучредитель. Благодаря тебе я могу заниматься стартапом и по-прежнему радоваться жизни. И больше всего я благодарен тебе за то, что ты хороший друг и человек.

O'Reilly Media

Спасибо за то, что выпустили еще одну мою книгу. Чтение и написание книг коренным образом изменили мою жизнь, и я горжусь тем, что вы помогаете мне делиться некоторыми из моих текстов с другими. Отдельная благодарность Брайану Андерсону за его помощь в подготовке первого издания в рекордные сроки и Вирджинии Уилсон, благодаря которой мне каким-то образом удалось поставить новый рекорд со вторым изданием.

Сотрудники Gruntwork

Не могу выразить, насколько я благодарен вам всем за то, что вы присоединились к нашему крошечному стартапу. Вы создаете потрясающее ПО! Спасибо, что удерживали компанию на плаву, пока я работал над вторым изданием этой книги. Вы замечательные коллеги и друзья.

Клиенты Gruntwork

Спасибо, что рискнули связаться с мелкой, неизвестной компанией и согласились стать подопытными кроликами для наших экспериментов с Terraform. Задача Gruntwork — на порядок упростить понимание, разработку и развертывание ПО. Нам не всегда это удается (в книге я описал многие из наших ошибок!), поэтому я благодарен за ваше терпение и желание принять участие в нашей дерзкой попытке улучшить мир программного обеспечения.

HashiCorp

Спасибо за создание изумительного набора инструментов для DevOps, включая Terraform, Packer, Consul и Vault. Вы улучшили мир DevOps, а заодно и жизни миллионов разработчиков.

Киф Моррис, Сет Варго, Маттиас Гис, Рокардо Феррейра, Акаш Махаян, Мориц Хейбер

Спасибо за вычитку первых черновиков книги и за большое количество подробных и конструктивных отзывов. Ваши советы улучшили эту книгу.

Читатели первого издания

Те из вас, кто купил первое издание, сделали возможным создание второго. Спасибо. Ваши отзывы, вопросы, предложения относительно исходного кода и постоянная жажда новостей послужили мотивацией примерно для 160 дополнительных страниц нового материала. Надеюсь, новый текст окажется полезным, и с нетерпением жду дальнейшего давления со стороны читателей.

Мама, папа, Лайла, Молли

Так получилось, что я написал еще одну книгу. Это, скорее всего, означает, что я проводил с вами меньше времени, чем мне бы хотелось. Спасибо за то, что отнеслись к этому с пониманием. Я вас люблю.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Подробности ищите в руководствах по обновлению Terraform по адресу www.terraform.io/upgrade-guides/index.html.

2 Список провайдеров для Terraform можно найти на странице www.terraform.io/docs/providers/.

1. Почему Terraform

Программное обеспечение (ПО) нельзя считать завершенным, если оно просто работает на вашем компьютере, проходит тесты и получает одобрение при обзоре кода (code review). ПО не готово, пока вы не *доставите* его пользователю.

Доставка ПО включает в себя множество задач, которые необходимо решить для того, чтобы сделать код доступным для клиента. Это подразумевает запуск кода на боевых (production) серверах, обеспечение его устойчивости к перебоям в работе и всплескам нагрузки, защиту от злоумышленников. Прежде чем погружаться в мир Terraform, стоит сделать шаг назад и поговорить о роли этого инструмента в большом деле доставки программного обеспечения.

В этой главе мы подробно обсудим следующие темы.

• Появление DevOps.

• Что такое инфраструктура как код.

• Как работает Terraform.

• Сравнение Terraform с другими инструментами для работы с инфраструктурой как с кодом.

Появление DevOps

Если бы в недалеком прошлом вы захотели создать компанию — разработчик ПО, вам бы пришлось иметь дело с большим количеством оборудования. Для этого нужно было бы подготовить шкафы и стойки, поместить в них серверы, подключить кабели и провода, установить систему охлаждения, предусмотреть резервные системы питания и т. д. В те дни было логично разделять работников на две команды: разработчиков (developers, Devs), которые занимались написанием программ, и системных администраторов (operations, Ops), в чьи обязанности входило управление этим оборудованием.

Типичная команда разработчиков собирала свое приложение и «перебрасывала его через стену» команде сисадминов. Последние должны были разобраться с тем, как его развертывать и запускать. Большая часть этого процесса выполнялась вручную. Отчасти потому, что это неизбежно требовало подключения физического аппаратного обеспечения (например, расстановки серверов по стойкам и разводки сетевых кабелей). Но программные аспекты работы системных администраторов, такие как установка приложений и их зависимостей, часто выполнялись вручную путем выполнения команд на сервере.

До поры до времени этого было достаточно, но по мере роста компании начинали возникать проблемы. Обычно выглядело так: поскольку ПО доставлялось вручную, а количество серверов увеличивалось, выпуск новых

версий становился медленным, болезненным и непредсказуемым. Команда сисадминов иногда допускала ошибки, и в итоге некоторые серверы требовали немного других настроек по сравнению со всеми остальными (это проблема, известная как *дрейф конфигурации*). Поэтому росло число программных ошибок. Разработчики пожимали плечами и отвечали: «У меня на компьютере все работает!» Перебои в работе становились все более привычными.

Команда администраторов, уставшая от вызовов в три часа ночи после каждого выпуска, решала снизить частоту выпуска новых версий до одного в неделю. Затем происходил переход на месячный и в итоге на полугодовой цикл. За несколько недель до полугодового выпуска команды пытались объединить все свои проекты, что приводило к большой неразберихе с конфликтами слияния. Никому не удавалось стабилизировать основную ветку. Команды начинали винить друг друга. Возникало недоверие. Работа в компании останавливалась.

В наши дни ситуация меняется коренным образом. Вместо обслуживания собственных вычислительных центров многие компании переходят в облако, пользуясь преимуществами таких сервисов, как Amazon Web Services (AWS), Microsoft Azure и Google Cloud Platform (GCP). Вместо того чтобы тесно заниматься оборудованием, многие команды системных администраторов проводят все свое время за работой с программным обеспечением, используя инструменты вроде Chef, Puppet, Terraform и Docker. Вместо расставления серверов по стойкам и подключения сетевых кабелей многие сисадмины пишут код.

В итоге обе команды, Dev и Ops, в основном занимаются работой с ПО, и граница между ними постепенно размывается. Возможно, наличие отдельных команд, отвечающих за прикладной и инфраструктурный код, все еще имеет смысл, но уже сейчас очевидно, что обе они должны работать вместе более тесно. Вот где берет начало *движение DevOps*.

DevOps не название команды, должности или какой-то определенной технологии. Это набор процессов, идей и методик. Каждый понимает под DevOps что-то свое, но в этой книге я буду использовать следующее определение: *цель DevOps — значительно повысить эффективность доставки ПО.*

Вместо многодневного кошмара со слиянием веток вы постоянно интегрируете свой код, поддерживая его в развертываемом состоянии. Вместо ежемесячных развертываний вы можете доставлять свой код десятки раз в день или даже после каждой фиксации. Вместо того чтобы иметь дело с постоянными простоями и перебоями в работе, вы сможете создавать устойчивые системы с автоматическим восстановлением,

используя мониторинг и оповещения для обнаружения проблем, которые требуют ручного вмешательства.

Компании, прошедшие через подобные трансформации, показывают изумительные результаты. Например, после применения в своей организации методик DevOps компания Nordstrom сумела удвоить количество выпускаемых ежемесячно функций, уменьшить число дефектов на 50 %, сократить сроки реализации идей в промышленных условиях на 60 % и снизить частоту сбоев в процессе эксплуатации ПО на 60–90 %. После того как в подразделении LaserJet Firmware компании HP стали использовать методики DevOps, доля времени, затрачиваемого на разработку новых возможностей, увеличилась с 5 до 40 %, а общая стоимость разработки была снижена на 40 %. До внедрения DevOps доставка кода в компании Etsy была нечастым процессом, сопряженным со стрессом и многочисленными перебоями в работе. Теперь развертывания выполняются по 25–50 раз в день с куда меньшим количеством проблем3.

Движение DevOps основано на четырех принципах: культуре, автоматизации, измерении и разделении (в англ. языке иногда используется аббревиатура CASM (http://bit.ly/2GS3CR3) — culture, automation, sharing, measurement). Эта книга не задумывалась как комплексный обзор DevOps (рекомендуемая литература приводится в приложении), поэтому сосредоточусь лишь на одном из указанных принципов: автоматизации.

Наша задача — автоматизировать как можно больше аспектов процесса доставки программного обеспечения. Это означает, что вы будете управлять своей инфраструктурой через код, а не с помощью веб-страницы или путем ввода консольных команд. Такую концепцию обычно называют *«инфраструктура как код»* (infrastructure as code, или IaC).

Что такое инфраструктура как код

Идея, стоящая за IaC, заключается в том, что для определения, развертывания, обновления и удаления инфраструктуры нужно писать и выполнять код. Это важный сдвиг в образе мышления, когда все аспекты системного администрирования воспринимаются как программное обеспечение — даже те, которые представляют оборудование (например, настройка физических серверов). Ключевым аспектом DevOps является то, что почти *всем* можно управлять внутри кода, включая серверы, базы данных, сети, журнальные файлы, программную конфигурацию, документацию, автоматические тесты, процессы развертывания и т. д.

Инструменты IaC можно разделить на пять общих категорий:

- специализированные скрипты;
- средства управления конфигурацией;

• средства шаблонизации серверов;
• средства оркестрации;
• средства инициализации ресурсов.

Рассмотрим каждую из них.

Специализированные скрипты

Самый простой и понятный способ что-либо автоматизировать — написать для этого *специальный скрипт*. Вы берете задачу, которая выполняется вручную, разбиваете ее на отдельные шаги, описываете каждый шаг в виде кода, используя любимый скриптовый язык, и выполняете получившийся скрипт на своем сервере, как показано на рис. 1.1.

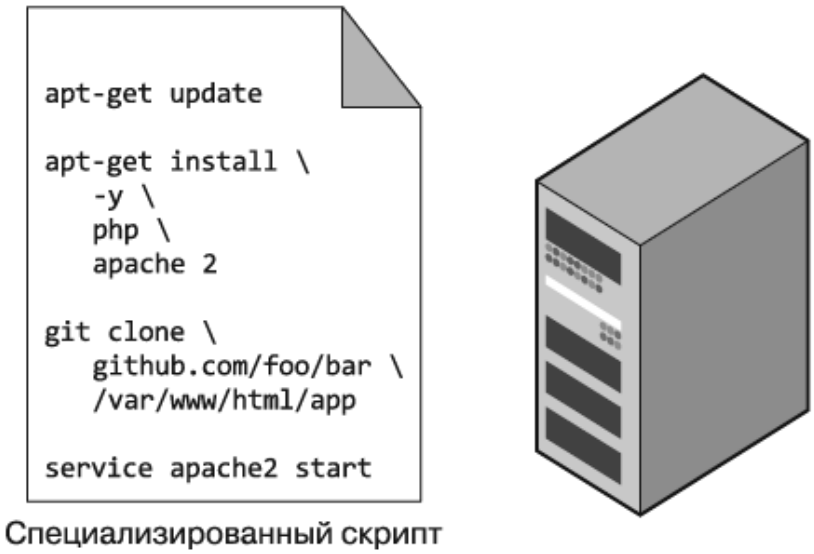


Рис. 1.1. Выполнение специализированного скрипта на сервере

Например, ниже показан bash-скрипт setup-webserver.sh, который конфигурирует веб-сервер, устанавливая зависимости, загружая код из Git-репозитория и запуская Apache:

```
# Обновляем кэш apt-get
sudo apt-get update
# Устанавливаем PHP и Apache
sudo apt-get install -y php apache2
# Копируем код из репозитория
sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app
# Запускаем Apache
sudo service apache2 start
```

Крайне удобной особенностью специализированных скриптов (и их огромным недостатком) является то, что код можно писать как угодно и с использованием популярных языков программирования общего назначения.

Если инструменты, специально созданные для IaC, предоставляют лаконичный API для выполнения сложных задач, языки программирования общего назначения подразумевают написание своего кода в каждом отдельно взятом случае. Более того, средства IaC обычно навязывают определенную структуру кода, тогда как в специализированных скриптах каждый разработчик использует собственный стиль и делает вещи по-своему. Если речь идет о скрипте из восьми строк, который устанавливает Apache, обе проблемы можно считать незначительными, но, если вы попытаетесь применить тот же подход к управлению десятками серверов, базами данных, балансировщиками нагрузки и сетевой конфигурацией, все пойдет наперекосяк.

Если вам когда-либо приходилось поддерживать большой репозиторий bash-скриптов, вы знаете, что это почти всегда превращается в «кашу» из плохо структурированного кода. Специализированные скрипты отлично подходят для небольших одноразовых задач, но, если вы собираетесь управлять всей своей инфраструктурой в виде кода, следует использовать специально предназначенный для этого инструмент IaC.

Средства управления конфигурацией

Chef, Puppet, Ansible и SaltStack являются *средствами управления конфигурацией*. Это означает, что они предназначены для установки и администрирования программного обеспечения на существующих серверах. Например, ниже показана *роль Ansible* под названием web-server.yml, которая настраивает тот же веб-сервер Apache, что и скрипт setup-webserver.sh:

```
- name: Update the apt-get cache
  apt:
    update_cache: yes
- name: Install PHP
  apt:
    name: php
- name: Install Apache
  apt:
    name: apache2
- name: Copy the code from the repository
  git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app
- name: Start Apache
  service: name=apache2 state=started enabled=yes
```

Этот код похож на bash-скрипт, но использование такого инструмента, как Ansible, дает ряд преимуществ.

•*Стандартизированное оформление кода*. Ansible требует, чтобы код имел предсказуемую структуру. Это касается документации, структуры файлов и каталогов, параметров с понятными именами, управления конфиденциальными данными и т. д. Если каждый разработчик организует свои специализированные скрипты по-разному, то большинство средств управления конфигурацией имеют набор правил и соглашений, которые упрощают навигацию по коду.

• *Идемпотентность*4. Написать рабочий специализированный скрипт не так уж и трудно. Намного сложнее написать скрипт, который будет работать корректно вне зависимости от того, сколько раз вы его запустите. Каждый раз, когда вы создаете в своем скрипте папку, нужно убедиться, что ее еще не существует. Всякий раз, когда вы добавляете строчку в конфигурационный файл, необходимо проверить, существует ли эта строчка. И всегда, когда вы хотите запустить программу, надо определить, выполняется ли она в данный момент.

Код, который работает корректно независимо от того, сколько раз вы его запускаете, называется *идемпотентным*. Чтобы сделать идемпотентным bash-скрипт из предыдущего раздела, придется добавить много строчек кода, включая уйму условных выражений. Для сравнения: большинство функций Ansible идемпотентно по умолчанию. Например, роль Ansible web-server.yaml установит сервер Apache только в случае, если он еще не установлен, и попытается его запустить лишь при условии, что он еще не выполняется.

• *Распределенность*. Специализированные скрипты предназначены для выполнения на одном локальном компьютере. Ansible и другие средства управления конфигурацией специально «заточены» под работу с большим количеством удаленных серверов, как показано на рис. 1.2.

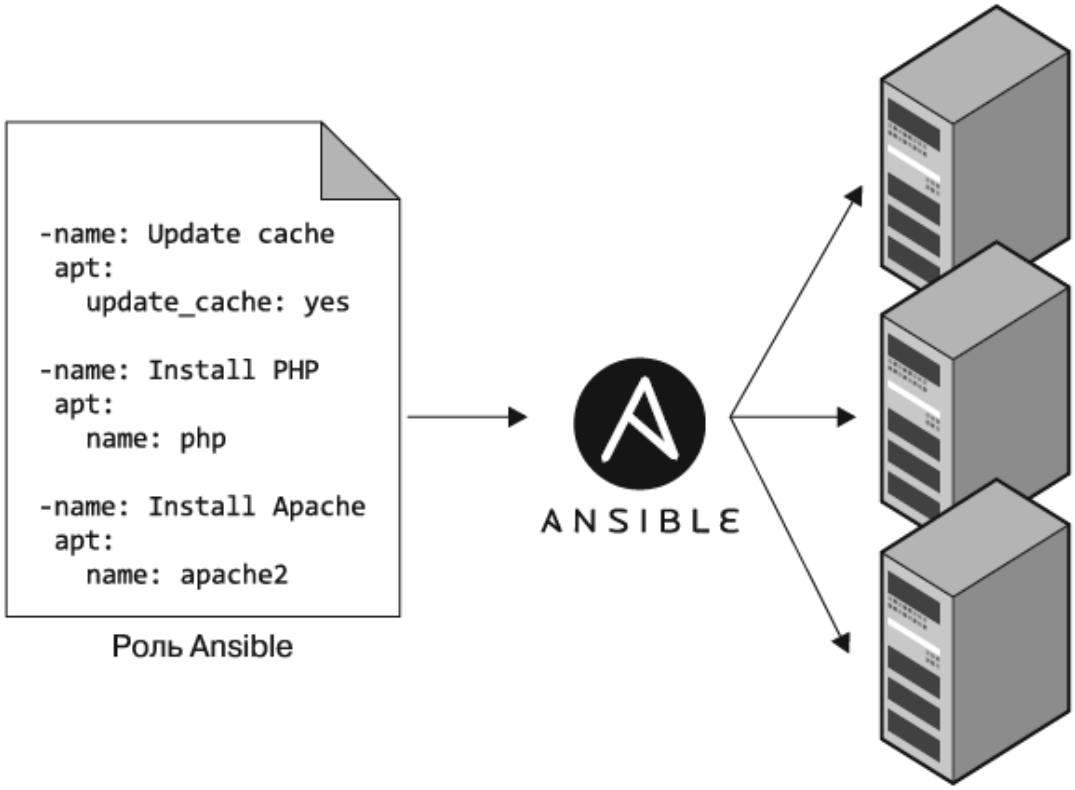


Рис. 1.2. Такие средства управления конфигурацией, как Ansible, способны выполнять ваш код на большом количестве серверов

Например, чтобы применить роль web-server.yml к пяти серверам, нужно сначала создать файл под названием hosts, который содержит IP-адреса этих серверов:

[webservers]
11.11.11.11
11.11.11.12
11.11.11.13
11.11.11.14
11.11.11.15

Далее следует определить такой *плейбук (playbook) Ansible*:

- hosts: webservers
  roles:
  - webserver

И в конце этот плейбук выполняется:

ansible-playbook playbook.yml

Это заставит Ansible параллельно сконфигурировать все пять серверов. В качестве альтернативы в плейбуке можно указать параметр под названием serial. Это позволит выполнить *скользящее развертывание*, которое обновит серверы пакетным образом. Например, если присвоить serial значение 2,

Ansible будет обновлять сразу по два сервера, пока не будут обновлены все пять. Дублирование любой части этой логики в специализированных скриптах потребовало бы написания десятков или даже сотен строчек кода.

Средства шаблонизации серверов

Альтернативой управлению конфигурацией, набирающей популярность в последнее время, являются *средства шаблонизации серверов*, такие как Docker, Packer и Vagrant. Вместо того чтобы вводить кучу серверов и настраивать их, запуская на каждом один и тот же код, средства шаблонизации создают *образ* сервера, содержащий полностью самодостаточный «снимок» операционной системы (ОС), программного обеспечения, файлов и любых других важных деталей. Затем, как показано на рис. 1.3, этот образ можно будет установить на все ваши серверы, используя другие инструменты IaC.

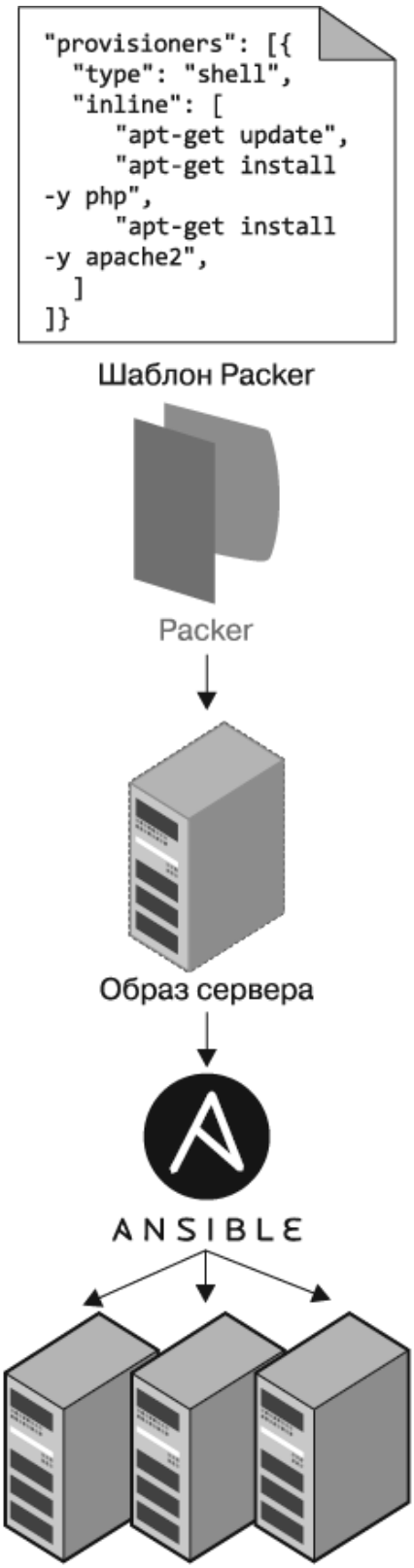


Рис. 1.3. С помощью таких средств шаблонизации, как Packer, можно создавать самодостаточные образы серверов. Затем, используя другие

инструменты, такие как Ansible, эти образы можно установить на все ваши серверы

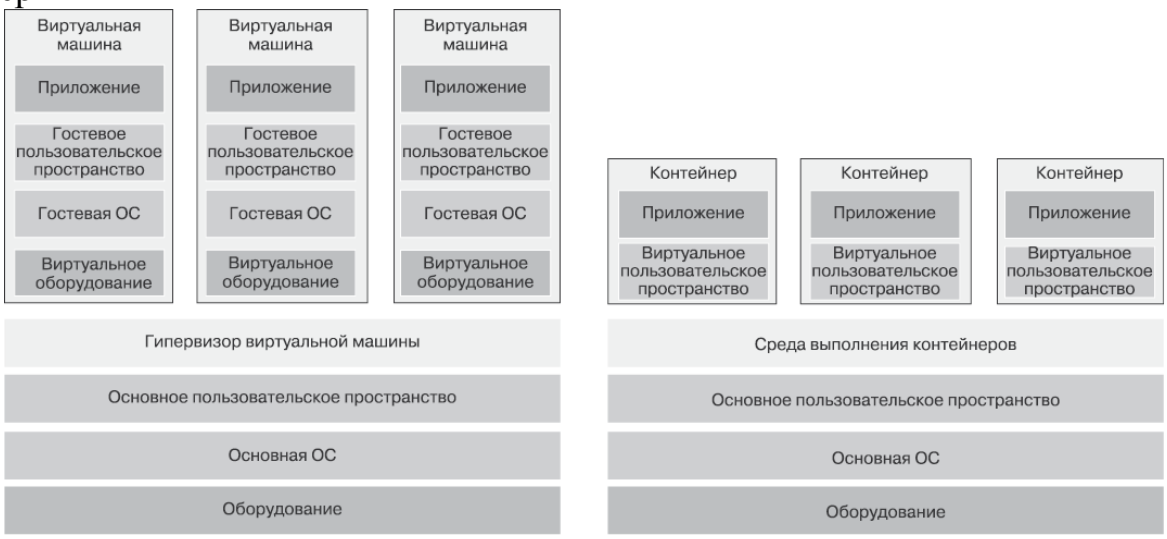


Рис. 1.4. Существует два вида образов: ВМ *(слева)* и контейнеры *(справа)*. ВМ виртуализируют оборудование, тогда как контейнеры — только пользовательское пространство

Как видно на рис. 1.4, средства для работы с образами можно разделить на две общие категории.

•*Виртуальные машины* эмулируют весь компьютер, включая аппаратное обеспечение. Для виртуализации (то есть симуляции) процессора, памяти, жесткого диска и сети запускается *гипервизор*, такой как VMWare, VirtualBox или Parallels. Преимущество подхода — любой *образ ВМ*, который работает поверх гипервизора, может видеть только виртуальное оборудование, поэтому он полностью изолирован от физического компьютера и любых других образов ВМ. И он выполняется аналогично во всех средах (например, на вашем компьютере, сервере проверки качества и боевом сервере). Недостаток в том, что виртуализация всего этого оборудования и запуск совершенно отдельной ОС для каждой ВМ требует большого количества ресурсов процессора и памяти, что влияет на время запуска. Образы ВМ можно описывать в виде кода, применяя такие инструменты, как Packer и Vagrant.

•*Контейнеры* эмулируют пользовательское пространство ОС5. Для изоляции процессов, памяти, точек монтирования и сети запускается *среда выполнения контейнеров*, такая как Docker, CoreOS rkt или cri-o. Преимущество этого подхода в том, что любой контейнер, который выполняется в данной среде, может видеть только собственно пользовательское пространство, поэтому он изолирован от основного

компьютера и других контейнеров. При этом он ведет себя одинаково в любой среде (например, на вашем компьютере, сервере проверки качества, боевом сервере и т. д.). Но есть и недостаток: все контейнеры, запущенные на одном сервере, одновременно пользуются ядром его ОС и его оборудованием, поэтому достичь того уровня изоляции и безопасности, который вы получаете в ВМ, намного сложнее6. Поскольку применяются общие ядро и оборудование, ваши контейнеры могут загружаться в считаные миллисекунды и практически не будут требовать дополнительных ресурсов процессора или памяти. Образы контейнеров можно описывать в виде кода, используя такие инструменты, как Docker и CoreOS rkt.

Например, ниже представлен шаблон Packer под названием web-server.json, создающий *Amazon Machine Image* (AMI) — образ ВМ, который можно запускать в AWS:

```
{
 "builders": [{
  "ami_name": "packer-example",
  "instance_type": "t2.micro",
  "region": "us-east-2",
  "type": "amazon-ebs",
  "source_ami": "ami-0c55b159cbfafe1f0",
  "ssh_username": "ubuntu"
 }],
 "provisioners": [{
  "type": "shell",
  "inline": [
   "sudo apt-get update",
   "sudo apt-get install -y php apache2",
                "sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app"
  ],
  "environment_vars": [
   "DEBIAN_FRONTEND=noninteractive"
  ]
 }]
}
```

Шаблон Packer настраивает тот же веб-сервер Apache, который мы видели в файле setup-webserver.sh, и использует тот же код на bash7. Единственное отличие от предыдущего кода в том, что Packer не запускает веб-сервер Apache (с помощью команды вроде sudoserviceapache2start). Дело в том, что шаблоны серверов обычно применяются для установки ПО в

образах, а запуск этого ПО должен происходить во время выполнения образа (например, когда он будет развернут на сервере).

Вы можете создать AMI из этого шаблона, запустив команду packerbuildwebserver.json. Когда сборка завершится, полученный образ AMI можно будет установить на все ваши серверы в AWS и сконфигурировать Apache для запуска во время загрузки компьютера (пример этого см. в подразделе «Средства оркестрации» на с. 35). В результате все они будут запущены абсолютно одинаково.

Имейте в виду, что разные средства шаблонизации серверов имеют различное назначение. Packer обычно используется для создания образов, выполняемых непосредственно поверх боевых серверов, таких как AMI (доступных для работы в вашей промышленной учетной записи). Образы, созданные в Vagrant, обычно запускаются на компьютерах для разработки. Это, к примеру, может быть образ VirtualBox, который работает на вашем ноутбуке под управлением Mac или Windows. Docker обычно делает образы для отдельных приложений. Их можно запускать на промышленных или локальных компьютерах при условии, что вы сконфигурировали на них Docker Engine, используя какой-то другой инструмент. Например, с помощью Packer часто создают образы AMI, у которых внутри установлен Docker Engine; дальше эти образы развертываются на кластере серверов в вашей учетной записи AWS, и затем в этот кластер доставляются отдельные контейнеры Docker для выполнения ваших приложений.

Шаблонизация серверов — это ключевой аспект перехода на *неизменяемую инфраструктуру*. Идея навеяна функциональным программированием, которое предполагает наличие «неизменяемых переменных». То есть после инициализации переменной ее значение больше нельзя изменить. Если нужно что-то обновить, вы создаете новую переменную. Благодаря этому код становится намного более понятным.

Неизменяемая инфраструктура работает по тому же принципу: если сервер уже развернут, в него больше не вносятся никакие изменения. Если нужно что-то обновить (например, развернуть новую версию кода), вы создаете новый образ из своего шаблона и развертываете его на новый сервер. Поскольку серверы никогда не меняются, вам намного проще следить за тем, что на них развернуто.

Средства оркестрации

Средства шаблонизации серверов отлично подходят для создания ВМ и контейнеров, но как ими после этого управлять? В большинстве реальных сценариев применения вам нужно выбрать какой-то способ выполнения следующих действий.

• Развертывать ВМ и контейнеры с целью эффективного использования ресурсов оборудования.

• Выкатывать обновления для своих многочисленных ВМ и контейнеров, используя такие стратегии, как скользящие, «сине-зеленые» и канареечные развертывания.

• Следить за работоспособностью своих ВМ и контейнеров, автоматически заменяя неисправные *(автовосстановление)*.

• Масштабировать количество ВМ и контейнеров в обе стороны в зависимости от нагрузки *(автомасштабирование)*.

• Распределять трафик между своими ВМ и контейнерами *(балансировка нагрузки)*.

• Позволять своим ВМ и контейнерам находить друг друга и общаться между собой по сети *(обнаружение сервисов)*.

Выполнение этих задач находится в сфере ответственности *средств оркестрации*, таких как Kubernetes, Marathon/Mesos, Amazon Elastic Container Service (Amazon ECS), Docker Swarm и Nomad. Например, Kubernetes позволяет описывать и администрировать контейнеры Docker в виде кода. Вначале развертывается *кластер Kubernetes*, который представляет собой набор серверов для выполнения ваших контейнеров Docker. У большинства облачных провайдеров есть встроенная поддержка развертывания управляемых кластеров Kubernetes: вроде Amazon Elastic Container Service for Kubernetes (Amazon EKS), Google Kubernetes Engine (GKE) и Azure Kubernetes Service (AKS).

Подготовив рабочий кластер, вы можете описать развертывание своего контейнера Docker в виде кода внутри YAML-файла:

```
apiVersion: apps/v1
# Используем объект Deployment для развертывания нескольких реплик вашего
# Docker-контейнера (возможно, больше одного) и декларативного выкатывания
# обновлений для него
kind: Deployment
# Метаданные этого развертывания, включая его имя
metadata:
  name: example-app
# Спецификация, которая конфигурирует это развертывание
spec:
  # Благодаря этому развертывание знает, как искать ваш контейнер
  selector:
    matchLabels:
```

```
      app: example-app
  # Приказываем объекту Deployment развернуть три реплики Docker-контейнера
  replicas: 3
  # Определяет способ обновления развертывания. Здесь указываем скользящие обновления
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate
  # Этот шаблон описывает, какие контейнеры нужно развернуть
  template:
  # Метаданные контейнера, включая метки
  metadata:
    labels:
      app: example-app
  # Спецификация контейнера
  spec:
    containers:
      # Запускаем Apache на порту 80
      - name: example-app
        image: httpd:2.4.39
        ports:
          - containerPort: 80
```

Этот файл говорит Kubernetes, что нужно создать *развертывание*, которое декларативно описывает следующее.

• Один или несколько контейнеров Docker для совместного запуска. Эта группа контейнеров называется *подом*, или *под-оболочкой*. Под, описанный в приведенном выше коде, содержит единственный контейнер, который запускает Apache.

• Настройки каждого контейнера Docker в под-оболочке. В нашем примере под-оболочка настраивает Apach для прослушивания порта 80.

• Сколько копий *(реплик)* под-оболочки должно быть в вашем кластере. У нас указано три реплики. Kubernetes автоматически определяет, в какой области кластера их следует развернуть, используя алгоритм планирования для выбора оптимальных серверов с точки зрения высокой доступности (например, каждая под-оболочка может оказаться на отдельном сервере, чтобы сбой на одном из них не остановил работу всего приложения), ресурсов (скажем, выбираются серверы с доступными портами,

процессором, памятью и другими ресурсами, необходимыми вашему контейнеру), производительности (в частности, выбираются наименее загруженные серверы) и т. д. Кроме того, Kubernetes постоянно следит за тем, чтобы в кластере всегда было три реплики. Для этого автоматически заменяется любая под-оболочка, вышедшая из строя или переставшая отвечать.

• Как развертывать обновления. Когда выходит новая версия контейнера Docker, наш код выкатывает три новые реплики, ждет, когда они станут работоспособными, и затем удаляет три старые копии.

Так много возможностей всего в нескольких строчках на YAML! Чтобы развернуть свое приложение в Kubernetes, нужно выполнить команду kubectlapply-fexample-app.yml. Чтобы выкатить обновления, вы можете отредактировать YAML-файл и снова запустить kubectlapply.

Средства инициализации ресурсов

В отличие от инструментов для управления конфигурацией, шаблонизации серверов и оркестрации, код которых выполняется на каждом сервере, *средства инициализации ресурсов*, такие как Terraform, CloudFormation и OpenStack Heat, отвечают за создание самих серверов. С их помощью можно создавать не только серверы, но и базы данных, кэши, балансировщики нагрузки, очереди, системы мониторинга, настройки подсетей и брандмауэра, правила маршрутизации, сертификаты SSL и почти любой другой аспект вашей инфраструктуры (рис. 1.5).

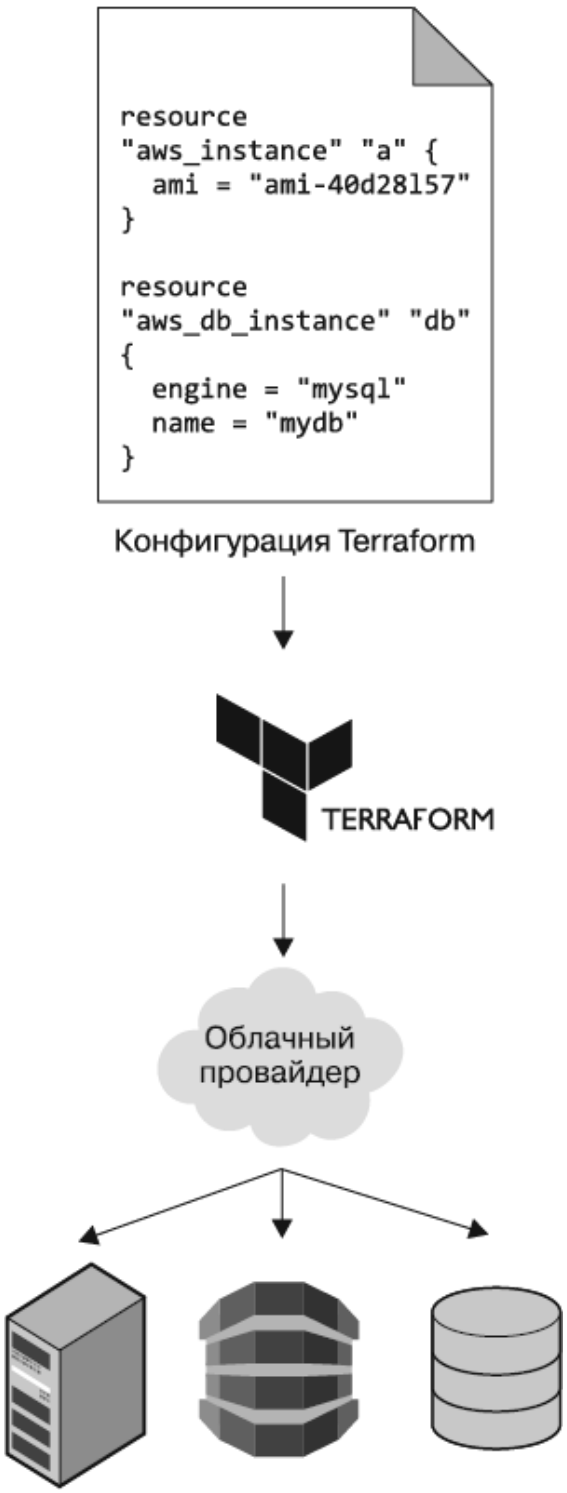


Рис. 1.5. Средства инициализации ресурсов можно использовать в связке с вашим облачным провайдером, чтобы создавать серверы, базы данных, балансировщики нагрузки и любые другие элементы вашей инфраструктуры

Например, следующий код развертывает веб-сервер с помощью Terraform:

```
resource "aws_instance" "app" {
  instance_type     = "t2.micro"
  availability_zone = "us-east-2a"
  ami               = "ami-0c55b159cbfafe1f0"
  user_data = <<-EOF
              #!/bin/bash
              sudo service apache2 start
              EOF
}
```

Не нужно волноваться, если вам непонятны какие-то элементы данного синтаксиса. Пока что сосредоточьтесь на двух параметрах.

•ami определяет идентификатор образа AMI, который нужно развернуть на сервере. Вы можете присвоить ему ID образа, собранного из шаблона Packer web-server.json в подразделе «Средства оркестрации» на с. 35. В нем содержатся PHP, Apache и исходный код приложения.

•user_data. Этот bash-скрипт выполняется при загрузке веб-сервера. В предыдущем примере этот скрипт используется для запуска Apache.

Иными словами, это демонстрация того, как объединить инициализацию ресурсов и шаблонизацию серверов, что является распространенной практикой в неизменяемой инфраструктуре.

Преимущества инфраструктуры как кода

Теперь, когда вы познакомились со всевозможными разновидностями IaC, можно задаться вопросом: зачем нам это нужно? Зачем изучать целую кучу новых языков и инструментов, обременяя себя еще большим количеством кода, который нужно поддерживать?

Дело в том, что код довольно мощный. Усилия, которые идут на преобразование ручных процессов в код, вознаграждаются огромным улучшением ваших возможностей по доставке ПО. Согласно докладу о состоянии DevOps за 2016 год (bit.ly/31kCUYX), организации, применяющие такие методики, как IaC, развертывают код в 200 раз чаще и восстанавливаются после сбоев в 24 раза быстрее, а на реализацию новых функций уходит в *2555* раз меньше времени.

Когда ваша инфраструктура определена в виде кода, можно существенно улучшить процесс доставки ПО, используя широкий диапазон методик из мира программирования. Это дает преимущества.

•*Самообслуживание*. В большинстве команд, которые развертывают код вручную, мало сисадминов (часто один), и только они знают все магические

заклинания для выполнения развертывания и имеют доступ к промышленной среде. Это становится существенным препятствием на пути роста компании. Если же ваша инфраструктура определена в виде кода, весь процесс развертывания можно автоматизировать, благодаря чему разработчики смогут доставлять свой код тогда, когда им это нужно.

• *Скорость и безопасность*. Автоматизация значительно ускоряет процесс развертывания, потому что компьютер может выполнить все его этапы куда быстрее человека. При этом повышается безопасность, так как автоматический процесс будет более последовательным, воспроизводимым и устойчивым к ошибкам с человеческим фактором.

• *Документация*. Вместо того чтобы держать состояние инфраструктуры в голове одного сисадмина, вы можете описать его в исходном файле, который каждый сможет прочитать. Иными словами, IaC играет роль документации, позволяя любому работнику компании понять, как все работает, даже если сисадмин уходит в отпуск.

• *Управление версиями*. Исходные файлы IaC можно хранить в системе управления версиями, благодаря чему в журнале фиксаций кода будет записана вся история вашей инфраструктуры. Это очень помогает при отладке, так как в случае возникновения проблемы всегда можно первым делом открыть журнал и посмотреть, что именно поменялось в вашей инфраструктуре. Вслед за этим проблему можно решить за счет простого отката к предыдущей версии кода IaC, в которой вы уверены.

• *Проверка*. Если состояние вашей инфраструктуры описано в файле, при каждом его изменении можно устраивать разбор кода, запускать набор автоматических тестов и прогонять его через средства статического анализа. Опыт показывает, что все это значительно уменьшает вероятность дефектов.

• *Повторное использование*. Вы можете упаковать свою инфраструктуру в универсальные модули, и вместо того, чтобы производить развертывание каждого продукта в каждой среде с нуля, у вас будет возможность использовать в качестве основы известные, задокументированные и проверенные на практике компоненты8.

•*Радость*. Есть еще одна очень важная причина, почему вы должны использовать IaC, которую часто упускают из виду: радость. Развертывание кода и управление инфраструктурой вручную — рутинный и утомительный процесс. Разработчики и сисадмины терпеть не могут такого рода работу, поскольку в ней нет никакого творчества, вызова или признания. Вы можете идеально развертывать код на протяжении месяцев, и никто даже не заметит, пока в один прекрасный день вы не напортачите. Это создает напряженную и неприятную обстановку. IaC предлагает лучшую альтернативу, которая

позволяет компьютерам и людям делать то, что они умеют лучше всего: автоматизировать и, соответственно, писать код.

Теперь вы понимаете, почему IaC важна. Следующий вопрос: является ли Terraform лучшим средством IaC именно для вас? Чтобы на это ответить, мы кратко рассмотрим принцип работы Terraform, а затем сравним его с другими популярными продуктами в этой области, такими как Chef, Puppet и Ansible.

Как работает Terraform

Вот обобщенная и немного упрощенная картина того, как работает Terraform. Terraform — это инструмент с открытым исходным кодом от компании HashiCorp, написанный на языке программирования Go. Код на Go компилируется в единый двоичный файл (если быть точным, по одному файлу для каждой поддерживаемой операционной системы) с предсказуемым названием terraform.

Этот файл позволяет развернуть инфраструктуру прямо с вашего ноутбука или сборочного сервера (либо любого другого компьютера), и для всего этого не требуется никакой дополнительной инфраструктуры. Все благодаря тому, что внутри исполняемый файл terraform делает от вашего имени API-вызовы к одному/нескольким провайдерам, таким как AWS, Azure, Google Cloud, DigitalOcean, OpenStack и т. д. Это означает, что Terraform использует инфраструктуру, которую эти провайдеры предоставляют для своих API-серверов, а также их механизмы аутентификации, которые вы уже применяете (например, ваши API-ключи для AWS).

Но откуда Terraform знает, какие API-вызовы нужно делать? Для этого вам необходимо создать текстовые файлы с *конфигурацией*, в которых описывается, какую инфраструктуру вы хотите создать. В концепции *«инфраструктура как код»* эти файлы играют роль кода. Вот пример конфигурации Terraform:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.example.public_ip]
```

}

Даже если вы никогда раньше не видели код Terraform, не должно быть особых проблем с тем, чтобы его понять. Этот фрагмент заставляет Terraform выполнить API-вызовы к двум провайдерам: к AWS, чтобы развернуть там сервер, и к Google Cloud, чтобы создать DNS-запись, которая указывает на IP-адрес сервера из AWS. Terraform позволяет использовать единый простой синтаксис (который вы изучите в главе 2) для развертывания взаимосвязанных ресурсов в нескольких разных облаках.

Вы можете описать всю свою инфраструктуру (серверы, базы данных, балансировщики нагрузки, топологию сети и т. д.) в конфигурационных файлах Terraform и сохранить их в системе управления версиями. Затем эту инфраструктуру можно будет развернуть с помощью определенных команд, таких как terraformapply. Утилита terraform проанализирует ваш код, преобразует его в последовательность API-вызовов к облачным провайдерам, которые в нем заданы, и выполнит эти API-вызовы от вашего имени максимально эффективным образом (рис. 1.6).

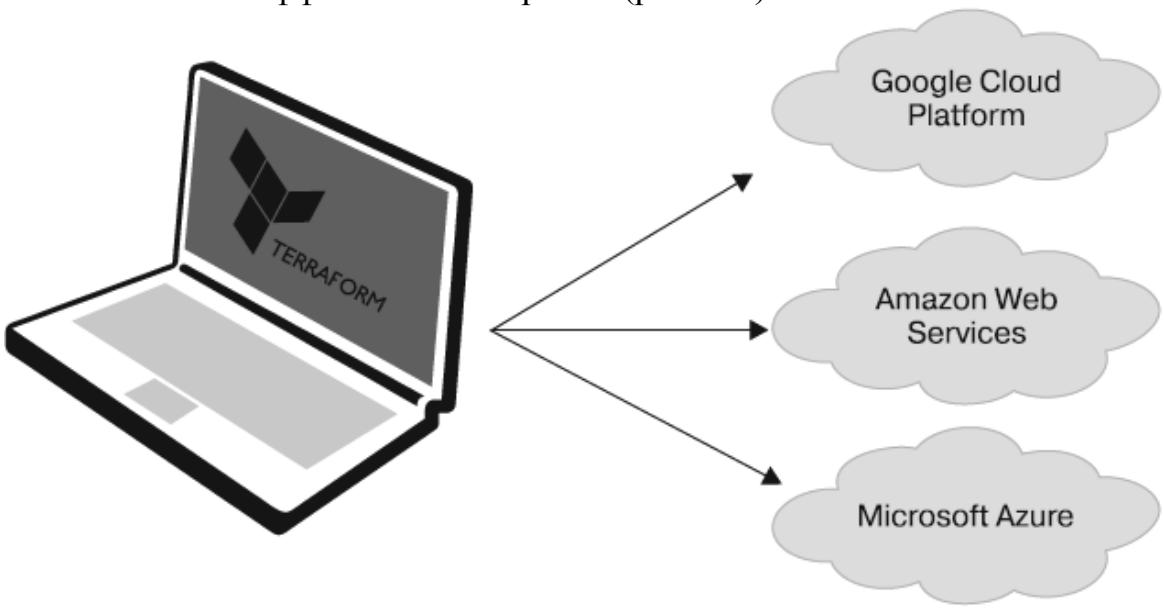


Рис. 1.6. Terraform — это утилита, которая преобразует содержимое ваших конфигурационных файлов в API-вызовы к облачным провайдерам

Если кто-то в вашей команде хочет изменить инфраструктуру, то вместо того, чтобы делать это вручную прямо на серверах, они редактируют конфигурационные файлы Terraform, проверяют их с помощью автоматических тестов и разбора кода, фиксируют обновленный код в системе управления версиями и затем выполняют команду terraformapply, чтобы сделать необходимые для развертывания изменений API-вызовы.

Прозрачная переносимость между облачными провайдерами

Поскольку Terraform поддерживает множество разных облачных провайдеров, часто возникает вопрос: обеспечивает ли этот инструмент прозрачную переносимость между ними? Например, если вы использовали Terraform для описания кучи серверов, баз данных, балансировщиков нагрузки и другой инфраструктуры в AWS, можете ли вы несколькими щелчками кнопкой мыши развернуть все это в другом облаке, таком как Azure или Google Cloud?

На практике этот вопрос оказывается не совсем корректным. Вы не можете развернуть «идентичную инфраструктуру» в разных облаках, поскольку инфраструктура, предоставляемая облачными провайдерами, разнится! Серверы, балансировщики нагрузки и базы данных, предлагаемые в AWS, Azure и Google Cloud, сильно различаются с точки зрения возможностей, конфигурации, управления, безопасности, масштабируемости, доступности, наблюдаемости и т. д. Не существует простого и прозрачного способа преодолеть эти различия, особенно учитывая то, что некоторые функции одного облачного провайдера часто отсутствуют во всех остальных.

Подход, который используется в Terraform, позволяет писать код для каждого провайдера отдельно, пользуясь его уникальными возможностями; при этом внутри для всех провайдеров применяются тот же язык, инструментарий и методики IaC.

Сравнение Terraform с другими средствами IaC

Инфраструктура как код — замечательная идея, чего нельзя сказать о процессе выбора инструментов IaC. Многие из них пересекаются в своих возможностях, имеют открытый исходный код и предоставляют коммерческую поддержку. Если вы не использовали каждый из них лично, не совсем понятно, по каким критериям следует выбирать.

Все усложняется тем, что большинство сравнений этих инструментов, которые вы найдете, просто перечисляют их общие характеристики. В итоге может показаться, что выбор любого из них будет одинаково удачным. Хотя теоретически все действительно так, это никак не помогает с выбором. Это как если бы вы уверяли новичка в программировании в том, что сайт можно одинаково успешно написать на PHP, C и ассемблере — формально все

верно, но при этом опускается много важной информации, без которой нельзя принять хорошее решение.

В следующих разделах я подробно сравню самые популярные средства управления конфигурацией и инициализации ресурсов: Terraform, Chef, Puppet, Ansible, SaltStack, CloudFormation и OpenStack Heat. Моя цель — помочь вам определиться с тем, является ли Terraform хорошим выбором. Для этого я объясню, почему моя компания, Gruntwork (www.gruntwork.io), выбрала Terraform в качестве средства IaC и — в каком-то смысле — почему я написал эту книгу9. Как и с любым техническим решением, все сводится к компромиссам и приоритетам. Даже если ваши задачи отличаются от моих, надеюсь, что описанный здесь ход мыслей поможет вам принять собственное решение.

Вам придется выбирать между такими вещами, как:

- управление конфигурацией или инициализация ресурсов;
- изменяемая или неизменяемая инфраструктура;
- процедурный или декларативный язык;
- наличие или отсутствие центрального сервера;
- наличие или отсутствие агента;
- большое или маленькое сообщество;
- зрелость или новизна.

Или же совместно использовать несколько инструментов.

Управление конфигурацией или инициализация ресурсов?

Как упоминалось ранее, Chef, Puppet, Ansible и SaltStack управляют конфигурацией, тогда как CloudFormation, Terraform и OpenStack Heat инициализируют ресурсы. Это не совсем четкое разделение, так как средства управления конфигурацией обычно в какой-то степени поддерживают инициализацию ресурсов (например, вы можете развернуть сервер с помощью Ansible), а средства инициализации ресурсов занимаются какого-то рода конфигурацией (скажем, на каждом сервере, инициализированном с помощью Terraform, можно запускать конфигурационные скрипты). Поэтому следует выбирать тот инструмент, который лучше всего подходит для вашего случая10.

В частности, если вы используете средство шаблонизации серверов, такое как Docker или Packer, оно уже покрывает большинство ваших нужд, связанных с управлением конфигурацией. После создания образа из Dockerfile или шаблона Packer вам остается лишь инициализировать для него инфраструктуру, здесь лучше всего подходят средства инициализации ресурсов.

Тем не менее, если вы не применяете инструменты для шаблонизации серверов, хорошей альтернативой будет связка из средств управления конфигурацией и инициализации ресурсов. Например, вы можете инициализировать серверы с помощью Terraform и затем запустить Chef, чтобы сконфигурировать каждый из них.

Выбор между изменяемой и неизменяемой инфраструктурой

Обычно в средствах управления конфигурацией, таких как Chef, Puppet, Ansible и SaltStack, по умолчанию применяется парадигма изменяемой инфраструктуры. Например, если приказать Chef установить новую версию OpenSSL, процесс обновления ПО запустится на существующем сервере, где и произойдут все изменения. Со временем обновления накапливаются, а вместе с ними и история изменений сервера. В итоге у каждого сервера появляются небольшие отличия по сравнению со всеми остальными серверами, что приводит к неочевидным ошибкам в конфигурации, которые трудно диагностировать и воспроизводить (та же проблема с дрейфом конфигурации, возникающая при ручном управлении серверами, хотя благодаря средствам управления конфигурацией у нее куда менее серьезные последствия). Их сложно обнаружить даже с использованием автоматических тестов. Измененная конфигурация может нормально работать в ходе тестирования и при этом вести себя совсем иначе на боевом сервере, который, в отличие от тестовой среды, месяцами накапливал обновления.

Если вы применяете средства инициализации ресурсов наподобие Terraform для развертывания системных образов Docker или Packer, большинство изменений будет заключаться в создании совершенно новых серверов. Например, чтобы развернуть новую версию OpenSSL, вы включаете ее в новый образ Packer, который развертывается на группе новых узлов, а затем удаляете старые узлы. Поскольку при каждом развертывании используются неизменяемые образы и свежие серверы, этот подход уменьшает вероятность дрейфа конфигурации, упрощает отслеживание того, какое ПО установлено на каждом сервере, и позволяет легко развернуть любую предыдущую версию ПО (любой предыдущий образ) в любой момент. Это также повышает эффективность вашего автоматического тестирования, поскольку неизменяемый образ, прошедший проверку в тестовой среде, скорее всего, будет вести себя аналогично и в промышленных условиях.

Конечно, с помощью средств управления конфигурацией можно выполнять и изменяемые развертывания, но для них такой подход не характерен; в то же время для средств инициализации ресурсов он является

вполне естественным. Стоит также отметить, что у неизменяемого подхода есть и недостатки. Например, даже в случае тривиального изменения придется заново собрать образ из шаблона сервера и снова развернуть его на всех ваших узлах, что займет много времени. Более того, неизменяемость сохраняется только до запуска образа. Как только сервер загрузится, он начнет производить запись на жесткий диск и испытывать дрейф конфигурации в той или иной степени (хотя это можно минимизировать, если делать частые развертывания).

Выбор между процедурными и декларативными языками

Chef и Ansible поощряют *процедурный* стиль — когда код пошагово описывает, как достичь желаемого конечного состояния. А вот Terraform, CloudFormation, SaltStack, Puppet и Open Stack Heat исповедуют более *декларативный подход*: вы описываете в своем коде нужное вам конечное состояние, а средства IaC сами разбираются с тем, как его достичь.

Чтобы продемонстрировать это различие, рассмотрим пример. Представьте, что вам нужно развернуть десять серверов (*экземпляров, или инстансов,EC2* в терминологии AWS) для выполнения AMI с идентификатором ami-0c55b159cbfafe1f0 (Ubuntu 18.04). Так выглядит шаблон Ansible, который делает это в процедурном стиле:

```
- ec2:
    count: 10
    image: ami-0c55b159cbfafe1f0
    instance_type: t2.micro
```

А вот упрощенный пример конфигурации Terraform, который делает то же самое, используя декларативный подход:

```
resource "aws_instance" "example" {
  count         = 10
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

На первый взгляд подходы похожи, и если выполнить их с помощью Ansible или Terraform, получатся схожие результаты. Но самое интересное начинается тогда, когда нужно что-то поменять.

Представьте, что у вас повысилась нагрузка и вы хотите увеличить количество серверов до 15. В случае с Ansible написанный ранее процедурный код становится бесполезным; если вы просто запустите его снова, поменяв значение на 15, у вас будет развернуто 15 новых серверов, что в сумме даст 25! Таким образом, чтобы добавить пять новых серверов,

вам нужно написать совершенно новый процедурный скрипт с учетом того, что у вас уже развернуто:

```
- ec2:
    count: 5
    image: ami-0c55b159cbfafe1f0
    instance_type: t2.micro
```

В случае с декларативным кодом нужно лишь описать желаемое конечное состояние, а Terraform разберется с тем, как этого достичь, учитывая любые изменения, сделанные в прошлом. Таким образом, чтобы развернуть еще пять серверов, вам достаточно вернуться к той же конфигурации Terraform и поменять поле count с 10 на 15:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Если вы примените эту конфигурацию, Terraform поймет, что у вас уже есть десять серверов и нужно создать еще пять. Еще до применения конфигурации можно воспользоваться командой Terraform plan, чтобы увидеть, какие изменения будут внесены:

```
$ terraform plan
# aws_instance.example[11] will be created
+ resource "aws_instance" "example" {
    + ami           = "ami-0c55b159cbfafe1f0"
    + instance_type  = "t2.micro"
    + (...)
  }
# aws_instance.example[12] will be created
+ resource "aws_instance" "example" {
    + ami           = "ami-0c55b159cbfafe1f0"
    + instance_type  = "t2.micro"
    + (...)
  }
# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
    + ami           = "ami-0c55b159cbfafe1f0"
    + instance_type  = "t2.micro"
    + (...)
  }
# aws_instance.example[14] will be created
```

```
+ resource "aws_instance" "example" {
    + ami           = "ami-0c55b159cbfafe1f0"
    + instance_type  = "t2.micro"
    + (...)
  }
Plan: 5 to add, 0 to change, 0 to destroy.
```

А если вы хотите развернуть другую версию приложения, например AMI с идентификатором ami-02bcbb802e03574ba? В случае с процедурным подходом оба шаблона Ansible, которые вы уже написали, снова становятся бесполезными. Поэтому необходим еще один шаблон, чтобы отследить те 10 (или уже 15?) серверов, которые вы развернули ранее, и тщательно обновить каждый из них до новой версии. Если использовать декларативный подход, предлагаемый Terraform, достаточно снова вернуться к тому же конфигурационному файлу и просто поменять параметр ami на ami-02bcbb802e03574ba:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-02bcbb802e03574ba"
  instance_type = "t2.micro"
}
```

Естественно, это упрощенные примеры. Ansible позволяет использовать теги для поиска имеющихся серверов EC2, прежде чем добавлять новые (скажем, с помощью параметров instance_tags и count_tag). Однако ручная организация такого рода логики для каждого ресурса, которым вы управляете с применением Ansible, с учетом истории его изменений может оказаться на удивление сложной. Вам придется искать существующие серверы не только по тегам, но также по версии образа и зоне доступности. Из этого вытекают две основные проблемы с процедурными средствами IaC.

•*Процедурный код не полностью охватывает состояние инфраструктуры*. Даже если вы прочитаете все три предыдущих шаблона Ansible, вы все равно не будете знать, что у вас развернуто. Помимо прочего, вам должен быть известен *порядок*, в котором эти шаблоны были применены. Если применить их не в том порядке, можно получить другую инфраструктуру, и этого нельзя увидеть по одной лишь кодовой базе. Иными словами, чтобы разобраться в коде Ansible или Chef, вам нужно знать историю всех изменений, которые когда-либо произошли.

•*Процедурный код ограничивает повторное использование*. Универсальность процедурного кода всегда ограничена, так как вам приходится самостоятельно учитывать текущее состояние инфраструктуры.

Поскольку оно постоянно меняется, код, который вы использовали неделю назад, может оказаться неактуальным, если состояние инфраструктуры, на которое он был рассчитан, больше не существует. В итоге процедурные кодовые базы со временем разрастаются и усложняются.

Благодаря декларативному подходу Terraform код всегда описывает текущее состояние вашей инфраструктуры. Вы можете с одного взгляда определить, что сейчас развернуто и как оно сконфигурировано, не заботясь об истории изменений или синхронизации. Это также облегчает написание кода, пригодного для повторного использования, поскольку вам не нужно самостоятельно учитывать текущее состояние окружающего мира. Вместо этого вы можете сосредоточиться лишь на описании нужного вам состояния, а Terraform автоматически сообразит, как к нему перейти. Поэтому кодовая база Terraform обычно остается компактной и понятной.

Конечно, у декларативного подхода есть свои недостатки. Отсутствие доступа к полноценному языку программирования сказывается на выразительности. Например, некоторые виды изменения инфраструктуры, такие как развертывание без простоя, сложно (но реально, как вы увидите в главе 5) выразить в чисто декларативном стиле. Аналогично ограниченные средства описания «логики» (такие как условные выражения и циклы) делают непростым написание универсального кода, который можно применять повторно. К счастью, Terraform предоставляет ряд мощных примитивов: входные и выходные переменные, модули, create_before_destroy, count, тернарный синтаксис и встроенные функции. Все это позволяет писать чистый, конфигурируемый, модульный код даже на декларативном языке. Мы вернемся к этим темам в главах 4 и 5.

Наличие или отсутствие центрального сервера

Chef, Puppet и SaltStack по умолчанию требуют наличия *центрального (master) сервера* для хранения состояния вашей инфраструктуры и распространения обновлений. Каждый раз, когда вы хотите что-то обновить в своей инфраструктуре, вам необходимо использовать клиент (например, утилиту командной строки), чтобы передать новые команды центральному серверу, который выкатывает обновления на все остальные серверы или позволяет им их загружать на регулярной основе.

У центрального сервера несколько преимуществ. Во-первых, это единое централизованное место, где вы можете просматривать и администрировать состояние своей инфраструктуры. У многих средств управления конфигурацией для этого даже есть веб-интерфейс (например, Chef Console, Puppet Enterprise Console), который помогает ориентироваться в происходящем. Во-вторых, некоторые центральные серверы умеют

работать непрерывно, в фоновом режиме, обеспечивая соблюдение вашей конфигурации. Таким образом, если кто-то поменяет состояние узла вручную, центральный сервер может откатить это изменение, тем самым предотвращая дрейф конфигурации.

Однако использование центрального сервера имеет серьезные недостатки.

•*Дополнительная инфраструктура*. Вам нужно развернуть дополнительный сервер или даже кластер дополнительных серверов (для высокой доступности и масштабируемости).

• *Обслуживание*. Центральный сервер нуждается в обслуживании, обновлении, резервном копировании, мониторинге и масштабировании.

•*Безопасность*. Вам нужно сделать так, чтобы клиент мог общаться с центральным сервером, а последний — со всеми остальными серверами. Это обычно требует открытия дополнительных портов и настройки дополнительных систем аутентификации, что увеличивает область потенциальных атак.

У Chef, Puppet и SaltStack есть разного уровня поддержка режимов работы без центральных серверов. Для этого на каждом сервере запускаются их агенты (обычно по расписанию; например, раз в пять минут), которые загружают последние обновления из системы управления версиями (а не из центрального сервера). Это существенно снижает количество вовлеченных компонентов, но, как вы увидите в следующем подразделе, все равно оставляет без ответа ряд вопросов, особенно касательно того, каким образом в этом случае будут инициализироваться серверы и устанавливаться сами агенты.

У Ansible, CloudFormation, Heat и Terraform по умолчанию нет центрального сервера. Или, если быть более точным, некоторые из них работают с центральным сервером, но он уже является частью используемой инфраструктуры, а не каким-то дополнительным компонентом, требующим отдельного внимания. Предположим, Terraform общается с облачными провайдерами через их API, которые в каком-то смысле являются центральными серверами. Вот только им не нужно никакой дополнительной инфраструктуры или механизмов аутентификации (то есть вы просто применяете свои API-ключи). Ansible подключается к каждому серверу напрямую по SSH, поэтому вам не нужны дополнительная инфраструктура или механизмы аутентификации (то есть вы просто используете свои SSH-ключи).

Наличие или отсутствие агентов

Chef, Puppet и SaltStack требуют установки своих *агентов* (вроде Chef Client, Puppet Agent и Salt Minion) на каждый сервер, который вы хотите настраивать. Агент обычно работает в фоне и отвечает за установку последних обновлений конфигурации.

У этого подхода есть несколько недостатков.

•*Требуется предварительная подготовка*. Как изначально происходит инициализация серверов и установка на них агентов? Некоторые средства управления конфигурацией игнорируют этот момент, подразумевая, что об этом за них позаботится какой-то внешний процесс (например, вначале используется Terraform для развертывания кучи серверов с образами AMI, в которых уже установлен агент). У других предусмотрен специальный подготовительный процесс, в ходе которого вы выполняете одноразовые команды для инициализации серверов (с помощью API облачного провайдера) и установки на них агентов (по SSH).

• *Необходимо выполнять обслуживание*. Вам требуется тщательно и регулярно обновлять программное обеспечение агента, чтобы синхронизировать его с центральным сервером (если таковой имеется). Нужно также следить за агентами и перезапускать их, если они выйдут из строя.

• *Следует обеспечить безопасность*. Если ПО агента загружает конфигурацию с центрального сервера (или какого-то другого сервера, если у вас нет центрального), вам придется открыть исходящие порты на каждом узле. Если центральный сервер сам передает конфигурацию агентам, вам нужно будет открыть на каждом узле входящие порты. В любом случае вы должны найти способ аутентификации агента на сервере, с которым он взаимодействует. Все это увеличивает область потенциальных атак.

И снова Chef, Puppet и SaltStack предлагают разного уровня поддержку режимов работы без агента (например, salt-ssh), но все они выглядят так, будто о них вспомнили задним числом, и ни в одном из них не доступен полный набор возможностей по управлению конфигурацией. В связи с этим в реальных условиях стандартный или идиоматический способ использования Chef, Puppet и SaltStack подразумевает наличие агента и, как правило, центрального сервера (рис. 1.7).

Все эти дополнительные компоненты создают много слабых мест в вашей инфраструктуре. Каждый раз, когда вы получаете отчет о сбое в три часа ночи, вам нужно понять, куда закралась ошибка: в код приложения или IaC, а может, в клиент управления конфигурацией, или в центральный (-е) сервер (-ы), или в процесс взаимодействия между центральным (-и) сервером (-ами) и клиентами или остальными серверами, или же…

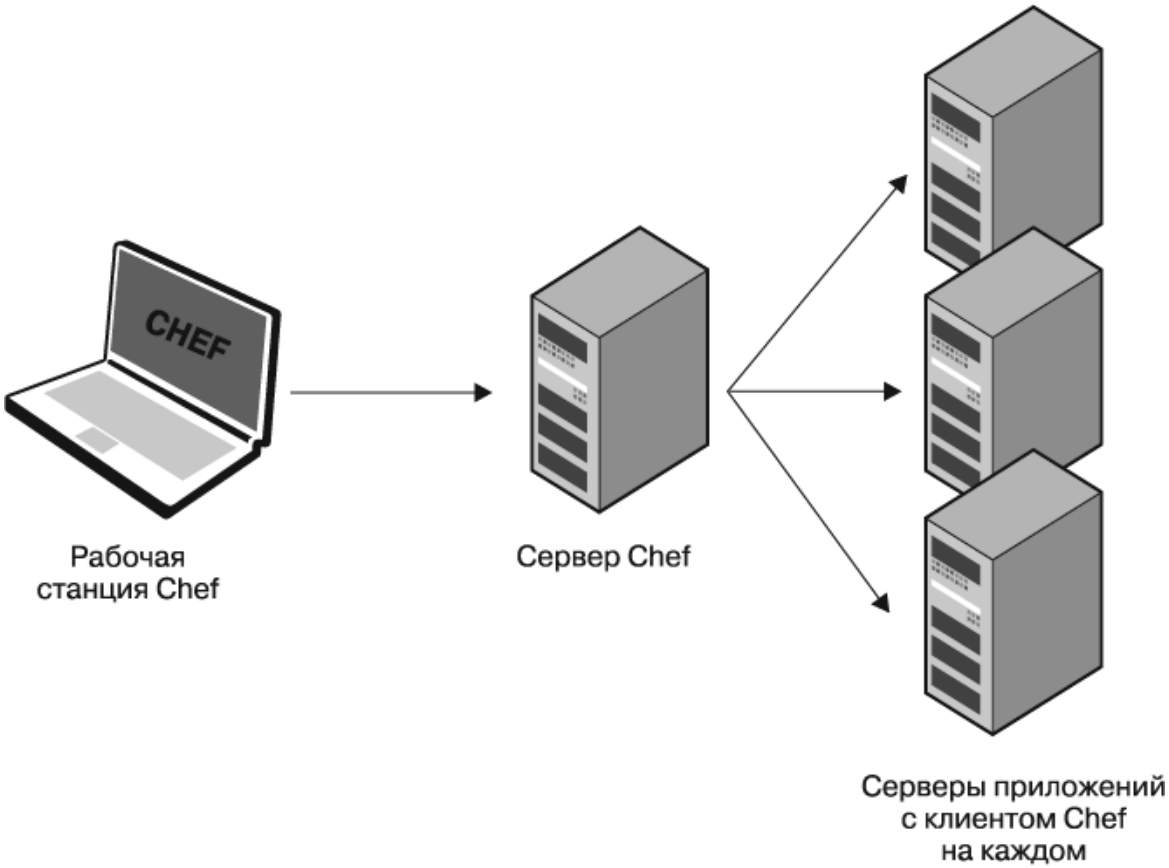


Рис. 1.7. Типичная архитектура Chef, Puppet и SaltStack состоит из множества компонентов. Например, в стандартной конфигурации клиент Chef, запущенный на вашем компьютере, общается с центральным сервером Chef, который развертывает изменения, взаимодействуя с клиентами Chef на всех остальных серверах

Ansible, CloudFormation, Heat и Terraform не требуют установки никаких дополнительных агентов. Или, если быть более точным, некоторым из них нужны агенты, но обычно они уже установлены в рамках используемой вами инфраструктуры. Например, AWS, Azure, Google Cloud и любые другие облачные провайдеры сами занимаются установкой, администрированием и аутентификацией ПО агента на всех своих физических серверах. Вам как пользователю Terraform не нужно об этом беспокоиться: вы просто вводите команды, а агенты облачного провайдера выполняют их для вас на каждом сервере, как показано на рис. 1.8. В случае с Ansible на серверах должен быть запущен демон SSH, который обычно и так есть в большинстве систем.

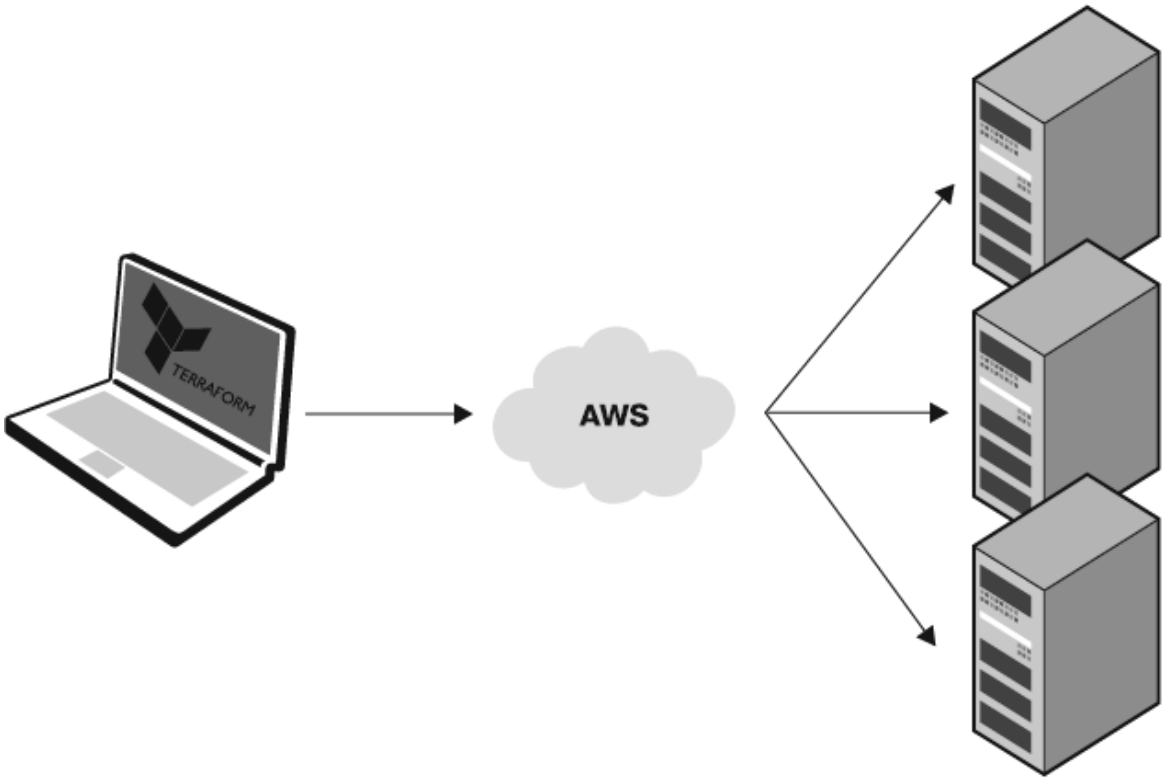


Рис. 1.8. Terraform использует архитектуру, не требующую наличия центральных серверов и агентов. Вам достаточно лишь запустить клиент Terraform, а тот уже позаботится обо всем остальном, применяя API облачных провайдеров, таких как AWS

Размер сообщества

Вместе с технологией вы выбираете и сообщество. Во многих случаях впечатления от использования проекта могут в большей степени зависеть от экосистемы вокруг него, чем от качества самой технологии. Сообщество определяет то, сколько людей помогают проекту, сколько у него подключаемых модулей и расширений, насколько просто получить помощь в Интернете (например, посредством статей или вопросов на StackOverflow) и проблематично ли нанять того, кто мог бы вам помочь (вроде работника, консультанта или компании технической поддержки).

Сложно провести точное сравнение между разными сообществами, но вы можете заметить некоторые тенденции, используя поисковую систему. В табл. 1.1 сравниваются популярные средства IaC с использованием данных, которые я собрал в мае 2019 года. Здесь учитывается, имеет ли инструмент открытый исходный код, с какими провайдерами он совместим, общее количество участников проекта и звезд в GitHub, сколько фиксаций кода и активных заявок было с середины апреля до середины мая, сколько открытых библиотек доступно для этого инструмента, количество вопросов

о нем на StackOverflow и в скольких вакансиях на Indeed.com он упоминается11.

Естественно, это неидеальное сравнение равнозначных показателей. Например, некоторые инструменты имеют больше одного репозитория, а некоторые используют другие методы для отслеживания ошибок и вопросов. Поиск вакансий по таким общеупотребимым словам, как chef или puppet, нельзя считать надежным. В 2017 году код провайдеров Terraform был разделен по отдельным репозиториям, поэтому оценка активности лишь по основному репозиторию будет крайне заниженной (минимум в десять раз).

Таблица 1.1. Сравнение сообществ IaC12131415161718192021222 3

Инструмент	**Код**	**Облака**	**Участники**	**Звезды**	**Фиксации (30 дней)**	**Заявки (30 дней)**	**Библиотеки**	**StackOverflow**	**Вакансии**
Chef	Откр.	Все	562	5794	435	86	38322	5982	43783
Puppet	Откр.	Все	515	5299	94	3144	61105		

3585
42006
Ansible
Откр.
Все
4386
37 161
506
523
20 6777
11 746
8787
SaltStack
Откр.
Все
2237
9901
608
441
3188
1062
1622
CloudFormation
Закр.
AWS
?
?
?
?
3779
3315
3218
Heat
Откр.
Все
361
349
12
60010
011

88
220112
Terraform
Откр.
Все
1261
16 837
173
204
146213
2730
3641

Тем не менее некоторые тенденции очевидны. Во-первых, все средства IaC в этом сравнении имеют открытый исходный код и совместимы со многими облачными провайдерами; исключение составляет проект с закрытым исходным кодом CloudFormation, который работает только с AWS. Во-вторых, в плане популярности лидирует проект Ansible, за которым с небольшим отставанием следуют Salt и Terraform.

Еще одна интересная тенденция — все эти цифры поменялись с момента выхода первого издания. В табл. 1.2 показано относительное изменение каждого показателя по сравнению с той информацией, которую я собрал в сентябре 2016 года.

Таблица 1.2. Как изменились сообщества IaC с сентября 2016 по май 2019 года

Инструмент
Код
Облака
Участники
Звезды
Фиксации (30 дней)
Заявки (30 дней)
Библиотеки
StackOverflow
Вакансии
Chef
Откр.
Все
+18 %
+31 %

+139 %
+48 %
+26 %
+43 %
–22 %
Puppet
Откр.
Все
+19 %
+27 %
+19 %
+42 %
+38 %
+36 %
–19 %
Ansible
Откр.
Все
+195 %
+97 %
+49 %
+66 %
+157 %
+223 %
+125 %
SaltStack
Откр.
Все
+40 %
+44 %
+79 %
+27 %
+33 %
+73 %
+257 %
CloudFormation
Закр.
AWS
?
?

?
?
+57 %
+441 %
+249 %
Heat
Откр.
Все
+28 %
+23 %
–85 %
+1566 %
0
+69 %
+2957 %
Terraform
Откр.
Все
+93 %
+194 %
–61 %
–58 %
+3555 %
+1984 %
+8288 %

Это неидеальные данные, но их достаточно, чтобы заметить четкую тенденцию: Terraform и Ansible испытывают взрывной рост. Увеличение количества участников, звезд, открытых библиотек, вопросов на StackOverflow и вакансий просто зашкаливает24. Сегодня оба инструмента имеют большие и активные сообщества, которые, судя по приведенным выше тенденциям, продолжат расти.

Выбор между зрелостью и новизной

Еще один ключевой фактор при выборе любой технологии — ее зрелость.

В табл. 1.3 приводятся даты выпуска начальной версии каждого инструмента IaC и их версии на данный момент (по состоянию на май 2019 года).

Таблица 1.3. Сравнение инструментов IaC в плане зрелости по состоянию на май 2019 года

Инструмент	**Начальный выпуск**	**Текущая версия**
Puppet	2005	6.12.0
Chef	2009	13.1.58
CloudFormation	2011	???
SaltStack	2011	3000
Ansible	2012	2.9.5
Heat	2012	13.0.0
Terraform	2014	0.12.21

Здесь сравниваются не совсем равнозначные вещи, поскольку разные инструменты используют разные методы управления версиями, но некоторые тенденции бросаются в глаза. Terraform, безусловно, является самым молодым инструментом IaC в этом сравнении. Он все еще не достиг версии 1.0.0, поэтому не ожидайте гарантий стабильного или обратно совместимого API, и программные ошибки встречаются относительно часто (хотя большинство из них незначительные). Это самое слабое место Terraform: несмотря на достижение огромной популярности за короткое время и применение передовых технологий, это менее зрелый проект по сравнению с некоторыми другими средствами IaC.

Совместное использование нескольких инструментов

Я сравнивал разные инструменты IaC на протяжении всей этой главы, но в реальности при построении своей инфраструктуры вам, скорее всего,

придется работать сразу с несколькими из них. У каждого представленного здесь инструмента есть свои сильные и слабые стороны, и вы должны выбрать подходящий для ваших задач вариант.

Далее описываются три распространенные комбинации, которые хорошо себя проявили в ряде компаний.

Инициализация ресурсов плюс управление конфигурацией

Пример: Terraform и Ansible. Terraform используется для развертывания всей внутренней инфраструктуры, включая топологию сети (то есть виртуальные частные облака (virtual private cloud, или VPC), подсети, таблицы маршрутизации), хранилища данных (MySQL, Redis), балансировщики нагрузки и серверы. Ansible берет на себя развертывание ваших приложений поверх этих серверов, как показано на рис. 1.9.

Рис. 1.9. Совместное использование Terraform и Ansible

Этот подход позволяет быстро приступить к работе, поскольку вам не нужна никакая дополнительная инфраструктура (Terraform и Ansible — сугубо клиентские приложения) и оба инструмента можно интегрировать множеством разных способов (например, Terraform назначает вашим серверам специальные теги, которые Ansible использует для поиска и конфигурации этих серверов). Основной недостаток состоит в том, что применение Ansible обычно подразумевает много процедурного кода и изменяемые серверы, поэтому расширение кодовой базы, инфраструктуры и вашей команды может осложнить обслуживание.

Инициализация ресурсов плюс шаблонизация серверов

Пример: Terraform и Packer. Packer используется для упаковки ваших приложений в виде образов ВМ. Затем Terraform развертывает: а) серверы с помощью этих образов; б) всю остальную инфраструктуру, включая топологию сети (то есть VPC, подсети, таблицы маршрутизации), хранилища данных (как MySQL, Redis) и балансировщики нагрузки. Это проиллюстрировано на рис. 1.10.

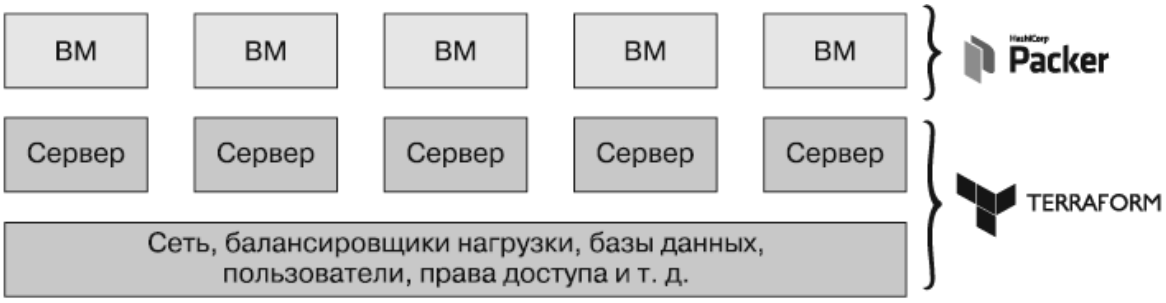


Рис. 1.10. Совместное применение Terraform и Packer

Этот подход тоже позволяет быстро приступить к работе, так как вам не нужна никакая дополнительная инфраструктура (Terraform и Packer являются сугубо клиентскими приложениями). Позже в этой книге вы сможете вдоволь попрактиковаться в развертывании образов ВМ с помощью Terraform. Кроме того, вы получаете неизменяемую инфраструктуру, что упростит ее обслуживание. Однако у этой комбинации есть два существенных недостатка. Во-первых, на сборку и развертывание образов ВМ может уходить много времени, что замедлит выпуск обновлений. Во-вторых, как вы увидите в последующих главах, Terraform поддерживает ограниченный набор стратегий развертывания (например, сам по себе этот инструмент не позволяет реализовать «сине-зеленые» обновления), поэтому вам придется либо написать много сложных скриптов, либо обратиться к средствам оркестрации, как это будет показано далее.

Инициализация ресурсов плюс шаблонизация серверов плюс оркестрация

Пример: Terraform, Packer, Docker и Kubernetes. Packer используется для создания образов ВМ с установленными Docker и Kubernetes. Затем Terraform развертывает: а) серверы с помощью этих образов; б) всю остальную инфраструктуру, включая топологию сети (VPC, подсети, таблицы маршрутизации), хранилища данных (MySQL, Redis) и балансировщики нагрузки. Когда серверы загрузятся, они сформируют кластер Kubernetes, которым вы будете запускать и администрировать свои приложения в виде контейнеров Docker (рис. 1.11).

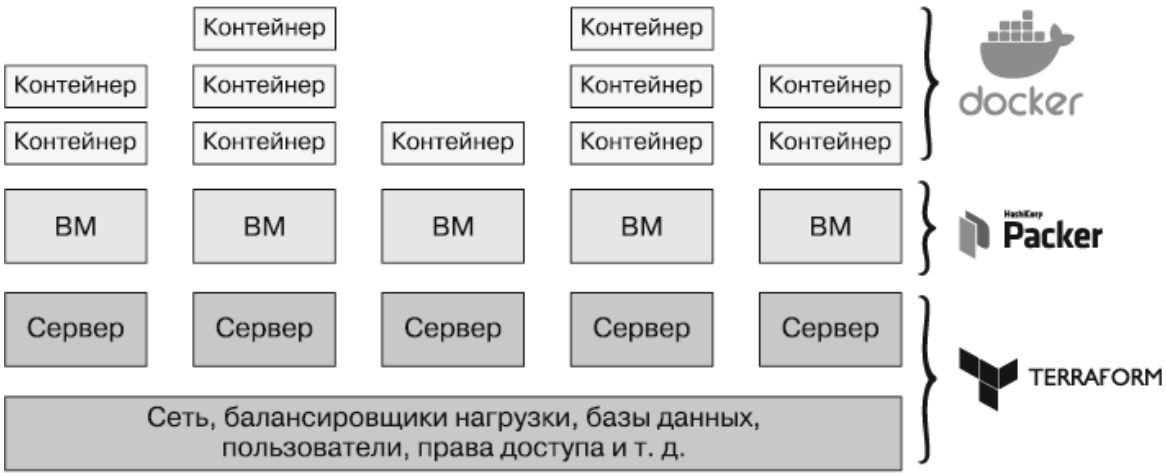


Рис. 1.11. Совместное использование Terraform, Packer, Docker и Kubernetes

Преимущество этого подхода в том, что образы Docker собираются довольно быстро, поэтому их можно запускать и тестировать на локальном компьютере. Вы также можете использовать богатые возможности Kubernetes, включая различные стратегии развертывания, автовосстановление, автомасштабирование и т. д. Недостатки связаны с повышением сложности — как с точки зрения инфраструктуры (развертывание кластеров Kubernetes является сложным и дорогим, хотя большинство основных облачных провайдеров теперь предоставляют управляемые сервисы Kubernetes, на которые можно возложить часть этой работы), так и в смысле дополнительных слоев абстракции (Kubernetes, Docker, Packer), которые необходимо изучать, обслуживать и отлаживать.

Резюме

Ниже показана сводная таблица самых популярных средств IaC (табл. 1.4). Заметьте, что здесь приводятся *стандартные* и *самые распространенные* способы их применения. Как уже упоминалось ранее, данные инструменты достаточно гибкие, чтобы их можно было применять и в других конфигурациях (например, Chef можно запускать без центрального сервера, а Salt поддерживает неизменяемую инфраструктуру).

Таблица 1.4. Сравнение наиболее распространенных способов использования самых популярных средств IaC

Инструмент

Код

Облака

Тип

Инфраструктура

Язык	**Агент**	**Вед. сервер**	**Сообщество**	**Зрелость**
Chef	Откр.	Все	Упр. конф.	Изменяемая
Процедурный	Есть	Есть	Большое	Высокая
Puppet	Откр.	Все	Упр. конф.	Изменяемая
Декларативный	Есть	Есть	Большое	Высокая
Ansible	Откр.	Все	Упр. конф.	Изменяемая
Процедурный	Нет	Нет	Огромное	Средняя
SaltStack	Откр.	Все	Упр. конф.	Изменяемая

Декларативный
Есть
Есть
Большое
Средняя
CloudFormation
Закр.
AWS
Иниц. рес.
Неизменяемая
Декларативный
Нет
Нет
Маленькое
Средняя
Heat
Откр.
Все
Иниц. рес.
Неизменяемая
Декларативный
Нет
Нет
Маленькое
Низкая
Terraform
Откр.
Все
Иниц. рес.
Неизменяемая
Декларативный
Нет
Нет
Огромное
Низкая

В компании Gruntwork нам нужно было открытое, не привязанное к конкретному облаку средство инициализации ресурсов, которое поддерживает неизменяемую инфраструктуру, декларативный язык, архитектуру, не требующую наличия центральных серверов и агентов, и имеет большое сообщество и зрелую кодовую базу. Как показывает

табл. 1.4, проект Terraform хоть и неидеальный, но именно он больше всего удовлетворяет нашим критериям.

Подходит ли он под ваши критерии? Если да, то переходите к главе 2, в которой вы научитесь его использовать.

3 Источник: *Kim G., Humble J., Debois P., WillisJ.* TheDevOpsHandbook: HowtoCreateWorld-ClassAgility, Reliability & SecurityinTechnologyOrganizations. — ITRevolutionPress, 2016.

4 Идемпотентность — свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом.

5 В современных операционных системах код выполняется в одном из двух «пространств»: в пространстве ядра и пространстве пользователя. Код, который работает в пространстве ядра, имеет прямой и неограниченный доступ ко всему оборудованию. На него не распространяются ограничения безопасности: вы можете выполнять любые процессорные инструкции, обращаться к любому участку жесткого диска, записывать в любой адрес памяти. При этом сбой в пространстве ядра обычно приводит к сбою всего компьютера. В связи с этим оно обычно отводится для самых низкоуровневых и доверенных функций ОС (которые называют ядром). Код в пользовательском пространстве не имеет непосредственного доступа к аппаратному обеспечению и вместо этого должен использовать API, предоставляемые ядром ОС. Эти интерфейсы могут накладывать ограничения безопасности (скажем, права доступа) и локализовать сбои в пользовательских приложениях, поэтому весь прикладной код работает в пользовательском пространстве.

6 Той изоляции, которую предоставляют контейнеры, как правило, достаточно для выполнения собственного кода. Но если нужно выполнять сторонний код (например, если вы хотите создать свое публичное облако), который может намеренно предпринять вредоносные действия, вам пригодятся повышенные гарантии изоляции ВМ.

7 В качестве альтернативы bash для настройки образов в Packer можно использовать средства управления конфигурацией, такие как Ansible и Chef.

8 В качестве примера можно взять библиотеку IaC от компании Gruntwork по адресу bit.ly/2H3Y7yT.

9 Docker, Packer и Kubernetes не участвуют в сравнении, так как их можно использовать в сочетании с любыми средствами управления конфигурацией или инициализации ресурсов.

10 На сегодня различие между управлением конфигурацией и инициализацией ресурсов еще менее очевидно, поскольку ряд популярных средств, входящих в первую категорию, таких как Chef Provisioning

(https://github.com/chef-boneyard/chef-provisioning) и Puppet AWS Module (bit.ly/2YIQIuN), постепенно улучшил поддержку инициализации ресурсов.

11 Большая часть этих данных, включая количество участников, звезд, изменений и заявок, была взята из репозиториев открытого исходного кода и систем отслеживания ошибок (в основном в GitHub) каждого отдельного инструмента. Поскольку проект CloudFormation имеет закрытый исходный код, некоторые из этих сведений для него недоступны.

12 Это количество руководств в Chef Supermarket (bit.ly/2MNXWuS).

13 Чтобы избежать ложных срабатываний для chef, я искал по chef devops.

14 На основе учетной записи Puppet Labs в JIRA (bit.ly/2ZN3ppq).

15 Это количество модулей в Puppet Forge (forge.puppet.com).

16 Чтобы избежать ложных срабатываний для puppet, я искал по puppet devops.

17 Это количество универсальных ролей в Ansible Galaxy (galaxy.ansible.com).

18 Это количество формул в учетной записи Salt Stack Formulas в GitHub (github.com/saltstack-formulas).

19 Это количество шаблонов в учетной записи awslabs в GitHub (github.com/awslabs).

20 На основе системы отслеживания ошибок OpenStack (bit.ly/31jeDCH).

21 Мне не удалось найти ни одной коллекции шаблонов Heat, подготовленной сообществом.

22 Чтобы избежать ложных срабатываний для heat, я искал по openstack.

23 Это количество модулей в реестре Terraform (registry.terraform.io).

24 Снижение количества фиксаций кода и заявок у Terraform вызвано лишь тем фактом, что я измеряю только основной репозиторий проекта, тогда как в 2017 году код всех провайдеров был распределен по отдельным репозиториям. Поэтому здесь не учитывается огромная доля активности в репозиториях более чем 100 провайдеров.

2. Приступаем к работе с Terraform

В этой главе вы научитесь основам применения Terraform. Этот инструмент прост в изучении, поэтому за следующие 40 страниц вы пройдете путь от выполнения ваших первых команд до развертывания кластера серверов с балансировщиком нагрузки, который распределяет между ними трафик. Такая инфраструктура будет хорошей отправной точкой для запуска масштабируемых высокодоступных веб-сервисов. В следующих главах мы продолжим улучшать этот пример.

Terraform умеет инициализировать инфраструктуру как в публичных облаках, вроде Amazon Web Services (AWS), Azure, Google Cloud и DigitalOcean, так и в частных облачных платформах и системах виртуализации вроде OpenStack и VMWare. Практически во всех примерах кода в книге будет использоваться AWS. Это хороший выбор для изучения Terraform по следующим причинам.

• AWS, вне всяких сомнений, является самым популярным провайдером облачной инфраструктуры. Его доля на рынке облачных решений составляет 45 %, что больше, чем у трех ближайших конкурентов (Microsoft, Google и IBM), вместе взятых (http://bit.ly/2kWCuCm).

• AWS предоставляет широчайший спектр надежных и масштабируемых сервисов с облачным размещением, включая Amazon Elastic Compute Cloud (Amazon EC2), который можно использовать для развертывания виртуальных серверов, Auto Scaling Groups (ASGs), упрощающий управление кластером виртуальных серверов, и Elastic Load Balancers (ELBs), с помощью которого можно распределять трафик между виртуальными серверами кластера25.

• В первый год использования AWS предлагает щедрый бесплатный тариф (https://aws.amazon.com/free/), который позволяет выполнить все эти примеры без денежных затрат. Если вы уже исчерпали свои бесплатные кредиты, работа с примерами из этой книги будет стоить не дороже нескольких долларов.

Не нужно волноваться, если вы прежде не использовали AWS или Terraform. Этот учебник подойдет для новичков в обеих технологиях. Я проведу вас через такие этапы, как:

• подготовка вашей учетной записи в AWS;
• установка Terraform;
• развертывание одного сервера;
• развертывание одного веб-сервера;
• развертывание конфигурируемого веб-сервера;
• развертывание кластера веб-серверов;
• развертывание балансировщика нагрузки;
• удаление ненужных ресурсов.

Пример кода

Напоминаю, что все примеры кода из этой книги доступны по адресу github.com/brikis98/terraform-up-and-running-code.

Подготовка вашей учетной записи в AWS

Если у вас нет учетной записи в AWS, пройдите на страницу aws.amazon.com и зарегистрируйтесь. Сразу после регистрации вы входите в систему в качестве *корневого пользователя*. Эта учетная запись позволяет делать что угодно, поэтому с точки зрения безопасности ее лучше не использовать регулярно. Она вам будет нужна *только* для создания других пользовательских учетных записей с ограниченными правами, на одну из которых вы должны немедленно переключиться26.

Чтобы создать более ограниченную учетную запись, следует использовать сервис Identity and Access Management (IAM). IAM — это то место, где происходит управление учетными записями пользователей и их правами. Чтобы создать нового *пользователя IAM*, перейдите в консоль IAM (https://amzn.to/33fM2jf), щелкните на ссылке Users (Пользователи) и затем нажмите кнопку Create New Users (Создать новых пользователей). Введите имя пользователя и убедитесь, что флажок Generate an access key for each user (Сгенерировать ключ доступа для каждого пользователя) установлен, как показано на рис. 2.1. Имейте в виду, что AWS вносит косметические изменения в свою веб-консоль, поэтому на момент чтения этой книги страницы IAM могут немного отличаться.

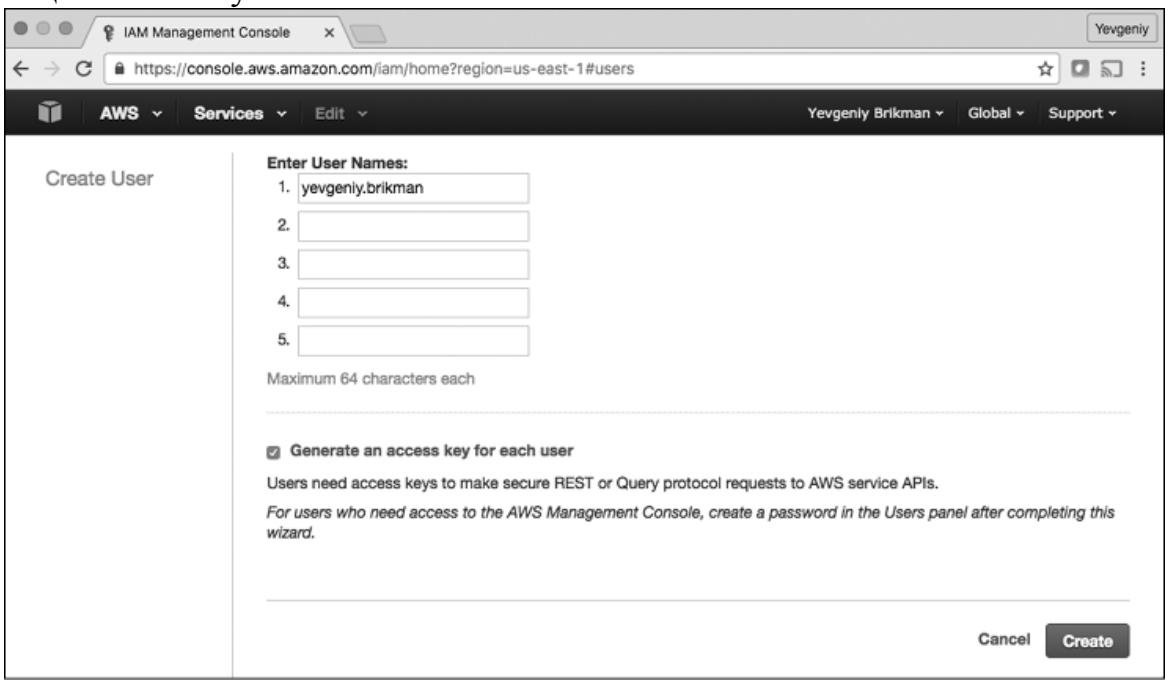


Рис. 2.1. Создание нового пользователя IAM

Нажмите кнопку Create (Создать). AWS покажет вам учетные данные доступа этого пользователя, которые, как видно на рис. 2.2, состоят из *ID ключа доступа* (Access Key ID) и *секретного ключа доступа* (Secret Access Key). Их следует немедленно сохранить, поскольку их больше никогда не покажут, а они в этом руководстве еще понадобятся. Помните, что эти данные дают доступ к вашей учетной записи в AWS, поэтому храните их в безопасном месте (например, в диспетчере паролей, таком как 1Password, LastPass или OS X Keychain) и никогда никому не давайте.

Сохранив свои учетные данные, нажмите кнопку Close (Закрыть). Таким образом вы перейдете к списку пользователей IAM. Щелкните на имени пользователя, которого только что создали, и выберите вкладку Permissions (Права доступа). Новые пользователи IAM по умолчанию лишены всяких прав и, следовательно, не могут ничего делать в рамках учетной записи AWS.

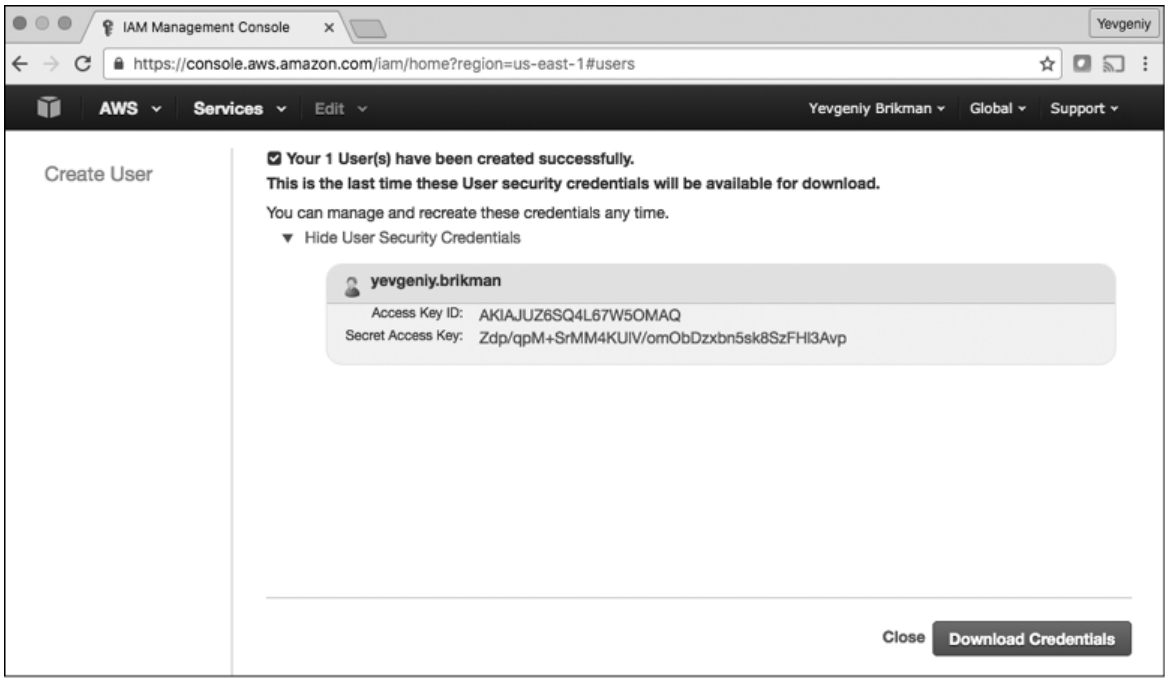


Рис. 2.2. Храните свои учетные данные AWS в надежном месте. Никому их не показывайте (не волнуйтесь, те, что на снимке экрана, — ненастоящие)

Чтобы выдать пользователю IAM какие-то права, вы должны связать его учетную запись с одной или несколькими политиками IAM. *Политика IAM* — это документ в формате JSON, который определяет, что пользователю позволено, а что — нет. Вы можете создавать свои собственные политики или обойтись уже готовыми, которые называются *управляемыми политиками*27.

Для выполнения примеров из этой книги вам нужно назначить своему пользователю IAM следующие управляемые политики (рис. 2.3).

•AmazonEC2FullAccess — требуется для этой главы.

• AmazonS3FullAccess — требуется для главы 3.

• AmazonDynamoDBFullAccess — требуется для главы 3.

• AmazonRDSFullAccess — требуется для главы 3.

• CloudWatchFullAccess — требуется для главы 5.

• IAMFullAccess — требуется для главы 5.

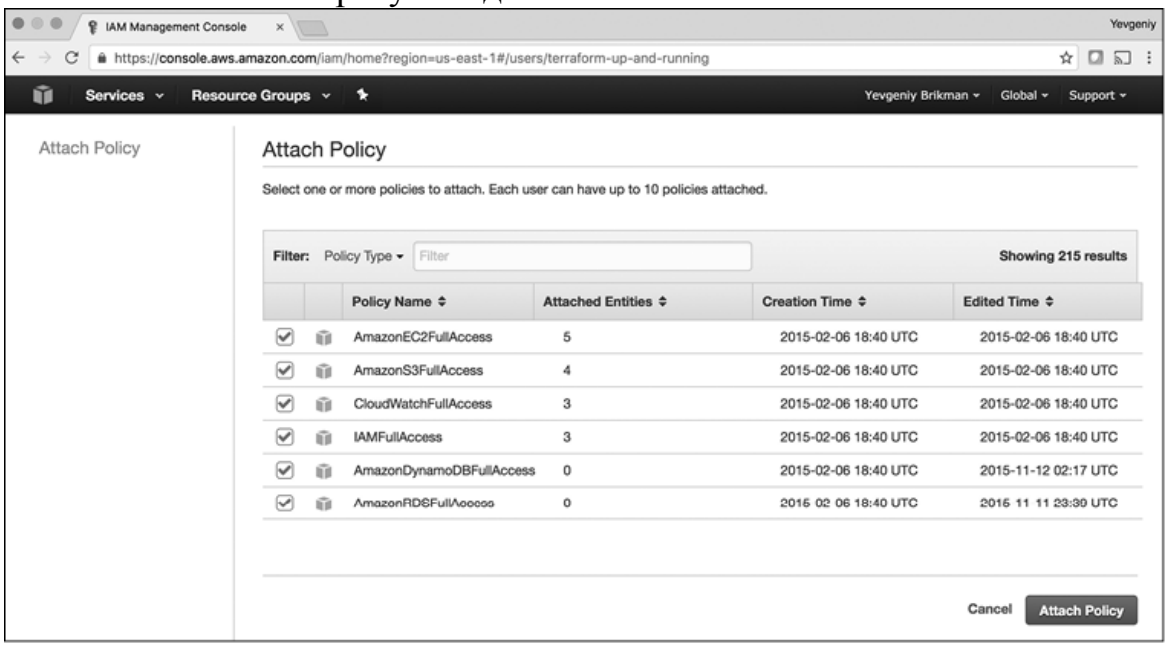


Рис. 2.3. Назначение нескольких управляемых политик IAM вашему новому пользователю IAM

Замечание о виртуальных частных облаках по умолчанию

Если вы используете существующую учетную запись AWS, у нее уже должно быть облако VPC по умолчанию. VPC (Virtual Private Cloud — виртуальное частное облако) — это изолированная область вашей учетной записи AWS с собственными виртуальной сетью и пространством IP-адресов. Практически любой ресурс AWS развертывается в VPC. Если не указать VPC явно, ресурс будет развернут в VPC по умолчанию, которое является частью всех новых учетных записей AWS. VPC по умолчанию применяется во всех примерах в этой книге, поэтому, если по какой-то

причине вы его удалили, переключитесь на другой регион (у каждого региона свое облако VPC по умолчанию) или создайте новое в веб-консоли AWS (https://amzn.to/31lVUWW). В противном случае вам придется обновить почти все примеры, добавив в них параметр vpc_id или subnet_id, который указывает на пользовательское VPC.

Установка Terraform

Вы можете загрузить Terraform на домашней странице проекта по адресу https://www.terraform.io. Щелкните на ссылке для загрузки, выберите подходящий пакет для своей операционной системы, сохраните ZIP-архив и распакуйте его в папку, в которую хотите установить Terraform. Архив содержит единственный двоичный файл под названием terraform, который следует добавить в переменную среды PATH. Как вариант, попробуйте поискать Terraform в диспетчере пакетов своей ОС; например, в OS X можно выполнить brewinstallterraform.

Чтобы убедиться, что все работает, запустите команду terraform. Вы должны увидеть инструкции по применению:

```
$ terraform
Usage: terraform [-version] [-help] <command> [args]
Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    (...)
```

Чтобы система Terraform могла вносить изменения в вашу учетную запись AWS, нужно прописать в переменных среды AWS_ACCESS_KEY_ID и AWS_SECRET_ACCESS_KEY учетные данные для пользователя IAM, которого вы создали ранее. Например, вот как это можно сделать в терминале Unix/Linux/macOS:

```
$ export AWS_ACCESS_KEY_ID=(your access key id)
$ export AWS_SECRET_ACCESS_KEY=(your secret access key)
```

Имейте в виду, что эти переменные среды действуют только в текущей командной оболочке, поэтому после перезагрузки компьютера или открытия нового окна терминала придется снова их экспортировать.

Способы аутентификации

Помимо переменных среды, Terraform поддерживает те же механизмы аутентификации, что и все утилиты командной строки и SDK для AWS. Таким образом, вы сможете использовать файл $HOME/.aws/credentials, который автоматически генерируется, если запустить AWS CLI с командой configure, или роли IAM, которые можно назначить почти любому ресурсу в AWS. Подробнее об этом — в статье A Comprehensive Guide to Authenticating to AWS on the Command Line по адресу http://bit.ly/2M11muR.

Развертывание одного сервера

Код Terraform пишется на *языке конфигурации HashiCorp* (HashiCorp Configuration Language, или HCL) и хранится в файлах с расширением .tf28. Это декларативный язык, поэтому ваша задача — описать нужную вам инфраструктуру, а Terraform разберется с тем, как ее создать. Terraform умеет создавать инфраструктуру на разнообразных платформах (или *провайдерах* в терминологии проекта), включая AWS, Azure, Google Cloud, DigitalOcean и многие другие.

Код Terraform можно писать в практически любом текстовом редакторе. Если поискать, можно найти подсветку синтаксиса Terraform для большинства редакторов (обратите внимание, что вам, возможно, нужно искать по слову HCL, а не Terraform), включая vim, emacs, Sublime Text, Atom, Visual Studio Code и IntelliJ (у последнего даже есть поддержка рефакторинга, поиска вхождений и перехода к объявлению).

Первым делом при использовании Terraform обычно следует выбрать провайдера (одного или несколько), с которым вы хотите работать. Создайте пустую папку и поместите в нее файл с именем main.tf и следующим содержанием:

```
provider "aws" {
  region = "us-east-2"
}
```

Это говорит Terraform о том, что в качестве провайдера вы собираетесь использовать AWS и хотите развертывать свою инфраструктуру в регионе us-east-2. Вычислительные центры AWS разбросаны по всему миру и сгруппированы по регионам. *Регионы AWS* — это отдельные географические области, такие как us-east-2 (Огайо), eu-west-1 (Ирландия) и ap-southeast-2 (Сидней). Внутри каждой области находится несколько изолированных вычислительных центров, известных как *зоны доступности*: например, us-east-2a, us-east-2b и т. д.

Каждый тип провайдеров поддерживает создание разного вида *ресурсов*, таких как серверы, базы данных и балансировщики нагрузки. Обобщенный синтаксис создания ресурса в Terraform выглядит так:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

PROVIDER — имя провайдера (предположим, aws), TYPE — тип ресурса, создаваемого в этом провайдере (вроде instance), NAME — идентификатор, с помощью которого вы хотите ссылаться на ресурс в коде Terraform (скажем, my_instance), а CONFIG содержит один или несколько *аргументов*, предусмотренных специально для этого ресурса.

Например, чтобы развернуть в AWS один (виртуальный) сервер *(экземпляр EC2)*, укажите в файле main.tf ресурс aws_instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Ресурс aws_instance поддерживает много разных аргументов, но пока что вам нужны только два из них, которые являются обязательными.

•ami. Образ Amazon Machine Image (AMI), который будет запущен на сервере EC2. В AWS Marketplace (https://aws.amazon.com/marketplace/) можно найти платные и бесплатные образы. Вы также можете создать собственный экземпляр AMI, применяя такие инструменты, как Packer (обсуждение образов и шаблонизации серверов ищите в подразделе «Средства шаблонизации серверов» на с. 31). Код, представленный выше, назначает параметру ami идентификатор AMI Ubuntu 18.04 в us-east-2. Этот образ можно использовать бесплатно.

• instance_type. Тип сервера EC2, который нужно запустить. У каждого типа есть свой объем ресурсов процессора, памяти, дискового пространства и сети. Все доступные варианты перечислены на соответствующей странице по адресу https://amzn.to/2H49EOH. В приведенном выше примере используется тип t2.micro, который имеет один виртуальный процессор с 1 Гбайт памяти и входит в бесплатный тариф AWS.

Используйте документацию!

Terraform поддерживает десятки провайдеров, у каждого из которых есть десятки ресурсов, и для каждого ресурса предусмотрены десятки аргументов. Запомнить все это невозможно. При написании кода Terraform следует регулярно сверяться с документацией, чтобы посмотреть, какие ресурсы вам доступны и как их использовать. Например, вот документация для ресурса aws_instance по адресу http://bit.ly/33dmi7g. Несмотря на свой многолетний опыт использования Terraform, я все еще обращаюсь к этой документации по нескольку раз в день!

Откройте терминал, перейдите в папку, в которой вы создали файл main.tf, и выполните команду terraforminit:

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (terraform-providers/aws) 2.10.0...
```

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.aws: version = "~> 2.10"
Terraform has been successfully initialized!
```

Исполняемый файл terraform поддерживает основные команды Terraform, но он не содержит никакого кода для провайдеров (вроде AWS, Azure или GCP). Поэтому, начиная работу с этим инструментом, вы должны выполнить terraforminit, чтобы он мог просканировать ваш код, определить, с какими провайдерами вы работаете, и загрузить для них подходящие модули. По умолчанию код провайдеров загружается в папку .terraform, которая является рабочей папкой Terraform (стоит добавить ее в .gitignore). В последующих главах вы познакомитесь с другими сценариями использования команды init и папки .terraform. А пока что просто знайте, что init необходимо выполнять каждый раз, когда вы начинаете писать новый код Terraform, и это можно делать многократно (это идемпотентная команда).

Теперь, загрузив код провайдера, выполните команду terraformplan:

```
$ terraform plan
(...)
Terraform will perform the following actions:
  # aws_instance.example will be created
```

```
  + resource "aws_instance" "example" {
      + ami                          = "ami-0c55b159cbfafe1f0"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      (...)
    }
  Plan: 1 to add, 0 to change, 0 to destroy.
```

Команда plan позволяет увидеть, что сделает Terraform, без внесения каких-либо изменений. Это хорошая возможность еще раз проверить свой код перед выпуском его во внешний мир. По своему выводу команда plan похожа на утилиту diff, которая является частью Unix, Linux и git: знак плюс (+) помечает все, что будет создано; знак минус (-) — что будет удалено, а то, что помечено тильдой (~), будет изменено. В предыдущем выводе вы можете видеть, что Terraform планирует создать лишь один сервер EC2 и ничего другого — именно то, что нам нужно.

Чтобы инициировать создание сервера, нужно выполнить команду terraformapply:

```
  $ terraform apply
  (...)
  Terraform will perform the following actions:
    # aws_instance.example will be created
    + resource "aws_instance" "example" {
      + ami                          = "ami-0c55b159cbfafe1f0"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
```

```
      + host_id                     = (known after apply)
      + id                          = (known after apply)
      + instance_state              = (known after apply)
      + instance_type               = "t2.micro"
      + ipv6_address_count          = (known after apply)
      + ipv6_addresses              = (known after apply)
      + key_name                    = (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:
```

Вы можете заметить, что команда apply отображает такой же вывод, как и plan, и спрашивает вас, действительно ли вы хотите перейти к осуществлению этого плана. plan — отдельная команда, которая в основном подходит для быстрой проверки и разбора кода (больше об этом — в главе 8), но в большинстве случаев вы будете сразу выполнять команду apply, проверяя ее вывод.

Введите yes и нажмите клавишу Enter, чтобы развернуть сервер EC2:

```
  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
  aws_instance.example: Creating...
  aws_instance.example: Still creating... [10s elapsed]
  aws_instance.example: Still creating... [20s elapsed]
  aws_instance.example: Still creating... [30s elapsed]
  aws_instance.example: Creation complete after 38s [id=i-
07e2a3e006d785906]
  Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Поздравляем, вы развернули сервер EC2 в своей учетной записи AWS, используя Terraform! Чтобы в этом убедиться, перейдите в консоль EC2 (https://amzn.to/2GOFxdI); вы должны увидеть страницу, похожую на ту, что изображена на рис. 2.4.

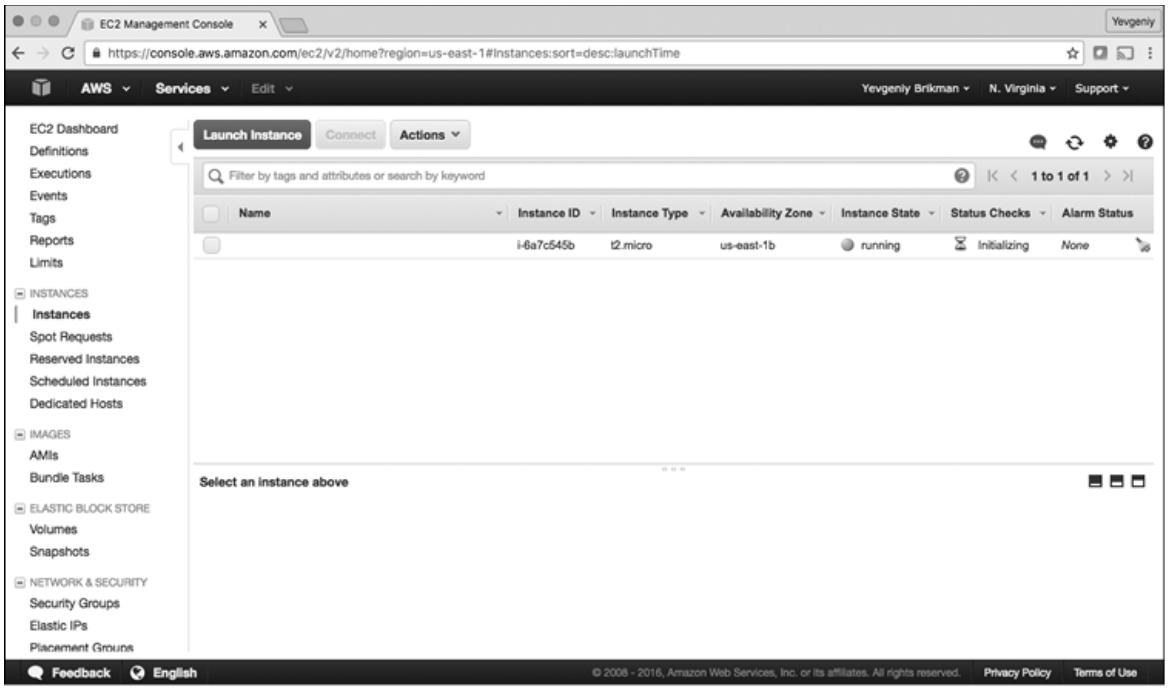


Рис. 2.4. Один сервер EC2

Мы действительно видим наш сервер! Однако нужно признать, что это не самый впечатляющий пример. Сделаем его немного более интересным. Для начала обратите внимание на то, что у нашего сервера EC2 нет имени. Чтобы его указать, добавьте к ресурсу aws_instance параметр tags:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = {
    Name = "terraform-example"
  }
}
```

Чтобы увидеть последствия этих изменений, еще раз выполните команду terraformapply:

```
$ terraform apply
aws_instance.example: Refreshing state...
(...)
Terraform will perform the following actions:
  # aws_instance.example will be updated in-place
  ~ resource "aws_instance" "example" {
      ami                          = "ami-0c55b159cbfafe1f0"
      availability_zone            = "us-east-2b"
      instance_state               = "running"
      (...)
```

```
    + tags = {
      + "Name" = "terraform-example"
    }
    (...)
  }
Plan: 0 to add, 1 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:
```

Система Terraform отслеживает все ресурсы, которые были созданы для этого набора конфигурационных файлов. Поэтому ей известно, что сервер EC2 уже существует (обратите внимание на строчку Refreshingstate… при выполнении команды apply), и она может вывести разницу между тем, что развернуто на данный момент, и тем, что описано в вашем коде (это одно из преимуществ использования декларативного кода по сравнению с процедурным; см. раздел «Сравнение Terraform с другими средствами IaC» на с. 42). В предыдущем сравнении видно, что Terraform хочет создать один тег под названием Name. Это именно то, что вам нужно, поэтому введите yes и нажмите клавишу Enter.

Обновив свою консоль EC2, вы увидите примерно такое окно, как на рис. 2.5.

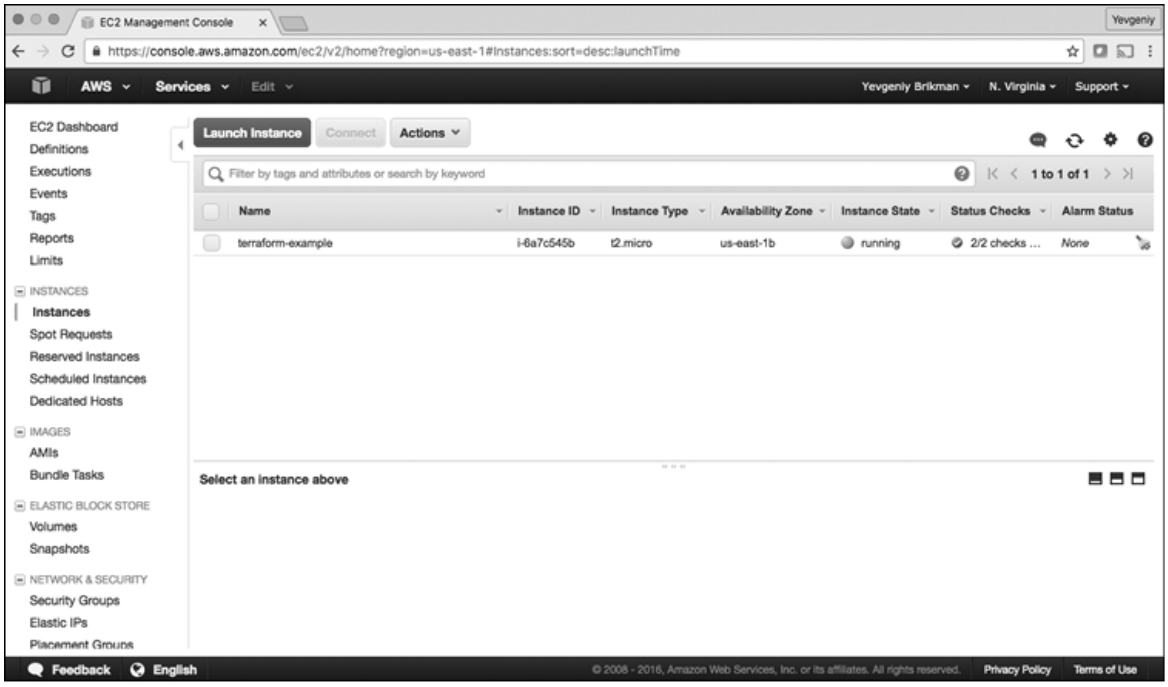


Рис. 2.5. У сервера EC2 теперь есть тег с именем

Итак, у вас теперь есть рабочий код Terraform, стоит сохранить его в системе управления версиями. Это позволит вам делиться им с другими членами команды, отслеживать историю всех изменений инфраструктуры и использовать журнал фиксаций для отладки. Ниже показан пример того, как создать локальный Git-репозиторий и применять его для хранения конфигурационного файла Terraform:

```
git init
git add main.tf
git commit -m "Initial commit"
```

Следует также создать файл под названием .gitignore. Он заставляет Git игнорировать определенные типы файлов, которые вы бы не хотели сохранить по случайности:

```
.terraform
*.tfstate
*.tfstate.backup
```

Файл .gitignore, показанный выше, заставляет Git игнорировать папку .terraform, которую Terraform использует в качестве временной рабочей папки, а также файлы вида *.tfstate, в которых Terraform хранит состояние (в главе 3 вы узнаете, почему файлы состояния нельзя записывать в репозиторий). Файл .gitignore тоже нужно зафиксировать:

```
git add .gitignore
git commit -m "Add a .gitignore file"
```

Чтобы поделиться этим кодом со своими коллегами, вам необходимо создать общий Git-репозиторий, к которому вы все можете обращаться. Для этого можно использовать GitHub. Перейдите на сайт https://github.com, зарегистрируйтесь (если вы этого еще не сделали), создайте новый репозиторий и укажите его в качестве удаленной точки входа под названием origin для своего локального Git-репозитория:

```
git remote add origin git@github.com:<YOUR_USERNAME>/<YOUR_REPO_NAME>.git
```

Теперь, когда вы хотите поделиться своими изменениями с коллегами, можете загрузить изменения *в* репозиторий origin:

```
git push origin master
```

Если вы хотите посмотреть, какие изменения внесли ваши коллеги, можете загрузить изменения *из* репозитория origin:

```
git pull origin master
```

При дальнейшем чтении этой книги и в целом при применении Terraform не забывайте регулярно фиксировать (gitcommit) и загружать (gitpush) свои изменения. Это позволит работать над кодом совместно с членами вашей команды. К тому же все изменения вашей инфраструктуры

будут записываться в журнал фиксаций, что придется очень кстати в случае отладки. Больше о командной работе с Terraform вы узнаете в главе 8.

Развертывание одного веб-сервера

Следующим шагом будет развертывание на этом инстансе веб-сервера. Мы попытаемся развернуть простейшую веб-архитектуру: один веб-сервер, способный отвечать на HTTP-запросы (рис. 2.6).

В реальных условиях для создания веб-сервера использовался бы веб-фреймворк наподобие Ruby on Rails или Django. Чтобы не усложнять этот пример, мы запустим простейший веб-сервер, который всегда возвращает текст Hello,World29:

```
#!/bin/bash
echo "Hello, World" > index.html
nohup busybox httpd -f -p 8080 &
```

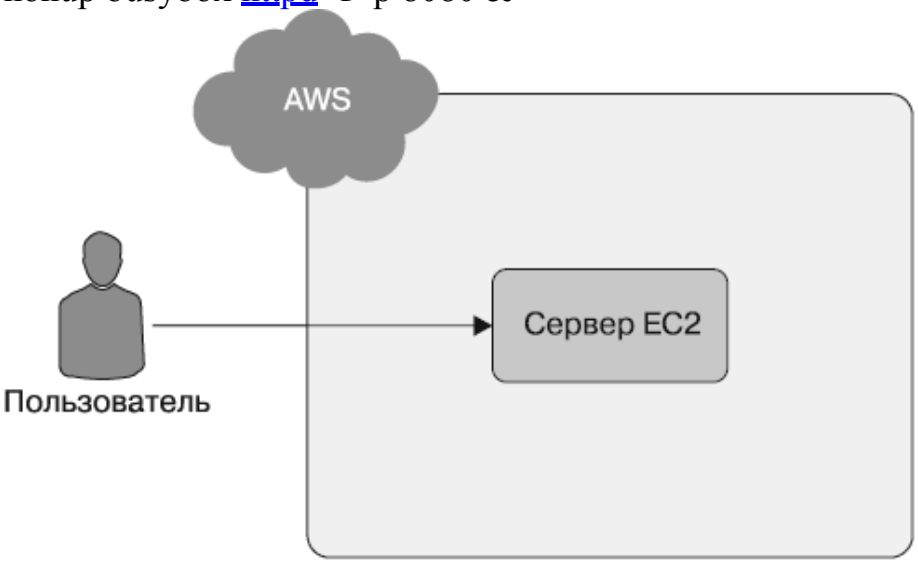


Рис. 2.6. Начнем с простой архитектуры: одного веб-сервера, запущенного в AWS, который отвечает на HTTP-запросы

Это bash-скрипт, записывающий текст Hello, World в файл index.html, который затем раздается на порте 8080 с помощью веб-сервера, запущенного с использованием инструмента под названием busybox (https://busybox.net/) (в Ubuntu установлен по умолчанию). Я указал команду busybox между nohup и &, чтобы веб-сервер постоянно работал в фоне, даже после завершения bash-скрипта.

Номера портов

В этом примере вместо стандартного HTTP-порта 80 используется 8080. Дело в том, что все порты меньше 1024 требуют администраторских привилегий. Это потенциальный риск безопасности, поскольку любой злоумышленник, которому удастся взломать ваш сервер, тоже получит привилегии администратора.

В связи с этим веб-сервер рекомендуется запускать от имени обычного пользователя с ограниченными правами доступа. Это означает, что вам следует прослушивать порты с более высокими номерами, но, как вы увидите позже в этой главе, вы можете настроить балансировщик нагрузки так, чтобы он работал на порте 80 и перенаправлял трафик на более высокие порты ваших серверов.

Как запустить этот скрипт на сервере EC2? Как уже упоминалось в подразделе «Средства шаблонизации серверов» на с. 31, для этого обычно применяется инструмент вроде Packer, который создает пользовательский образ AMI с предустановленным веб-сервером. Поскольку в этом примере в качестве веб-сервера используется однострочный скрипт на основе busybox, вы можете обойтись стандартным образом Ubuntu 18.04 и запустить скрипт Hello, World в рамках конфигурации *пользовательских данных* вашего сервера EC2. Вы можете передать параметру user_data в коде Terraform то, что будет выполнено при загрузке сервера: либо скрипт командной оболочки, либо директиву cloud-init. Вариант со скриптом показан ниже:

```
resource "aws_instance" "example" {
  ami                 = "ami-0c55b159cbfafe1f0"
  instance_type       = "t2.micro"
  user_data = <<-EOF
              #!/bin/bash
              echo "Hello, World" > index.html
              nohup busybox httpd -f -p 8080 &
              EOF
  tags = {
    Name = "terraform-example"
  }
}
```

<<-EOF и EOF — элементы синтаксиса *heredoc*, который позволяет создавать многострочные строковые литералы без использования множества символов перехода на новую строку.

Перед запуском этого веб-сервера нужно сделать еще кое-что. AWS по умолчанию закрывает для сервера EC2 весь входящий и исходящий трафик.

Чтобы ваш сервер мог принимать запросы на порте 8080, необходимо создать *группу безопасности*:

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"
  ingress {
    from_port   = 8080
    to_port     = 8080
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Этот код создает новый ресурс под названием aws_security_group (обратите внимание, что все ресурсы в AWS начинаются с aws_) и делает так, чтобы эта группа разрешала принимать на порте 8080 TCP-запросы из блока CIDR 0.0.0.0/0. *Блок CIDR* — это краткая запись диапазона IP-адресов. Например, блок CIDR 10.0.0.0/24 представляет все IP-адреса между 10.0.0.0 и 10.0.0.255. Блок CIDR 0.0.0.0/0 охватывает диапазон всех возможных IP-адресов, поэтому данная группа безопасности разрешает принимать на порте 8080 запросы с любого IP30.

Создания группы безопасности как таковой будет недостаточно. Нужно сделать так, чтобы сервер EC2 ее использовал. Для этого вы должны передать ее идентификатор аргументу vpc_security_group_ids ресурса aws_instance. Но сначала необходимо познакомиться с *выражениями* Terraform.

В Terraform выражением является все, что возвращает значение. Вы уже видели простейший тип выражений — *литералы*, такие как строки (например, "ami-0c55b159cbfafe1f0") и числа (скажем, 5). Terraform поддерживает много других разновидностей выражений, которые будут встречаться на страницах этой книги.

Особенно полезным типом выражений является *ссылка*, которая позволяет обращаться к значениям с других участков кода. Чтобы указать ID группы безопасности, нужно *сослаться на атрибут ресурса* с помощью такого синтаксиса:

<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>

PROVIDER — это имя провайдера (например, aws), TYPE — это тип ресурса (вроде security_group), NAME — имя этого ресурса (в нашем случае группа безопасности называется "instance"), а ATTRIBUTE — это либо один из аргументов ресурса (скажем, name), либо один из атрибутов, которые он *экспортировал* (список доступных атрибутов можно найти в документации

каждого ресурса). Группа безопасности экспортирует атрибут под названием id, поэтому к нему можно обратиться с помощью такого выражения:

```
aws_security_group.instance.id
```

Вы можете использовать идентификатор этой группы безопасности в аргументе vpc_security_group_ids ресурса aws_instance:

```
resource "aws_instance" "example" {
  ami                    = "ami-0c55b159cbfafe1f0"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  user_data = <<-EOF
              #!/bin/bash
              echo "Hello, World" > index.html
              nohup busybox httpd -f -p 8080 &
              EOF
  tags = {
    Name = "terraform-example"
  }
}
```

Ссылаясь в одном ресурсе на другой, вы создаете *неявную зависимость*. Terraform анализирует такие зависимости, строит из них граф и применяет его для автоматического определения порядка, в котором должны создаваться ресурсы. Например, если бы этот код развертывался с нуля, система Terraform знала бы о том, что группу безопасности нужно создать раньше, чем сервер EC2, поскольку последний использует ID этой группы. Вы можете даже вывести граф зависимостей с помощью команды graph:

```
$ terraform graph
digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] aws_instance.example"
                  [label = "aws_instance.example", shape = "box"]
                "[root] aws_security_group.instance"
                  [label = "aws_security_group.instance", shape = "box"]
                "[root] provider.aws"
                  [label = "provider.aws", shape = "diamond"]
                "[root] aws_instance.example" ->
                  "[root] aws_security_group.instance"
                "[root] aws_security_group.instance" ->
```

```
        "[root] provider.aws"
      "[root] meta.count-boundary (EachMode fixup)" ->
        "[root] aws_instance.example"
      "[root] provider.aws (close)" ->
        "[root] aws_instance.example"
      "[root] root" ->
        "[root] meta.count-boundary (EachMode fixup)"
      "[root] root" ->
        "[root] provider.aws (close)"
    }
}
```

Вывод выполнен на языке описания графов под названием DOT. Сам граф можно отобразить, как это сделано на рис. 2.7, с использованием настольного приложенияGraphviz или его веб-версии GraphvizOnline (bit.ly/2mPbxmg).

При прохождении по дереву зависимостей Terraform пытается как можно сильнее распараллелить создание ресурсов, что приводит к довольно эффективному применению изменений. В этом прелесть декларативного языка: вы просто описываете то, что вам нужно, а Terraform определяет наиболее эффективный способ реализации.

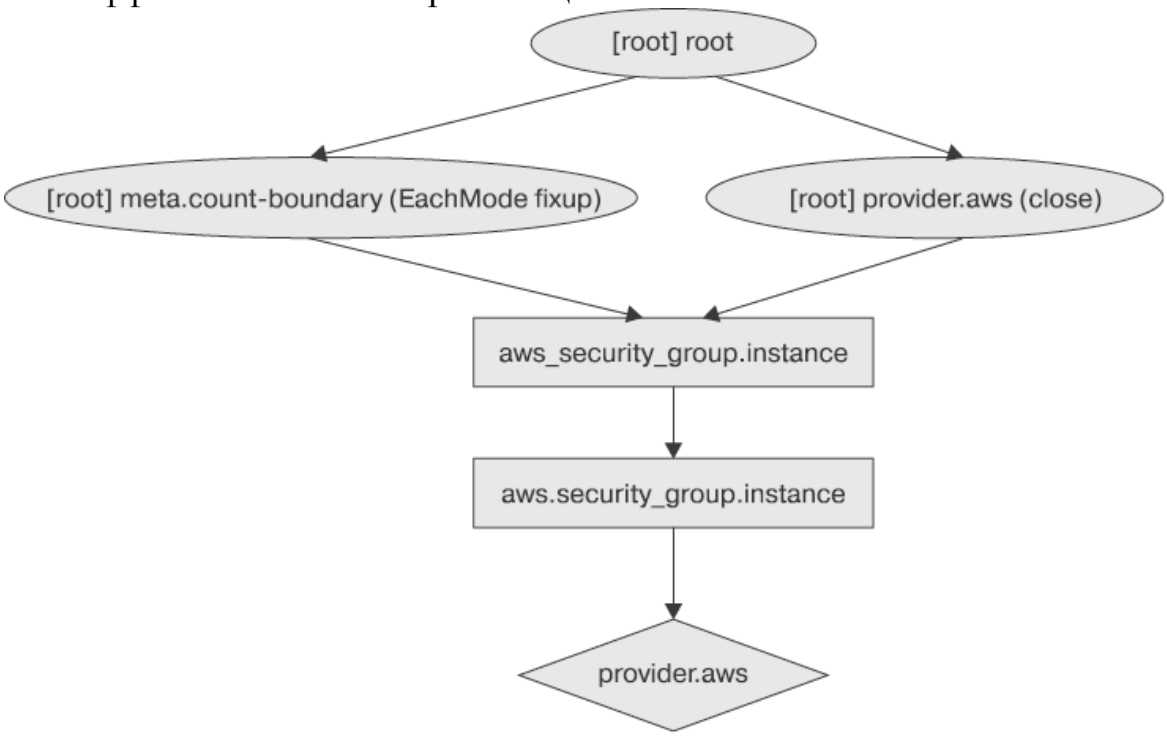


Рис. 2.7. Граф зависимостей для сервера EC2 и его группы безопасности

Выполнив команду apply, вы увидите, что Terraform хочет создать группу безопасности и заменить имеющийся сервер EC2 другим — с новыми пользовательскими данными:

```
$ terraform apply
(...)
Terraform will perform the following actions:
  # aws_instance.example must be replaced
-/+ resource "aws_instance" "example" {
      ami                    = "ami-0c55b159cbfafe1f0"
    ~ availability_zone      = "us-east-2c" -> (known after apply)
    ~ instance_state         = "running" -> (known after apply)
      instance_type          = "t2.micro"
      (...)
    + user_data              = "c765373..." # forces replacement
    ~ volume_tags            = {} -> (known after apply)
    ~ vpc_security_group_ids = [
        - "sg-871fa9ec",
      ] -> (known after apply)
      (...)
  }
# aws_security_group.instance will be created
+ resource "aws_security_group" "instance" {
    + arn                  = (known after apply)
    + description          = "Managed by Terraform"
    + egress               = (known after apply)
    + id                   = (known after apply)
    + ingress              = [
        + {
            + cidr_blocks      = [
                + "0.0.0.0/0",
              ]
            + description      = ""
            + from_port        = 8080
            + ipv6_cidr_blocks = []
            + prefix_list_ids  = []
            + protocol         = "tcp"
            + security_groups  = []
            + self             = false
            + to_port          = 8080
          },
```

```
      ]
    + name                = "terraform-example-instance"
    + owner_id            = (known after apply)
    + revoke_rules_on_delete = false
    + vpc_id              = (known after apply)
  }
Plan: 2 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:
```

Символы -/+ в выводе плана означают «заменить». Чтобы понять, чем продиктовано то или иное изменение, поищите в выводе плана словосочетание forces replacement. Изменение многих аргументов ресурса aws_instance приводит к замене. Это означает, что имеющийся сервер EC2 будет удален, а его место займет совершено новый сервер. Это пример парадигмы неизменяемой инфраструктуры, которую мы обсуждали в подразделе «Средства шаблонизации серверов» на с. 31. Стоит отметить, что, несмотря на замену веб-сервера, ни один из его пользователей не заметит перебоев в работе; в главе 5 вы увидите, как с помощью Terraform выполнять развертывания с нулевым временем простоя.

Похоже, с планом все в порядке, поэтому введите yes, и вы увидите, как развертывается новый сервер EC2 (рис. 2.8).

Если щелкнуть на новом сервере, внизу страницы, на панели с описанием, можно увидеть его публичный IP-адрес. Дайте ему минуту или две, чтобы он загрузился, и затем сделайте HTTP-запрос по этому адресу на порте 8080, используя браузер или утилиту вроде curl:

```
$ curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
Hello, World
```

Ура! Теперь у вас есть рабочий веб-сервер, запущенный в AWS!

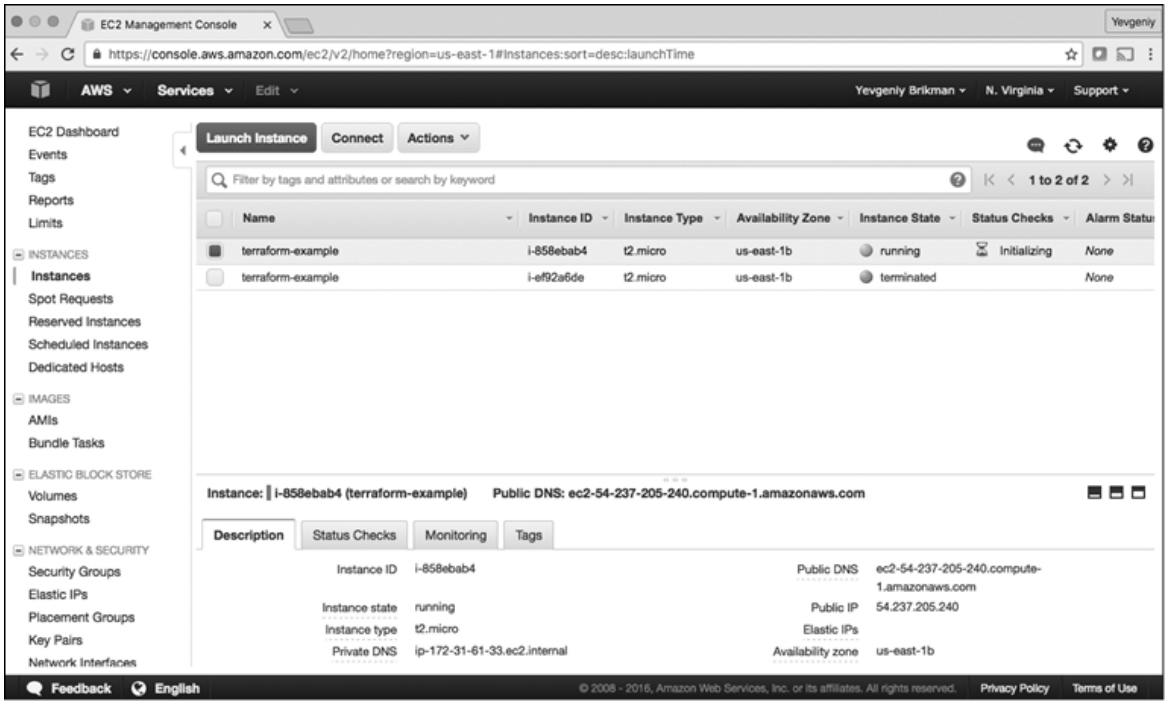


Рис. 2.8. Вместо старого сервера EC2 мы получаем новый, с кодом веб-сервера

Сетевая безопасность

Чтобы не усложнять примеры в этой книге, развертывание происходит не только в VPC по умолчанию (как упоминалось ранее), но и в стандартные подсети этого VPC. VPC состоит из одной или нескольких подсетей, каждая из которых имеет собственные IP-адреса. Все подсети в VPC по умолчанию являются публичными — их IP-адреса доступны из Интернета. Благодаря этому вы можете проверить работу своего сервера EC2 на домашнем компьютере.

Размещение сервера в публичной подсети подходит для быстрого эксперимента, но в реальных условиях это потенциальный риск безопасности. Хакеры со всего мира постоянно сканируют IP-адреса случайным образом в надежде найти какие-нибудь уязвимости. Если ваши серверы доступны снаружи, достаточно лишь оставить незащищенным один порт или воспользоваться устаревшим кодом с известной уязвимостью — и кто-то сможет проникнуть внутрь.

Таким образом, в промышленных системах все серверы и уж точно все хранилища данных следует развертывать в закрытых подсетях, IP-адреса которых доступны только внутри VPC, но не из публичного Интернета. Все, что должно находиться в публичных подсетях, — это небольшое количество обратных прокси и балансировщиков нагрузки, в которых закрыто все, что только можно (позже в этой главе вы увидите пример того, как развернуть балансировщик нагрузки).

Развертывание конфигурируемого веб-сервера

Вы, наверное, заметили, что код веб-сервера дублирует порт 8080 в группе безопасности и конфигурации пользовательских данных. Это противоречит принципу *«не повторяйся»* (don't repeat yourself, или DRY): каждый элемент информации в системе должен иметь единое, однозначное и достоверное представление31. Если номер порта указан в двух местах, легко оказаться в ситуации, когда одно из значений обновлено, а другое — нет.

Чтобы можно было сделать ваш код более конфигурируемым и отвечающим принципу DRY, Terraform позволяет определять *входные переменные*. Для этого предусмотрен следующий синтаксис:

```
variable "NAME" {
  [CONFIG ...]
}
```

Тело объявления переменной может содержать три необязательных параметра.

•description. Этот параметр всегда желательно указывать для документирования того, как используется переменная. Ваши коллеги смогут просмотреть это описание не только при чтении кода, но и во время выполнения команд plan или apply (пример этого показан чуть ниже).

• default. Вы можете присвоить значение переменной несколькими способами, в том числе через командную строку (с помощью параметра -var), файл (указывая параметр -var-file) или переменную среды (Terraform ищет переменные среды вида TF_VAR_<имя_переменной>). Если переменная не инициализирована, ей присваивается значение по умолчанию. Если такого нет, Terraform запросит его у пользователя в интерактивном режиме.

•type. Позволяет применить к переменным, которые передает пользователь, *ограничения типов*. Terraform поддерживает ряд ограничений для таких типов, как string, number, bool, list, map, set, object, tuple и any. Если тип не указан, Terraform воспринимает значение как any.

Вот пример входной переменной, которая проверяет, является ли передаваемое значение числом:

```
variable "number_example" {
  description = "An example of a number variable in Terraform"
  type        = number
  default     = 42
}
```

А вот пример переменной, которая проверяет, является ли значение списком:

```
variable "list_example" {
  description = "An example of a list in Terraform"
  type        = list
  default     = ["a", "b", "c"]
}
```

Ограничения типов можно сочетать. Например, вот входная переменная, которая принимает список и требует, чтобы все значения этого списка были числовыми:

```
variable "list_numeric_example" {
  description = "An example of a numeric list in Terraform"
  type        = list(number)
  default     = [1, 2, 3]
}
```

А вот ассоциативный массив, который требует, чтобы все значения были строковыми:

```
variable "map_example" {
  description = "An example of a map in Terraform"
  type        = map(string)
  default = {
    key1 = "value1"
    key2 = "value2"
    key3 = "value3"
  }
}
```

Вы также можете создавать более сложные *структурные типы*, используя ограничения object и tuple:

```
variable "object_example" {
  description = "An example of a structural type in Terraform"
  type        = object({
    name    = string
    age     = number
```

```
  tags    = list(string)
  enabled = bool
 })
 default  = {
  name    = "value1"
  age     = 42
  tags    = ["a", "b", "c"]
  enabled = true
 }
}
```

Выше создается входная переменная, которая требует, чтобы значение было объектом с ключами name (строка), age (число), tags (список строк) и enabled (булево значение). Если попытаться присвоить такой переменной значение, которое не соответствует этому типу, Terraform немедленно вернет ошибку типизации. В следующем примере демонстрируется попытка присвоить enabled строку вместо булева значения:

```
variable "object_example_with_error" {
 description = "An example of a structural type in Terraform with an error"
 type        = object({
  name    = string
  age     = number
  tags    = list(string)
  enabled = bool
 })
 default  = {
  name    = "value1"
  age     = 42
  tags    = ["a", "b", "c"]
  enabled = "invalid"
 }
}
```

Вы получите следующую ошибку:

```
$ terraform apply
Error: Invalid default value for variable
 on variables.tf line 78, in variable "object_example_with_error":
 78:   default  = {
 79:     name    = "value1"
 80:     age     = 42
 81:     tags    = ["a", "b", "c"]
 82:     enabled = "invalid"
```

```
  83:   }
```

This default value is not compatible with the variable's type constraint: a bool is required.

Для примера с веб-сервером достаточно переменной, которая хранит номер порта:

```
variable "server_port" {
  description = "The port the server will use for HTTP requests"
  type        = number
}
```

Обратите внимание, что у входной переменной server_port нет поля default, поэтому, если выполнить команду apply прямо сейчас, Terraform сразу же попросит ввести значение для server_port и покажет вам описание description:

```
$ terraform apply
var.server_port
  The port the server will use for HTTP requests
  Enter a value:
```

Если вы не хотите иметь дело с интерактивной строкой ввода, можете предоставить значение переменной с помощью параметра командной строки -var:

```
$ terraform plan -var "server_port=8080"
```

То же самое можно сделать, добавив переменную среды вида TF_VAR_<name>, где name — имя переменной, которую вы хотите установить:

```
$ export TF_VAR_server_port=8080
$ terraform plan
```

Если же вы не хотите держать в голове дополнительные аргументы командной строки при каждом выполнении команды plan или apply, можете указать значение по умолчанию:

```
variable "server_port" {
  description = "The port the server will use for HTTP requests"
  type        = number
  default     = 8080
}
```

Чтобы использовать значение входной переменной в коде Terraform, следует воспользоваться выражением типа *«ссылка на переменную»*, которое имеет следующий синтаксис:

```
var.<VARIABLE_NAME>
```

Например, так можно присвоить параметрам группы безопасности from_port и to_port значение переменной server_port:

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"
  ingress {
    from_port   = var.server_port
    to_port     = var.server_port
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Хорошей идеей будет использование одной и той же переменной при задании порта в скрипте user_data. Указать ссылку внутри строкового литерала можно с помощью *строковой интерполяции*, которая имеет следующий синтаксис:

```
"${...}"
```

Внутри фигурных скобок можно разместить любую корректную ссылку, и Terraform преобразует ее в строку. Например, вот как можно воспользоваться значением var.server_port внутри строки user_data:

```
user_data = <<-EOF
            #!/bin/bash
            echo "Hello, World" > index.html
            nohup busybox httpd -f -p ${var.server_port} &
            EOF
```

Помимо входных переменных, Terraform позволяет определять и *выходные*. Для этого предусмотрен такой синтаксис:

```
output "<NAME>" {
  value = <VALUE>
   [CONFIG ...]
}
```

NAME — это имя выходной переменной, а в качестве VALUE можно указать любое выражение Terraform, которое вы хотите вывести. CONFIG может иметь два дополнительных (и необязательных) параметра.

•description. Этот параметр всегда желательно применять для документирования того, как используется выходная переменная.

•sensitive. Если присвоить данному параметру true, Terraform не станет сохранять этот вывод в журнал после выполнения команды terraformapply. Это полезно, когда переменная содержит деликатный материал или конфиденциальные данные, такие как пароли или секретные ключи.

Например, вместо того, чтобы вручную бродить по консоли EC2 в поисках IP-адреса своего сервера, вы можете вывести его в виде выходной переменной:

```
output "public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP address of the web server"
}
```

Здесь мы опять ссылаемся на атрибуты, на этот раз на атрибут public_ip ресурса aws_instance. Если снова выполнить команду apply, Terraform не внесет никаких изменений (поскольку вы не меняли никакие ресурсы), но покажет вам в самом конце новый вывод:

```
$ terraform apply
(...)
aws_security_group.instance: Refreshing state... [id=sg-078ccb4f9533d2c1a]
aws_instance.example: Refreshing state... [id=i-028cad2d4e6bddec6]
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
public_ip = 54.174.13.5
```

Как видите, после выполнения terraformapply выходная переменная выводится в консоли, что может пригодиться пользователям Terraform (например, вы будете знать, какой IP-адрес нужно проверить после развертывания веб-сервера). Вы также можете ввести команду terraformoutput, чтобы вывести список всех выходных значений без применения каких-либо изменений:

```
$ terraform output
public_ip = 54.174.13.5
```

Чтобы посмотреть значение определенной выходной переменной, можно воспользоваться командой terraformoutput<ИМЯ_ПЕРЕМЕННОЙ>:

```
$ terraform output public_ip
54.174.13.5
```

Это будет особенно полезно при написании скриптов. Например, вы можете создать скрипт развертывания, который развертывает веб-сервер с помощью команды terraformapply, берет его публичный IP-адрес из terraformoutputpublic_ip и обращается к этому адресу, используя curl. В итоге получится проверка по горячим следам, которая подтвердит, что развертывание работает.

Входные и выходные переменные также являются неотъемлемыми составляющими при создании конфигурируемого инфраструктурного кода, пригодного к повторному применению. Подробнее об этом — в главе 4.

Развертывание кластера веб-серверов

Запуск одного сервера — хорошее начало. Однако в реальном мире это означает наличие единой точки отказа. Если этот сервер выйдет из строя или перестанет справляться с нагрузкой из-за слишком большого объема трафика, пользователи не смогут открыть ваш сайт. В качестве решения можно запустить кластер серверов: если один из них откажет, запросы допускается перенаправить к другому серверу, а размер самого кластера можно увеличивать и уменьшать в зависимости от трафика32.

Ручное управление таким кластером потребует много усилий. К счастью, как показано на рис. 2.9, AWS может позаботиться об этом за вас, используя группу автомасштабирования (англ. auto scaling group, или ASG). ASG автоматически выполняет множество задач, включая запуск кластера серверов EC2, мониторинг работоспособности каждого сервера, замену неисправных серверов и изменение размера кластера в зависимости от нагрузки.

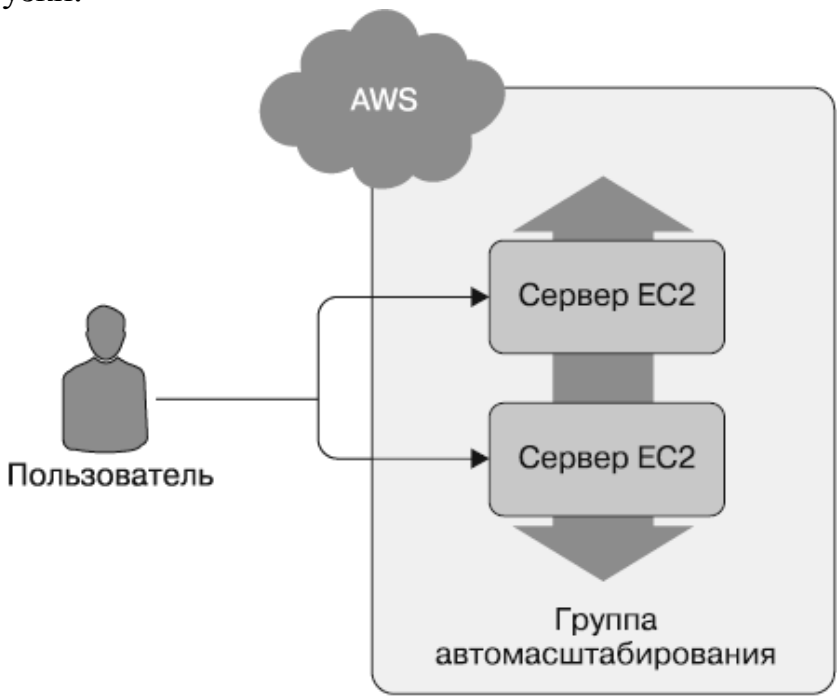


Рис. 2.9. Вместо одного веб-сервера группа автомасштабирования запускает кластер веб-серверов

Первое, что нужно сделать при создании ASG, — это написать *конфигурацию запуска*, которая определяет, как нужно настроить каждый сервер EC2 в вашей группе. Ресурс aws_launch_configuration использует почти все те же параметры, что и aws_instance (только у двух из них отличаются имена: image_id вместо ami и security_groups вместо vpc_security_group_ids), поэтому вы можете их легко заменить:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
```

```
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]
  user_data = <<-EOF
              #!/bin/bash
              echo "Hello, World" > index.html
              nohup busybox httpd -f -p ${var.server_port} &
              EOF
}
```

Теперь вы можете создать саму группу ASG, используя ресурс aws_autoscaling_group:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  min_size = 2
  max_size = 10
  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

Эта группа ASG включает в себя от двух до десяти серверов EC2 (сначала запускается только два), каждый из которых имеет тег terraform-asg-example. Обратите внимание, что ASG использует ссылку в качестве имени конфигурации запуска. Это приводит к проблеме: конфигурация запуска неизменяема, поэтому, если вы измените любой ее параметр, Terraform попытается заменить ее целиком. Обычно при замене ресурса Terraform сначала удаляет его старую версию и затем создает новую, но, поскольку ASG теперь ссылается на старый ресурс, Terraform не сможет его удалить.

Чтобы решить эту проблему, можно воспользоваться параметром *жизненного цикла*. Он поддерживается всеми ресурсами Terraform и определяет их создание, обновление и/или удаление. Особенно полезным параметром жизненного цикла является create_before_destroy. Если присвоить ему true, Terraform поменяет порядок замены ресурсов на противоположный. В итоге сначала будет создана замена (с обновлением всех ссылок таким образом, чтобы они указывали на нее, а не на старый ресурс) и только потом произойдет удаление старого ресурса. Добавьте раздел lifecycle в aws_launch_configuration, как показано ниже:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
```

```
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]
  user_data = <<-EOF
              #!/bin/bash
              echo "Hello, World" > index.html
              nohup busybox httpd -f -p ${var.server_port} &
              EOF
  # Требуется при использовании группы автомасштабирования
  # в конфигурации запуска.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

Для работы группы ASG требуется еще один параметр: subnet_ids. Он определяет подсети VPC, в которых должны быть развернуты серверы EC2 (справочная информация о подсетях дается во врезке «Сетевая безопасность» на с. 79). Каждая подсеть находится в изолированной зоне доступности AWS (то есть в отдельном вычислительном центре), поэтому, развертывая свои серверы по разным подсетям, вы гарантируете, что ваш сервис продолжит работать, даже если некоторые из вычислительных центров выйдут из строя. Список подсетей можно прописать прямо в коде, но такое решение будет сложно поддерживать и переносить. Вместо этого лучше использовать соответствующий *источник данных* для получения списка подсетей, принадлежащих к вашей учетной записи AWS.

Источник данных представляет собой фрагмент информации, доступной сугубо для чтения, который извлекается из провайдера (в нашем случае из AWS) при каждом запуске Terraform. Добавляя источник данных в конфигурацию Terraform, вы не создаете ничего нового. Это просто возможность запросить информацию из API провайдера, чтобы сделать ее доступной для остального кода Terraform. Каждый провайдер предоставляет целый ряд источников. Например, провайдер AWS позволяет запрашивать данные о VPC и подсетях, идентификаторы AMI, диапазоны IP-адресов, идентификатор текущего пользователя и многое другое.

Синтаксис использования источников данных очень похож на синтаксис ресурса:

```
data "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

PROVIDER — имя провайдера (например, aws), TYPE — тип источника данных, который вы хотите использовать (скажем, vpc), NAME — идентификатор, с помощью которого можно ссылаться на этот источник данных в коде Terraform, а CONFIG состоит из одного или нескольких аргументов, предусмотренных специально для этого источника. Вот как можно воспользоваться источником данных aws_vpc, чтобы запросить информацию о вашем облаке VPC по умолчанию (справочную информацию ищите во врезке «Замечание о виртуальных частных облаках по умолчанию» на с. 63):

```
data "aws_vpc" "default" {
  default = true
}
```

Стоит отметить, что в случае с источниками данных в качестве аргументов обычно передаются фильтры, которые указывают на то, какую информацию вы ищете. Источнику данных aws_vpc нужно указать лишь один фильтр, default=true, который инициирует в вашей учетной записи AWS поиск VPC по умолчанию.

Чтобы получить данные из источника, нужно использовать следующий синтаксис доступа к атрибутам:

```
data.<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

Например, чтобы получить идентификатор VPC из источника данных aws_vpc, надо написать следующее:

```
data.aws_vpc.default.id
```

Вы можете добавить к этому еще один источник данных, aws_subnet_ids, чтобы найти подсети внутри этого облака VPC:

```
data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}
```

Вы можете извлечь идентификаторы подсетей из источника aws_subnet_ids и воспользоваться аргументом с довольно странным названием vpc_zone_identifier, чтобы ваша группа ASG использовала эти подсети:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  min_size = 2
  max_size = 10
  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
```

```
    propagate_at_launch = true
  }
}
```

Развертывание балансировщика нагрузки

Вы уже научились развертывать группу ASG, но при этом возникает небольшая проблема: у вас есть несколько серверов с отдельными IP-адресами, однако конечным пользователям обычно нужна единая точка входа. Одно из решений заключается в развертывании *балансировщика нагрузки*, который будет распределять трафик между вашими серверами и предоставлять всем вашим пользователям собственный IP-адрес (или, если быть точным, доменное имя). Создание балансировщика нагрузки с высокими доступностью и масштабируемостью требует много усилий. И опять вы можете положиться на AWS, воспользовавшись сервисом *Elastic Load Balancer* (ELB) от Amazon (рис. 2.10).

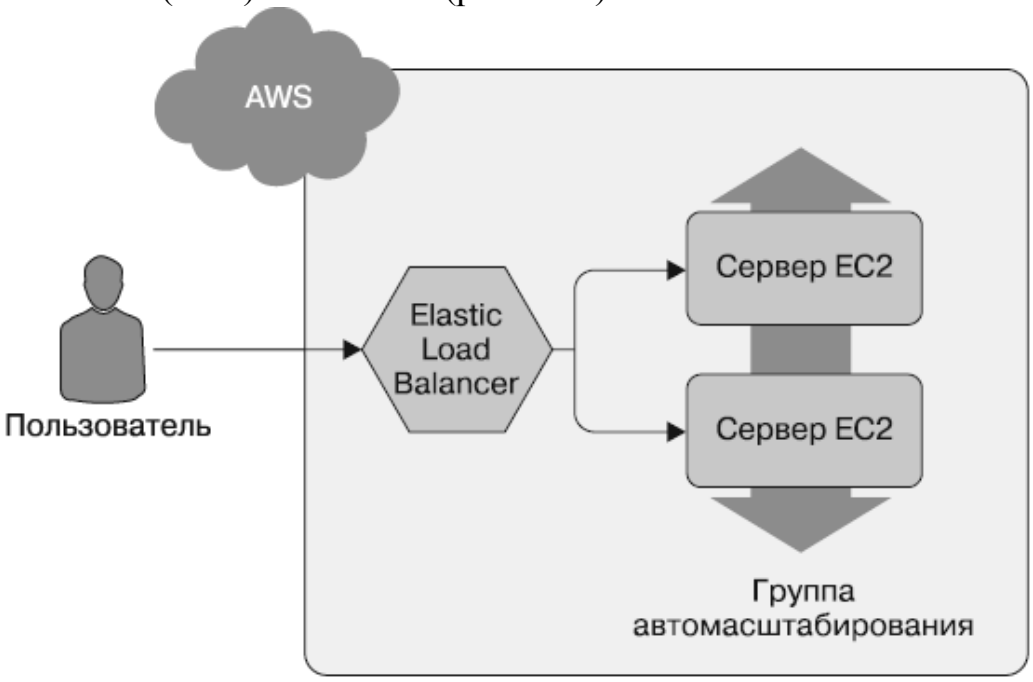


Рис. 2.10. Использование Amazon ELB для распределения трафика по группе автомасштабирования

AWS предлагает три типа балансировщиков нагрузки.

•*Application Load Balancer (ALB)*. Лучше всего подходит для балансировки трафика по протоколам HTTP и HTTPS. Работает на прикладном уровне (уровень 7) сетевой модели OSI.

• *Network Load Balancer (NLB)*. Лучше всего подходит для балансировки трафика по протоколам TCP, UDP и TLS. Если сравнивать с ALB, может быстрее масштабироваться в обе стороны в зависимости от

нагрузки (NLB рассчитан на масштабирование до десятков миллионов запросов в секунду). Работает на транспортном уровне (уровень 4) сетевой модели OSI.

•*Classic Load Balancer (CLB)*. Это «старый» балансировщик нагрузки, который появился раньше, чем ALB и NLB. Он способен работать с трафиком по протоколам HTTP, HTTPS, TCP и TLS, но ограничен в возможностях по сравнению со своими «преемниками». Работает как на прикладном, так и на транспортном уровне (L7 и L4) сетевой модели OSI.

Современные приложения должны использовать в основном ALB или NLB. Поскольку наш простой пример с веб-сервером является HTTP-приложением без серьезных требований к производительности, нам лучше всего подойдет ALB.

Как показано на рис. 2.11, ALB состоит из нескольких частей.

•*Прослушиватель (слушатель)*. Прослушивает определенные порты (например, 80) и протокол (скажем, HTTP).

• *Правило прослушивателя*. Берет запросы, направленные к прослушивателю, и передает те из них, которые соответствуют определенным путям (например, /foo или /bar) или сетевым именам (вроде foo.example.com или bar.example.com), заданным целевым группам.

•*Целевые группы*. Один или несколько серверов, которые принимают запросы от балансировщика нагрузки. Целевая группа также следит за работоспособностью этих серверов и шлет запросы только «здоровым» узлам.

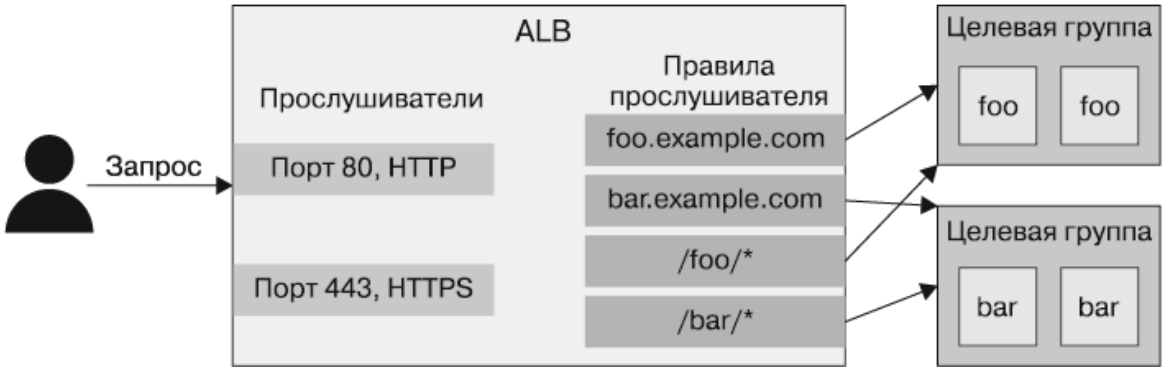


Рис. 2.11. Обзор Application Load Balancer (ALB)

Первым делом нужно создать сам балансировщик ALB, используя ресурс aws_lb:

```
resource "aws_lb" "example" {
  name               = "terraform-asg-example"
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
}
```

Обратите внимание, что параметр subnets настраивает балансировщик нагрузки для использования всех подсетей в вашем облаке VPC по умолчанию с помощью источника данных aws_subnet_ids33. Балансировщики нагрузки в AWS состоят не из одного, а сразу из нескольких серверов, которые могут работать в разных подсетях (и, следовательно, в отдельных вычислительных центрах). AWS автоматически масштабирует количество балансировщиков в зависимости от трафика и отрабатывает отказ, если один из этих серверов выйдет из строя. Таким образом, вы получаете как масштабируемость, так и высокую доступность.

Следующим шагом будет определение прослушивателя для этого ALB с помощью ресурса aws_lb_listener:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"
  # По умолчанию возвращает простую страницу с кодом 404
  default_action {
    type = "fixed-response"
    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

Этот прослушиватель настраивает ALB для прослушивания стандартного HTTP-порта (80), использования HTTP-протокола и возвращения простой страницы с кодом 404 в случае, если запрос не соответствует ни одному из правил прослушивания.

Стоит отметить, что по умолчанию для всех ресурсов AWS, включая ALB, закрыт входящий и исходящий трафик, поэтому нужно создать новую группу безопасности специально для балансировщика нагрузки. Эта группа должна разрешать входящие запросы на порт 80, чтобы вы могли обращаться к ALB по HTTP, а также исходящие запросы на всех портах, чтобы балансировщик мог проверять работоспособность:

```
resource "aws_security_group" "alb" {
  name = "terraform-example-alb"
  # Разрешаем все входящие HTTP-запросы
  ingress {
    from_port   = 80
```

```
  to_port     = 80
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
 }
 # Разрешаем все исходящие запросы
 egress {
  from_port   = 0
  to_port     = 0
  protocol    = "-1"
  cidr_blocks = ["0.0.0.0/0"]
 }
}
```

Нужно сделать так, чтобы ресурс aws_lb использовал указанную группу безопасности. Для этого установите аргумент security_groups:

```
resource "aws_lb" "example" {
 name               = "terraform-asg-example"
 load_balancer_type = "application"
 subnets            = data.aws_subnet_ids.default.ids
 security_groups    = [aws_security_group.alb.id]
}
```

Затем необходимо создать целевую группу для ASG, используя ресурс aws_lb_target_group:

```
resource "aws_lb_target_group" "asg" {
 name     = "terraform-asg-example"
 port     = var.server_port
 protocol = "HTTP"
 vpc_id   = data.aws_vpc.default.id
 health_check {
  path                = "/"
  protocol            = "HTTP"
  matcher             = "200"
  interval            = 15
  timeout             = 3
  healthy_threshold   = 2
  unhealthy_threshold = 2
 }
}
```

Обратите внимание на то, что данная целевая группа будет проверять работоспособность ваших серверов, отправляя им периодически HTTP-запросы; сервер считается работоспособным, только если его ответ

совпадает с заданным сопоставителем (например, вы можете сделать так, чтобы сопоставитель ожидал ответа 200OK). Если сервер не отвечает (возможно, из-за перебоев в работе или перегрузки), он будет помечен как неработоспособный и целевая группа автоматически прекратит отправлять ему трафик, чтобы минимизировать нарушение обслуживания ваших пользователей.

Но откуда целевая группа знает, каким серверам EC2 следует отправлять запросы? Вы могли бы прописать в ней статический список серверов ресурсом aws_lb_target_group_attachment, однако при работе с ASG серверы могут запускаться и удаляться в любой момент, поэтому такой подход не годится. Вместо этого можно воспользоваться первоклассной интеграцией между ASG и ALB. Вернитесь к ресурсу aws_autoscaling_group и сделайте так, чтобы его аргумент target_group_arns указывал на новую целевую группу:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"
  min_size = 2
  max_size = 10
  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

Вы также должны поменять значение health_check_type на "ELB". Значение по умолчанию, "EC2", подразумевает минимальную проверку работоспособности, которая считает сервер неисправным, только если гипервизор AWS утверждает, что ВМ совсем не работает или является недоступной. Значение "ELB" более надежно, поскольку вынуждает ASG проверять работоспособность целевой группы. К тому же серверы автоматически заменяются, если целевая группа объявляет их неисправными. Таким образом, серверы подлежат замене не только в случае полного отказа, но и когда они, к примеру, перестают обслуживать запросы из-за нехватки памяти или остановки критически важного процесса.

Пришло время собрать это все воедино. Для этого мы создадим правила прослушивателя, используя ресурс aws_lb_listener_rule:

```
resource "aws_lb_listener_rule" "asg" {
```

```
  listener_arn = aws_lb_listener.http.arn
  priority     = 100
  condition {
    field  = "path-pattern"
    values = ["*"]
  }
  action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}
```

Этот код добавляет правило прослушивателя, которое отправляет запросы, соответствующие любому пути, к целевой группе с ASG внутри.

Прежде чем разворачивать балансировщик нагрузки, нужно сделать еще кое-что — поменять старый вывод public_ip одного сервера EC2 на вывод доменного имени ALB:

```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "The domain name of the load balancer"
}
```

Выполните terraformapply и почитайте полученный план. Согласно ему наш исходный сервер EC2 удаляется, а вместо него Terraform создает конфигурацию запуска, ASG, ALB и группу безопасности. Если с планом все в порядке, введите yes и нажмите клавишу Enter. Когда команда apply завершит работу, вы должны увидеть вывод alb_dns_name:

```
Outputs:
alb_dns_name = terraform-asg-example-123.us-east-2.elb.amazonaws.com
```

Скопируйте этот URL-адрес. Подождите, пока наши серверы загрузятся и ALB пометит их как работоспособные. Тем временем можете просмотреть то, что вы развернули. Открыв раздел ASG консоли EC2 (https://amzn.to/2MH3mId), вы должны увидеть, что группа автомасштабирования уже создана (рис. 2.12).

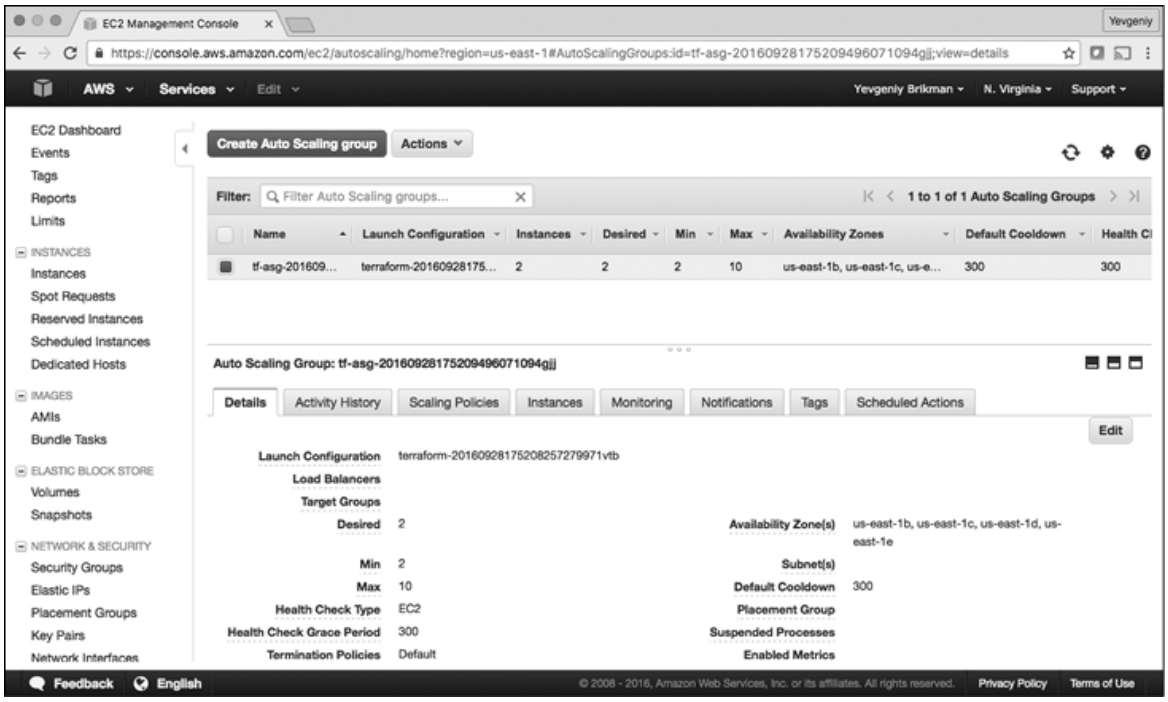


Рис. 2.12. Группа автомасштабирования

Если перейти на вкладку Instances (Серверы), можно увидеть запуск двух серверов EC2, как показано на рис. 2.13.

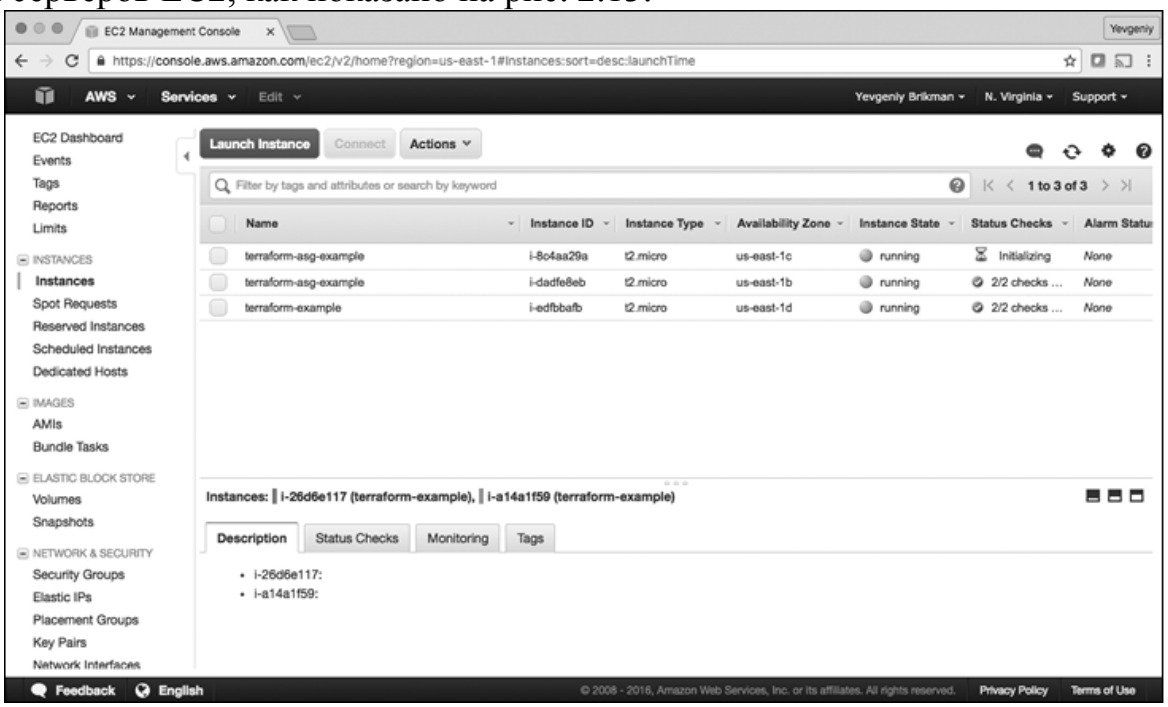


Рис. 2.13. Запуск серверов EC2 в ASG

Щелкнув на вкладке Load Balancers (Балансировщики нагрузки), вы увидите свой экземпляр ALB, как показано на рис. 2.14.

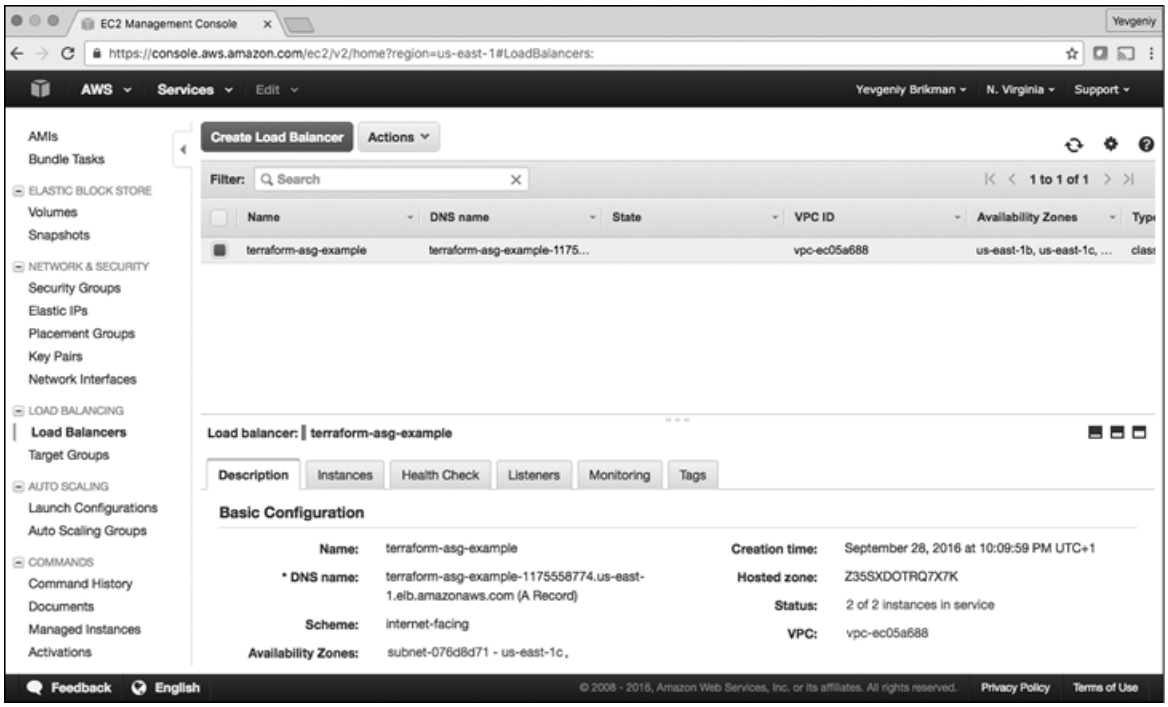


Рис. 2.14. Application Load Balancer

Чтобы найти целевую группу, как показано на рис. 2.15, перейдите на вкладку Target Groups (Целевые группы).

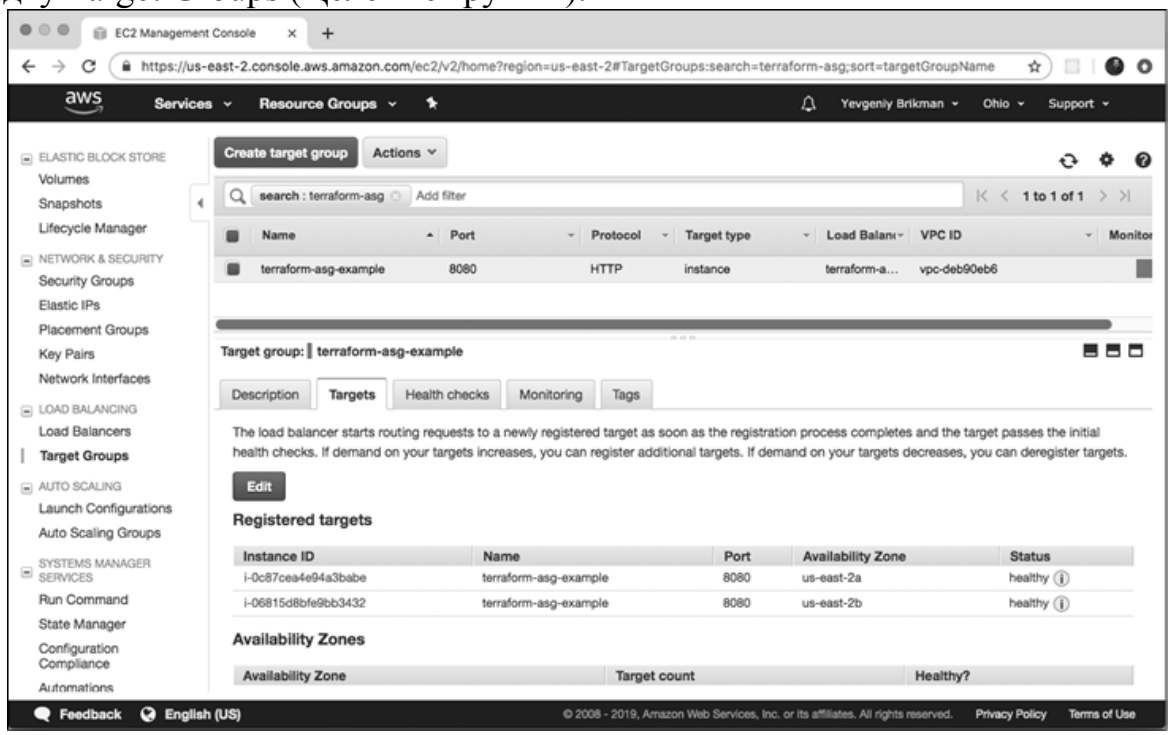


Рис. 2.15. Целевая группа

Если вы выберете свою целевую группу и щелкнете на вкладке Targets (Цели) в нижней части экрана, то сможете увидеть, как ваши серверы

регистрируются в целевой группе и проходят проверки работоспособности. Подождите, пока их индикатор состояния не начнет показывать healthy. Это обычно занимает одну-две минуты. После проверьте вывод alb_dns_name, который вы скопировали ранее:

```
$ curl http://<alb_dns_name>
Hello, World
```

Получилось! ALB направляет трафик к вашим серверам EC2. Каждый раз, когда вы обращаетесь по этому URL-адресу, он выбирает другой сервер для обработки запроса. Вы получили полностью рабочий кластер веб-серверов!

На этом этапе можно видеть, как ваш кластер реагирует на создание новых и удаление старых серверов. Например, перейдите на вкладку Instances (Серверы) и удалите один из серверов: установите флажок, нажмите вверху кнопку Actions (Действия) и затем поменяйте состояние сервера на Terminate (Удалить). Если вы снова попробуете обратиться к тестовому URL-адресу ALB, каждый ваш запрос должен вернуть 200OK даже после удаления сервера. ALB автоматически обнаружит, что сервера больше нет, и перестанет направлять к нему трафик. Что еще интересней, вскоре после удаления ASG поймет, что у нас остался один сервер вместо двух, и автоматически запустит замену (самовосстановление!). Чтобы увидеть, как ASG меняет свой размер, можете добавить в свой код Terraform параметр desired_capacity и снова выполнить команды apply.

Удаление ненужных ресурсов

Когда вы закончите экспериментировать с Terraform в конце этой или одной из следующих глав, желательно удалить все созданные вами ресурсы, чтобы за них не пришлось платить. Поскольку Terraform отслеживает все, что вы создаете, это не составит проблемы. Достаточно выполнить команду destroy:

```
$ terraform destroy
(...)
Terraform will perform the following actions:
  # aws_autoscaling_group.example will be destroyed
  - resource "aws_autoscaling_group" "example" {
      (...)
    }
  # aws_launch_configuration.example will be destroyed
  - resource "aws_launch_configuration" "example" {
      (...)
    }
```

```
  # aws_lb.example will be destroyed
  - resource "aws_lb" "example" {
      (...)
    }
  (...)
Plan: 0 to add, 0 to change, 8 to destroy.
Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.
  Enter a value:
```

Вы, наверное, и сами понимаете, что команда destroy в промышленной среде должна использоваться редко (или вообще никогда). Ее нельзя отменить, поэтому Terraform даст вам последний шанс взглянуть на свои действия, отобразив на экране список всех ресурсов, которые будут удалены, и запросит у вас подтверждение. Если все выглядит хорошо, введите yes и нажмите клавишу Enter. Terraform построит граф зависимостей и удалит все ваши ресурсы в корректном порядке, как можно сильнее распараллеливая этот процесс. Через одну-две минуты ваша учетная запись AWS снова будет пустой.

Имейте в виду, что в дальнейшем мы продолжим разрабатывать этот пример, поэтому не удаляйте код Terraform! Но вы можете в любой момент выполнить команду destroy, чтобы удалить уже развернутые ресурсы. Прелесть концепции IaC в том, что вся информация об этих ресурсах описана в коде, поэтому при желании вы можете воссоздать их все с помощью одной команды: terraformapply. На самом деле последние внесенные изменения лучше зафиксировать в Git, чтобы вы могли отслеживать историю своей инфраструктуры.

Резюме

У вас теперь есть общее преставление об использовании Terraform. Декларативный язык позволяет легко описывать именно ту инфраструктуру, которую вы хотите создать. С помощью команды plan можно просмотреть потенциальные изменения и устранить ошибки, прежде чем приступать к развертыванию. Переменные, ссылки и зависимости позволяют избавить ваш код от повторяющихся фрагментов и сделать его максимально конфигурируемым.

Но это лишь начало. Из главы 3 вы узнаете, как Terraform следит за тем, какая инфраструктура уже была создана, и то, насколько сильно это влияет на структурирование кода Terraform. В главе 4 научитесь применять модули

Terraform для создания инфраструктуры, пригодной для повторного использования.

25 Если технологии AWS кажутся вам запутанными, обязательно почитайте статью AWS in Plain English по адресу expeditedsecurity.com/aws-in-plain-english/.

26 Подробнее о рекомендуемых подходах к управлению пользователями в AWS можно почитать по ссылке amzn.to/2lvJ8Rf.

27 Больше о политиках IAM можно узнать на сайте AWS по адресу docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html.

28 Код Terraform можно также писать на чистом JSON и хранить в файлах с расширением .tf.json. Больше о синтаксисе HCL и JSON можно узнать на сайте Terraform: bit.ly/2MDZyaN.

29 На GitHub по адресу bit.ly/2OIIrr2 можно найти полезный список однострочных HTTP-серверов.

30 Больше о принципе работы CIDR можно узнать на странице в «Википедии» по адресу bit.ly/2l8Ki9g. Для преобразования диапазонов IP-адресов в CIDR и обратно можете использовать веб-калькулятор cidr.xyz или установить команду ipcalc в своем терминале.

31 Из книги: *Хант Э., Томас Д.* Программист-прагматик. Путь от подмастерья к мастеру (The Pragmatic Programmer). — М.: Лори, 2009.

32 Более подробное обсуждение построения высокодоступных и масштабируемых систем в AWS приводится в статье по адресу bit.ly/2mpSXUZ.

33 Чтобы не усложнять эти примеры, мы запускаем серверы EC2 и ALB в одной подсети. В промышленных условиях их почти наверняка следовало бы разместить в разных подсетях: серверы EC2 в закрытой (чтобы они не были доступны непосредственно из Интернета), а ALB — в открытой (чтобы пользователи могли обращаться к нему напрямую).

3. Как управлять состоянием Terraform

В главе 2, используя Terraform для создания и обновления ресурсов, вы могли заметить, что при каждом выполнении команд terraformplan и terraformapply этой системе удавалось находить созданные ранее ресурсы и обновлять их соответствующим образом. Но откуда ей было известно о том, какие из них находятся под ее управлением? В вашей учетной записи AWS может находиться любая инфраструктура, развернутая с помощью различных механизмов (отчасти вручную, отчасти через Terraform, отчасти с помощью утилиты командной строки). Так как же Terraform определяет свои ресурсы?

В этой главе вы узнаете, как Terraform отслеживает состояние вашей инфраструктуры и каким образом это влияет на структуру файлов и каталогов, изоляцию и блокирование в проекте Terraform. Вот ключевые темы, по которым мы пройдемся.

- Что собой представляет состояние Terraform.
- Общее хранилище для файлов состояния.
- Ограничения внутренних хранилищ Terraform.
- Изоляция файлов состояния.
- Изоляция с помощью рабочих областей.
- Изоляция с помощью описания структуры файлов.
- Источник данных terraform_remote_state.

Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

Что собой представляет состояние Terraform

При каждом своем запуске система Terraform записывает информацию о созданной ею инфраструктуре в свой *файл состояния*. По умолчанию, если запуск происходит в /foo/bar, Terraform создает файл /foo/bar/terraform.tfstate. Этот файл имеет нестандартный формат JSON и связывает ресурсы Terraform в ваших конфигурационных файлах с их представлением в реальном мире. Представьте, к примеру, что у Terraform следующая конфигурация:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Ниже показан небольшой фрагмент файла terraform.tfstate (урезанный, чтобы его было легче читать), который будет создан после выполнения terraformapply:

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
```

```
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafe1f0",
            "availability_zone": "us-east-2c",
            "id": "i-00d689a0acc43af0f",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```


Благодаря формату JSON Terraform знает, что ресурс типа aws_instance с именем example соответствует серверу EC2 с идентификатором i-00d689a0acc43af0f в вашей учетной записи AWS. При каждом запуске Terraform может запросить у AWS текущее состояние этого сервера и сравнить его с вашей конфигурацией, чтобы определить, какие изменения следует внести. Иными словами, вывод команды plan — это расхождение между кодом на вашем компьютере и инфраструктурой, развернутой в реальном мире (согласно идентификаторам в файле состояния).

Файл состояния является приватным API

Файл состояния — это приватный API, который меняется с каждым новым выпуском и предназначен сугубо для внутреннего использования в

Terraform. Вы никогда не должны редактировать его вручную или считывать его напрямую.

Если вам по какой-то причине нужно модифицировать файл состояния (что должно быть редкостью), используйте команды terraform import или terraform state (примеры работы с ними показаны в главе 5).

Если вы используете Terraform в личном проекте, можно спокойно хранить файл terraform.tfstate локально на своем компьютере. Но если вы ведете командную разработку реального продукта, может возникнуть несколько проблем.

•*Общее хранилище для файлов состояния*. Чтобы обновлять инфраструктуру с помощью Terraform, у каждого члена команды должен быть доступ к одним и тем же файлам состояния. Это означает, что вам нужно хранить эти файлы в общедоступном месте.

• *Блокирование файлов состояния*. Разделение данных сразу же создает новую проблему: блокирование. Если два члена команды запускают Terraform одновременно, может возникнуть состояние гонки, так как обновление файлов состояния происходит параллельно со стороны двух разных процессов. Без блокирования это может привести к конфликтам, потере данных и повреждению файлов состояния.

•*Изоляция файлов состояния*. При изменении инфраструктуры рекомендуется изолировать разные окружения. Например, при правке состояния в среде предварительного или финального тестирования следует убедиться, что это никак не навредит промышленной системе. Но как изолировать изменения, если вся инфраструктура описана в одном и том же файле состояния Terraform?

В следующих разделах мы подробно исследуем все эти проблемы и посмотрим, как их решить.

Общее хранилище для файлов состояния

Самый распространенный метод, который позволяет нескольким членам команды работать с общим набором файлов, заключается в использовании системы управления версиями (например, Git). Хотя ваш код Terraform точно должен храниться именно таким образом, применение того же подхода к состоянию Terraform — *плохая идея* по нескольким причинам.

•*Человеческий фактор*. Вы можете легко забыть загрузить последние изменения из системы управления версиями перед запуском Terraform или сохранить свои собственные обновления постфактум. Рано или поздно кто-то в вашей команде случайно запустит Terraform с устаревшими файлами состояния, что приведет к откату или дублированию уже развернутых ресурсов.

• *Блокирование*. Большинство систем управления версиями не предоставляют никаких средств блокирования, которые могли бы предотвратить одновременное выполнение terraformapply двумя разными членами команды.

•*Наличие конфиденциальных данных*. Все данные в файлах состояния Terraform хранятся в виде обычного текста. Это чревато проблемами, поскольку некоторым ресурсам Terraform необходимо хранить чувствительные данные. Например, если вы создаете базу данных с помощью ресурса aws_db_instance, Terraform сохранит имя пользователя и пароль к ней в файле состояния в открытом виде. Открытое хранение конфиденциальных данных *где бы то ни было*, включая систему управления версиями, — плохая идея. По состоянию на май 2019 года сообщество Terraform обсуждает открытую заявку (http://bit.ly/33gqaVe), созданную по этому поводу, хотя для решения данной проблемы есть обходные пути, которые мы вскоре обсудим.

Вместо системы управления версиями для совместного управления файлами состояния лучше использовать удаленные хранилища, поддержка которых встроена в Terraform. *Хранилище* определяет то, как Terraform загружает и сохраняет свое состояние. По умолчанию для этого применяется *локальное хранилище*, с которым вы работали все это время. Оно хранит файлы состояния на вашем локальном диске. Но поддерживаются также *удаленные* хранилища с возможностью совместного доступа. Среди них можно выделить Amazon S3, Azure Storage, Google Cloud Storage и такие продукты, как Terraform Cloud, Terraform Pro и Terraform Enterprise от HashiCorp.

Удаленные хранилища решают все три проблемы, перечисленные выше.

•*Человеческий фактор*. После конфигурации удаленного хранилища Terraform будет автоматически загружать из него файл состояния при каждом выполнении команд plan и apply и аналогично сохранять его туда после выполнения apply. Таким образом, возможность ручной ошибки исключается.

• *Блокирование*. Большинство удаленных хранилищ имеют встроенную поддержку блокирования. При выполнении terraformapply Terraform автоматически устанавливает блокировку. Если в этот момент данную команду выполняет кто-то другой, блокировка уже установлена и вам придется подождать. Команду apply можно ввести с параметром -lock-timeout=<TIME>. Так Terraform будет знать, сколько времени нужно ждать снятия блокировки (например, если указать -lock-timeout=10m, ожидание будет продолжаться десять минут).

•*Конфиденциальные данные*. Большинство удаленных хранилищ имеют встроенную поддержку активного и пассивного шифрования файлов состояния. Более того, они обычно позволяют настраивать права доступа (например, при использовании политик IAM в сочетании с бакетом Amazon S3), чтобы вы могли управлять тем, кто может обращаться к вашим файлам состояния и конфиденциальным данным, которые могут в них находиться. Конечно, было бы лучше, если бы в Terraform поддерживалось шифрование конфиденциальных данных прямо в файлах состояния, но эти удаленные хранилища минимизируют большинство рисков безопасности (файл состояния не хранится в открытом виде где-нибудь на вашем диске).

Если вы используете Terraform в связке с AWS, лучшим выбором в качестве удаленного хранилища будет S3, управляемый сервис хранения файлов от Amazon. Этому есть несколько причин.

• Это управляемый сервис, поэтому для его использования не нужно развертывать и обслуживать дополнительную инфраструктуру.

• Он рассчитан на 99,999999999%-ную устойчивость и 99,99%-ную доступность. Это означает, что вам не стоит сильно волноваться о потере данных и перебоях в работе34.

• Он поддерживает шифрование, что снижает риск хранения чувствительных данных в файлах состояния. Хотя это лишь частичное решение, так как любой член вашей команды с доступом к бакету S3 сможет просматривать эти файлы в открытом виде, но так данные будут шифроваться при сохранении (Amazon S3 поддерживает шифрование на серверной стороне с помощью AES-256) и передаче (Terraform использует SSL для чтения и записи данных в Amazon S3).

• Он поддерживает блокирование с помощью DynamoDB (подробнее об этом —чуть позже).

• Он поддерживает *управление версиями*, поэтому вы сможете хранить каждую ревизию своего состояния и в случае возникновения проблем откатываться на более старую версию.

• Он недорогой, поэтому большинство сценариев применения Terraform легко вписываются в бесплатный тарифный план35.

Чтобы включить удаленное хранение состояния в Amazon S3, для начала нужно подготовить бакет S3. Создайте файл main.tf в новой папке (это не должна быть папка, в которой вы храните конфигурацию из главы 2) и вверху укажите AWS в качестве провайдера:

```
provider "aws" {
  region = "us-east-2"
}
```

Затем создайте бакет S3, используя ресурс aws_s3_bucket:

```
resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-up-and-running-state"
  # Предотвращаем случайное удаление этого бакета S3
  lifecycle {
    prevent_destroy = true
  }
  # Включаем управление версиями, чтобы вы могли просматривать
  # всю историю ваших файлов состояния
  versioning {
    enabled = true
  }
  # Включаем шифрование по умолчанию на стороне сервера
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

Этот код устанавливает четыре аргумента.

•bucket. Это имя бакета S3. Имейте в виду, что имена бакетов должны быть уникальными на *глобальном* уровне среди всех клиентов AWS. Поэтому вместо "terraform-up-and-running-state" вы должны подобрать свое собственное название (так как бакет с этим именем я уже создал36). Запомните его и обратите внимание на то, какой регион AWS вы используете: чуть позже вам понадобятся оба эти фрагмента информации.

• prevent_destroy. Это второй параметр жизненного цикла, с которым вы сталкиваетесь (первым был create_before_destroy в главе 2). Если присвоить ему true, при попытке удаления соответствующего ресурса (например, при выполнении terraformdestroy) Terraform вернет ошибку. Это позволяет предотвратить случайное удаление важных ресурсов, таких как бакет S3 со всем вашим состоянием Terraform. Конечно, если вы действительно хотите его удалить, просто закомментируйте этот параметр.

• versioning. В данном разделе включается управления версиями в бакете S3, в результате чего каждое обновление хранящегося в нем файла будет создавать его новую версию. Это позволяет просматривать старые версии и откатываться к ним в любой момент.

•server_side_encryption_configuration. В этом разделе включается шифрование по умолчанию на стороне сервера для всех данных, которые

записываются в бакет S3. Благодаря этому ваши файлы состояния и любые конфиденциальные данные, которые могут в них содержаться, всегда будут шифроваться при сохранении в S3.

Далее нужно создать таблицу DynamoDB, которая будет использоваться для блокирования. DynamoDB — это распределенное хранилище типа «ключ — значение» от Amazon. Оно поддерживает строго согласованное чтение и условную запись — все, что необходимо для распределенной системы блокирования. Более того, оно полностью управляемое, поэтому вам не нужно заниматься никакой дополнительной инфраструктурой, а большинство сценариев применения Terraform легко впишутся в бесплатный тарифный план37.

Чтобы использовать DynamoDB для блокирования в связке с Terraform, нужно создать таблицу с первичным ключом под названием LockID (с *аналогичным* написанием). Это можно сделать с помощью ресурса aws_dynamodb_table:

```
resource "aws_dynamodb_table" "terraform_locks" {
  name         = "terraform-up-and-running-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

Выполните terraforminit, чтобы загрузить код провайдера, а затем terraformapply, чтобы развернуть ресурсы. Примечание: чтобы иметь возможность развертывать этот код, у вашего пользователя IAM должны быть права на создание бакетов S3 и таблиц DynamoDB, как описано в разделе «Подготовка вашей учетной записи в AWS» на с. 60. После завершения развертывания вы получите бакет S3 и таблицу DynamoDB, но ваше состояние Terraform по-прежнему будет храниться локально. Чтобы хранить его в бакете S3 (с шифрованием и блокированием), нужно добавить в свой код раздел backend. Это конфигурация самой системы Terraform, поэтому она находится внутри блока terraform и имеет следующий синтаксис:

```
terraform {
  backend "<BACKEND_NAME>" {
    [CONFIG...]
  }
}
```

BACKEND_NAME — это имя хранилища, которое вы хотите использовать (например, "s3"), а CONFIG содержит один или несколько аргументов, предусмотренных специально для этого хранилища (скажем, имя бакета S3, который нужно использовать). Так выглядит конфигурация backend для бакета S3:

```
terraform {
  backend "s3" {
    # Поменяйте это на имя своего бакета!
    bucket         = "terraform-up-and-running-state"
    key            = "global/s3/terraform.tfstate"
    region         = "us-east-2"
    # Замените это именем своей таблицы DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Пройдемся по этим параметрам.

•bucket. Имя нужного бакета S3. Не забудьте поменять его на название созданного ранее бакета.

• key. Файловый путь внутри бакета S3, по которому Terraform будет записывать файл состояния. Позже вы увидите, почему в предыдущем примере этому параметру присвоено global/s3/terraform.tfstate.

• region. Регион AWS, в котором находится бакет S3. Не забудьте указать регион того бакета, который вы создали ранее.

• dynamodb_table. Таблица DynamoDB, которая будет использоваться для блокирования. Не забудьте указать имя той таблицы, которую вы создали ранее.

•encrypt. Если указать true, состояние Terraform будет шифроваться при сохранении в S3. Это дополнительная мера, которая гарантирует шифрование данных во всех ситуациях, так как мы уже включили шифрование по умолчанию для S3.

Чтобы состояние Terraform сохранялось в этом бакете, нужно опять выполнить terraforminit. Эта команда не только загрузит код провайдера, но и сконфигурирует хранилище Terraform (еще одно ее применение вы увидите чуть позже). Более того, она идемпотентная, поэтому ее повторное выполнение безопасно:

```
$ terraform init
Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
```

Pre-existing state was found while migrating the previous "local" backend to the newly configured "s3" backend. No existing state was found in the newly configured "s3" backend. Do you want to copy this state to the new "s3" backend? Enter "yes" to copy and "no" to start with an empty state.
Enter a value:

Terraform автоматически определит, что у вас уже есть локальный файл состояния, и с вашего позволения скопирует его в новое хранилище S3. Если ввести yes, можно увидеть следующее:

Successfully configured the backend "s3"! Terraform will automatically use this backend unless the backend configuration changes.

После выполнения этой команды ваше состояние Terraform будет сохранено в бакете S3. Чтобы в этом убедиться, откройте консоль управления S3 (https://amzn.to/ 2Kw5qAc) в своем браузере и выберите свой бакет. Вы должны увидеть нечто похожее на рис. 3.1.

После включения этого хранилища Terraform будет автоматически загружать последнее состояние из бакета S3 перед выполнением команды и сохранять его туда после того, как команда будет выполнена. Чтобы увидеть, как это работает, добавьте следующие выходные переменные:

```
output "s3_bucket_arn" {
  value       = aws_s3_bucket.terraform_state.arn
  description = "The ARN of the S3 bucket"
}
output "dynamodb_table_name" {
  value       = aws_dynamodb_table.terraform_locks.name
  description = "The name of the DynamoDB table"
}
```

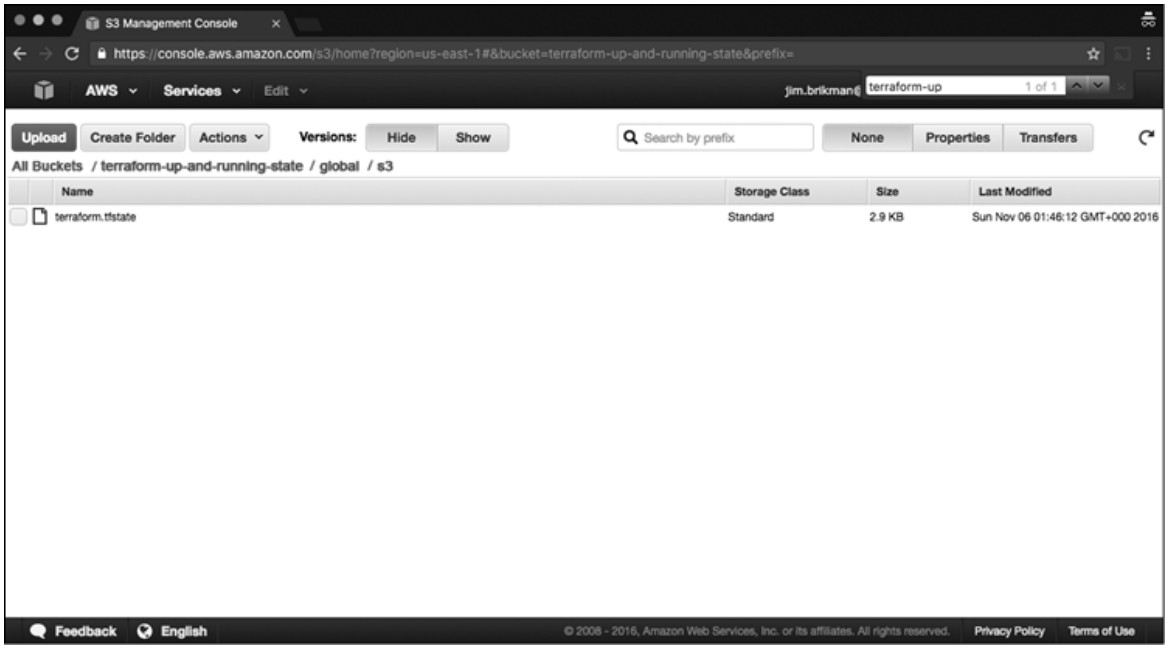


Рис. 3.1. Файл состояния Terraform, хранящийся в S3

Эти переменные выведут на экран ARN (Amazon Resource Name) вашего бакета S3 и имя вашей таблицы DynamoDB. Чтобы в этом убедиться, выполните terraformapply:

```
$ terraform apply
Acquiring state lock. This may take a few moments...
aws_dynamodb_table.terraform_locks: Refreshing state...
aws_s3_bucket.terraform_state: Refreshing state...
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...
Outputs:
dynamodb_table_name = terraform-up-and-running-locks
s3_bucket_arn = arn:aws:s3:::terraform-up-and-running-state
```

Заметьте, что теперь Terraform устанавливает блокировку перед запуском команды apply и снимает ее после!

Еще раз зайдите в консоль S3 по адресу https://amzn.to/2Kw5qAc, обновите страницу и нажмите серую кнопку Show (Показать) рядом с надписью Versions (Версии). На экране должно появиться несколько версий вашего файла terraform.tfstate, хранящегося в бакете S3 (рис. 3.2).

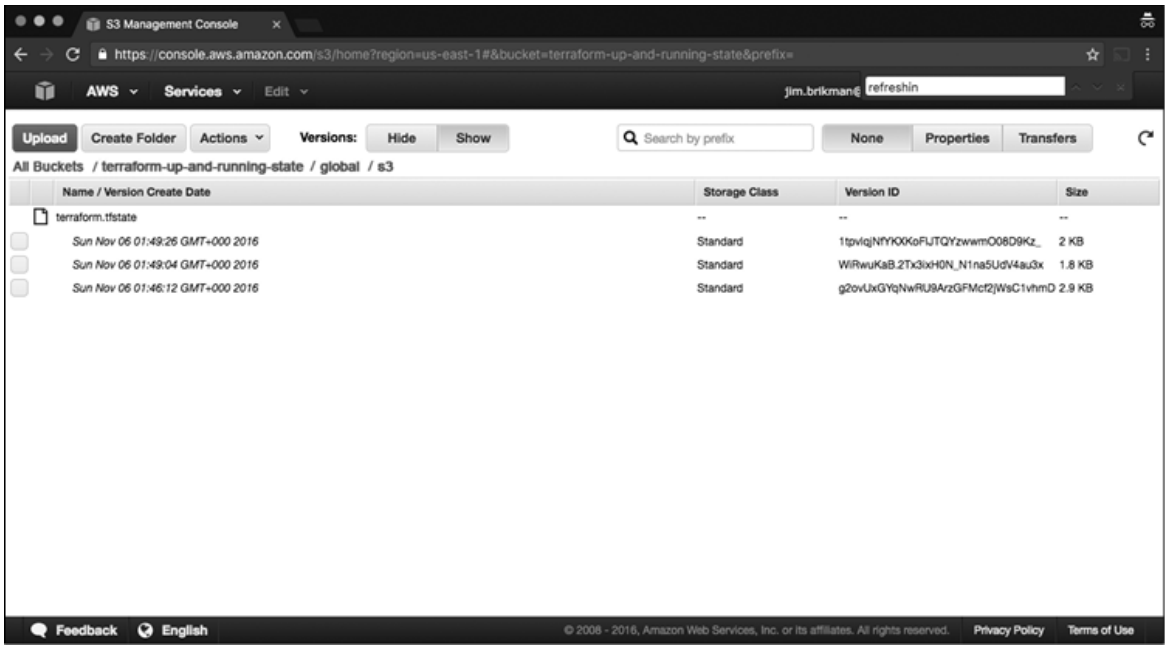


Рис. 3.2. Несколько версий состояния Terraform в S3

Это означает, что Terraform действительно загружает данные состояния в S3 и из него и ваш бакет хранит каждую ревизию файла состояния, что может пригодиться для отладки и отката к более старой версии, если что-то пойдет не так.

Ограничения хранилищ Terraform

У хранилищ Terraform есть несколько ограничений и подводных камней, о которых вам следует знать. Прежде всего, когда вы используете Terraform для создания бакета S3, в котором вы хотите хранить состояние Terraform, это похоже на ситуацию с курицей и яйцом. Чтобы этот подход работал, вам пришлось выполнить следующее.

1. Написать код Terraform, чтобы создать бакет S3 и таблицу DynamoDB, а затем развернуть этот код с использованием локального хранилища.

2. Вернуться к коду Terraform, добавить в него конфигурацию для удаленного хранилища, чтобы применить свежесозданные бакет S3 и таблицу DynamoDB, и выполнить команду terraforminit, чтобы скопировать ваше локальное состояние в S3.

Если вы когда-нибудь захотите удалить бакет S3 и таблицу DynamoDB, придется выполнить обратные действия.

1. Перейти к коду Terraform, удалить конфигурацию backend и снова выполнить команду terraforminit, чтобы скопировать состояние Terraform обратно на локальный диск.

2. Выполнить terraformdestroy, чтобы удалить бакет S3 и таблицу DynamoDB.

Этот двухступенчатый процесс довольно непростой, зато вы можете использовать во всем своем коде Terraform одни и те же бакет S3 и таблицу DynamoDB, поэтому нужно выполнить его лишь раз (или единожды для каждой учетной записи AWS, если у вас их несколько). Если у вас уже есть бакет S3, вы можете сразу указывать конфигурацию backend в своем коде Terraform без дополнительных действий.

Второе ограничение более болезненное: в разделе backend в Terraform нельзя применять никакие переменные или ссылки. Следующий код *не* будет работать.

```
# Это НЕ будет работать. В конфигурации хранилища нельзя использовать переменные.
terraform {
  backend "s3" {
    bucket         = var.bucket
    region         = var.region
    dynamodb_table = var.dynamodb_table
    key            = "example/terraform.tfstate"
    encrypt        = true
  }
}
```

Значит, необходимо вручную копировать и вставлять имя и регион бакета S3, а также название таблицы DynamoDB в каждый ваш модуль Terraform. Вы подробно познакомитесь с модулями Terraform в главах 4 и 6. Пока достаточно понимать, что модули — это способ организации и повторного использования кода Terraform и что настоящий код Terraform обычно состоит из множества мелких модулей. Что еще хуже, вам нужно быть очень осторожными, чтобы *не* скопировать значение key. Чтобы разные модули случайно не перезаписывали состояние друг друга, это значение должно быть уникальным для каждого из них! Частое копирование *и* ручное редактирование чреваты ошибками, особенно если нужно развертывать и администрировать множество модулей Terraform во многих средах.

По состоянию на май 2019 года единственным решением является использование *частичной конфигурации*, в которой можно опустить определенные параметры раздела backend и передавать их вместо этого в аргументе командной строки -backend-config при вызове terraforminit. Например, вы можете вынести повторяющиеся параметры хранилища, такие как bucket и region, в отдельный файл под названием backend.hcl:

```
# backend.hcl
```

```
bucket         = "terraform-up-and-running-state"
region         = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt        = true
```

В коде Terraform останется только параметр key, поскольку вам все равно нужно устанавливать ему разные значения в разных модулях:

```
# Частичная конфигурация. Другие параметры (такие как bucket, region) будут
# переданы команде 'terraform init' в виде файла с использованием
# аргументов -backend-config
terraform {
  backend "s3" {
    key = "example/terraform.tfstate"
  }
}
```

Чтобы собрать воедино все фрагменты вашей конфигурации, выполните команду terraforminit с аргументом -backend-config:

```
$ terraform init -backend-config=backend.hcl
```

Terraform объединит частичную конфигурацию из файла backend.hcl и вашего кода Terraform, чтобы получить полный набор параметров для вашего модуля.

Еще один вариант заключается в применении Terragrunt, инструмента с открытым исходным кодом, который пытается компенсировать то, чего не хватает в Terraform. Terragrunt может помочь избежать дублирования базовых параметров хранилища (имя и регион бакета, имя таблицы DynamoDB) за счет определения их в едином файле и автоматического применения относительного файлового пути модуля в качестве значения key. Пример с Terragrunt будет показан в главе 8.

Изоляция файлов состояния

Благодаря удаленным хранилищам и блокированию совместная работа больше не проблема. Но одна проблема у нас все же остается: изоляция. Когда вы начинаете использовать Terraform, может появиться соблазн описать всю свою инфраструктуру в одном файле или едином наборе файлов в одной папке. Недостаток этого подхода в том, что в одном файле хранится не только код, но и состояние Terraform и, чтобы все сломать, достаточно одной ошибки в любом месте.

Например, при попытке развертывания новой версии своего приложения в среде финального тестирования вы можете нарушить его работу в промышленных условиях. Или еще хуже, вы можете повредить

весь файл состояния (скажем, из-за отсутствия блокирования либо из-за редкой программной ошибки в Terraform), в результате чего ваша инфраструктура выйдет из строя во всех средах38.

Весь смысл поддержки нескольких сред состоит в том, что они изолированы друг от друга, поэтому, если вы управляете всеми ими из одного набора конфигурационных файлов Terraform, вы нарушаете эту изоляцию. По аналогии с переборками, которые не дают затопить все секции судна из-за протечки в одной из них, вы должны предусмотреть «переборки» для своей архитектуры Terraform (рис. 3.3).

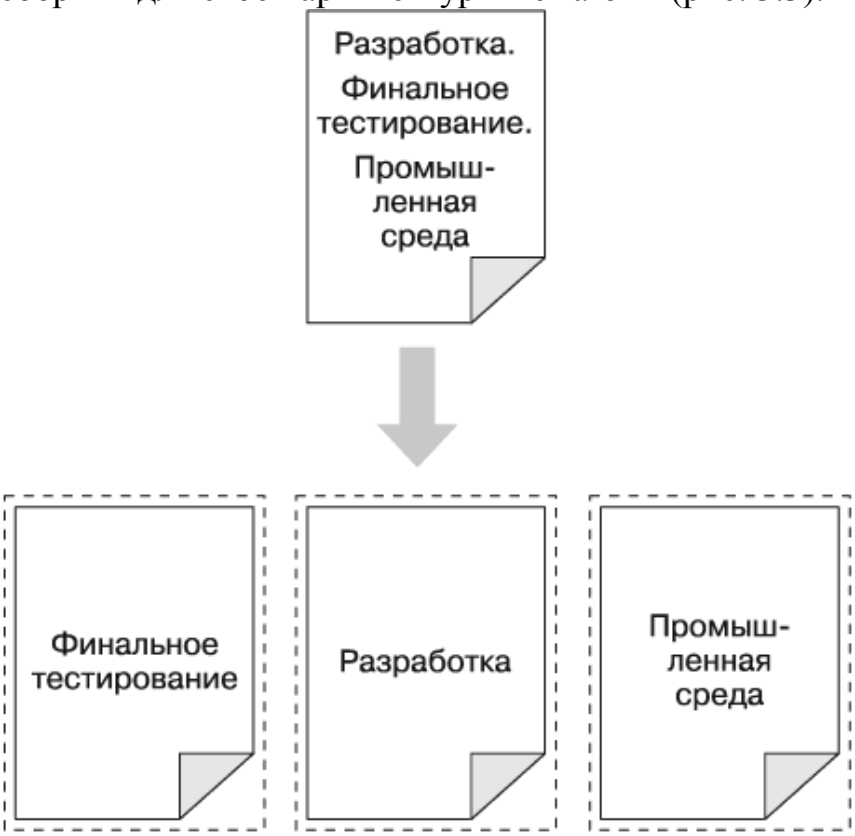


Рис. 3.3. Добавление «переборок» в архитектуру Terraform

Как проиллюстрировано на рис. 3.3, мы описываем все наши среды не в одном наборе конфигурационных файлов *(вверху)*, а в разных наборах *(внизу)*, поэтому проблема в одной среде полностью изолирована от других. Файлы состояния можно изолировать двумя способами.

•*Изоляция через рабочие области*. Подходит для быстрых изолированных проверок с одной и той же конфигурацией.

• *Изоляция с помощью описания структуры файлов*. Подходит для промышленного использования, когда требуется строгая изоляция между средами.

Изоляция через рабочие области

Вы можете хранить свое состояние Terraform в нескольких отдельных именованных *рабочих областях*. У Terraform изначально одна рабочая область, которая используется по умолчанию. Чтобы создать новую рабочую область или переключиться между областями, нужно выполнить команду terraformworkspace. Поэкспериментируем с неким кодом Terraform, который развертывает сервер EC2:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Сконфигурируйте для этого сервера хранилище на основе бакета S3 и таблицы DynamoDB, которые вы создали ранее в этой главе, но в качестве значения для key укажите workspaces-example/terraform.tfstate:

```
terraform {
  backend "s3" {
    # Поменяйте это на имя своего бакета!
    bucket = "terraform-up-and-running-state"
    key    = "workspaces-example/terraform.tfstate"
    region = "us-east-2"
    # Замените это именем своей таблицы DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Выполните команды terraforminit и terraformapply, чтобы развернуть этот код:

```
$ terraform init
Initializing the backend...
Successfully configured the backend "s3"! Terraform will automatically use this backend unless the backend configuration changes.
Initializing provider plugins...
(...)
Terraform has been successfully initialized!
$ terraform apply
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Состояние этого развертывания хранится в рабочей области по умолчанию. В этом можно убедиться с помощью команды

terraformworkspaceshow, которая показывает, в какой рабочей области вы сейчас находитесь:

$ terraform workspace show

default

Рабочая область по умолчанию хранит ваше состояние именно в том месте, которое вы указали в параметре key. Как видно на рис. 3.4, взглянув на свой бакет S3, вы увидите файл terraform.tfstate и папку workspaces-example.

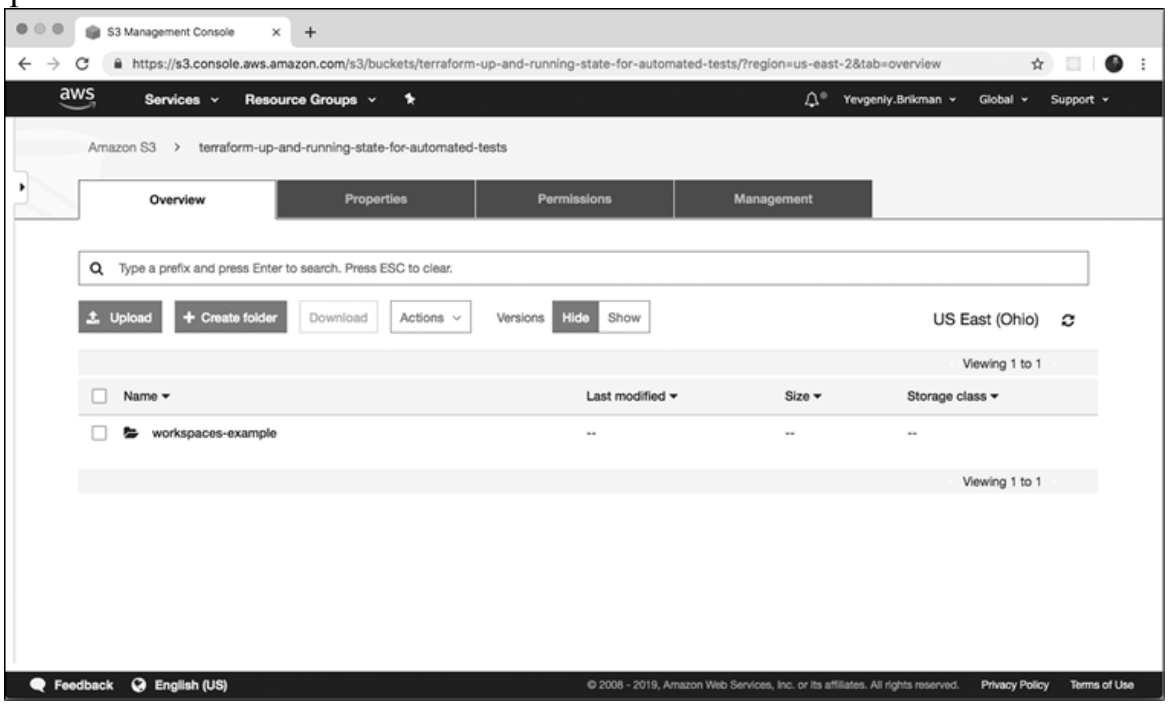


Рис. 3.4. Бакет S3 после сохранения состояния в рабочей области по умолчанию

Создадим новую рабочую область над названием example1, используя команду terraformworkspacenew:

$ terraform workspace new example1

Created and switched to workspace "example1"!

You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

Теперь посмотрите, что получится, если мы попытаемся выполнить terraformplan:

$ terraform plan

Terraform will perform the following actions:

aws_instance.example will be created

+ resource "aws_instance" "example" {

+ ami = "ami-0c55b159cbfafe1f0"

```
  + instance_type          = "t2.micro"
  (...)
}
```

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

Terraform хочет создать с нуля совершенно новый сервер EC2! Дело в том, что файлы состояния в каждой рабочей области изолированы друг от друга и, поскольку вы теперь в рабочей области example1, Terraform больше не использует файл состояния из рабочей области по умолчанию. Следовательно, не видит сервер EC2, который был там создан.

Попробуйте выполнить команду terraformapply, чтобы развернуть этот второй сервер EC2 в новой рабочей области:

```
$ terraform apply
(...)
```

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

Повторим этот пример еще один раз и создадим новую рабочую область под названием example2:

```
$ terraform workspace new example2
```

Created and switched to workspace "example2"!

You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

Снова выполните terraformapply, чтобы развернуть третий сервер EC2:

```
$ terraform apply
(...)
```

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

Теперь у вас есть три рабочие области; в этом можно убедиться с помощью команды terraformworkspacelist:

```
$ terraform workspace list
  default
  example1
* example2
```

Вы можете переключиться между ними в любой момент, используя команду terraformworkspaceselect:

```
$ terraform workspace select example1
```

Switched to workspace "example1".

Чтобы понять, как это работает внутри, еще раз загляните в свой бакет. Вы должны увидеть новую папку под названием env:, как показано на рис. 3.5.

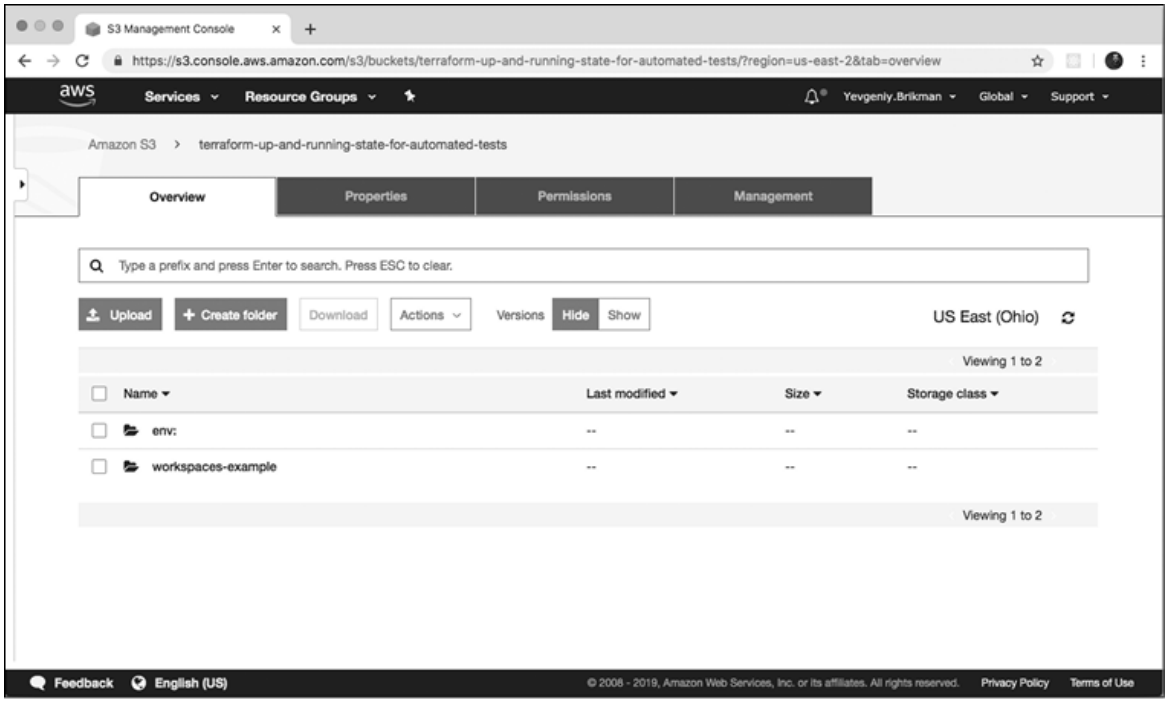


Рис. 3.5. Бакет S3 после того, как вы начали использовать собственные рабочие области

Внутри env: вы найдете по одной папке для каждой из ваших рабочих областей (рис. 3.6).

Внутри каждой из этих рабочих областей Terraform использует значение key, которое вы указали в конфигурации хранилища, поэтому вы должны найти файлы example1/workspaces-example/terraform.tfstate и example2/workspaces-example/terraform.tfstate. Иными словами, переключение на другую рабочую область равнозначно изменению пути хранения вашего файла состояния.

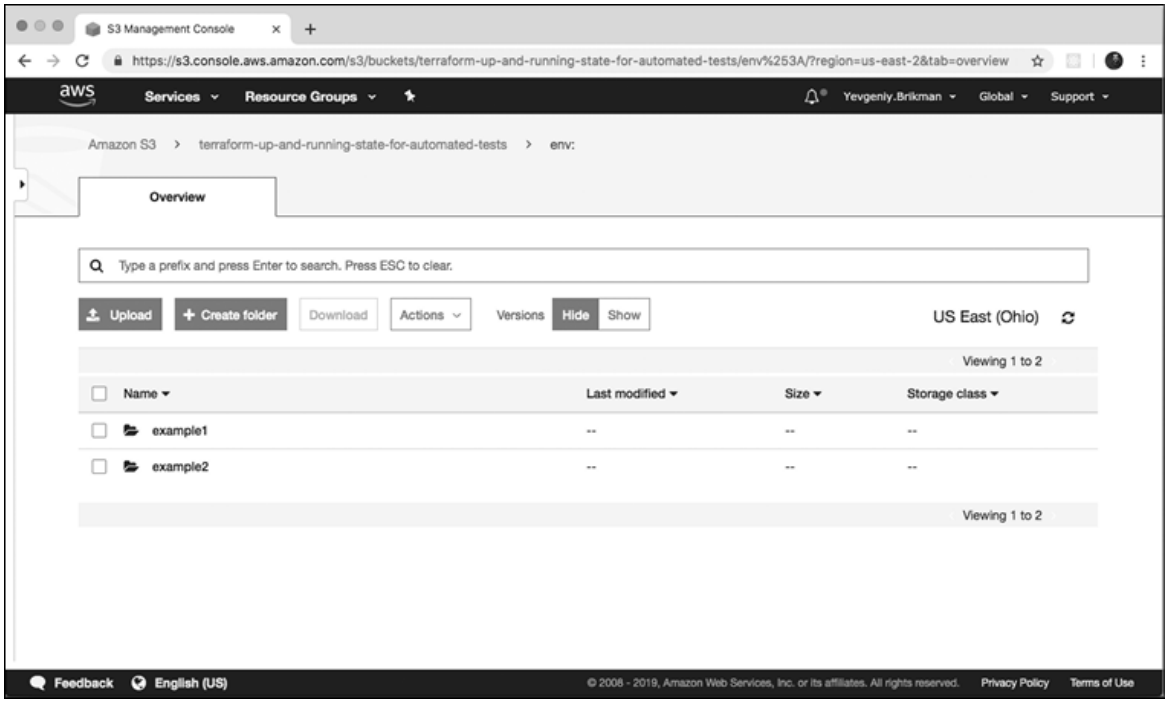


Рис. 3.6. Terraform создает по одной папке для каждой рабочей области

Это удобно, когда вы уже развернули модуль Terraform и хотите с ним поэкспериментировать (например, попробовать изменить структуру кода), но так, чтобы ваши эксперименты не отразились на состоянии уже развернутой инфраструктуры. Вы можете выполнить команду terraformworkspacenew и развернуть новую копию той же инфраструктуры, но с отдельным файлом состояния.

Вы можете даже сделать так, чтобы этот модуль менял свое поведение в зависимости от того, в какой рабочей области вы находитесь. Для этого он может считывать имя рабочей области с помощью выражения terraform.workspace. Например, в рабочей области по умолчанию можно указать тип сервера EC2 t2.medium, а во всех остальных областях — t2.micro (чтобы, скажем, сделать свои эксперименты более экономными):

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
    instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"
}
```

Этот код использует *тернарный синтаксис*, чтобы присвоить параметру instance_type значение t2.medium либо t2.micro в зависимости от значения terraform.workspace. Все подробности о тернарном синтаксисе и условной логике в Terraform изложены в главе 5.

Рабочие области Terraform отлично подходят для быстрого развертывания и удаления разных версий вашего кода, но у них есть несколько недостатков.

• Файлы состояния всех ваших рабочих областей находятся в одном и том же хранилище (например, в одном бакете S3). Это означает, что все они используют аналогичные механизмы аутентификации и управления доступом, что является одной из основных причин, почему они не подходят для изоляции разных окружений, таких как среды для финального тестирования и промышленного применения.

• Рабочие области не видны в коде или терминале без использования команд terraformworkspace. При чтении кода невозможно сказать, в скольких рабочих областях развернут модуль: в одной или десяти, так как выглядят они идентично. Это усложняет обслуживание, поскольку у вас нет полноценного представления о вашей инфраструктуре.

• Из двух предыдущих пунктов вытекает тот факт, что рабочие области могут быть довольно сильно предрасположены к ошибкам. Из-за нехватки прозрачности можно легко забыть, в какой рабочей области вы находитесь, и внести изменения не в том месте (например, случайно выполнить команду terraformdestroy в рабочей области production вместо staging). Поскольку вам приходится использовать один и тот же механизм аутентификации для всех рабочих областей, у вас нет другого уровня защиты от подобных ошибок.

Чтобы как следует изолировать разные окружения, вместо рабочих областей лучше использовать структуру файлов и каталогов, о которой пойдет речь в следующем подразделе. Но прежде, чем двигаться дальше, не забудьте удалить три сервера EC2, которые вы только что развернули. Для этого выполните terraformworkspaceselect<имя> и terraformdestroy в каждой из трех рабочих областей.

Изоляция с помощью описания структуры файлов

Чтобы достичь полной изоляции между окружениями, нужно сделать следующее.

• Поместить конфигурационные файлы Terraform для каждой среды в отдельную папку. Например, вся конфигурация для среды финального тестирования может находиться в папке stage, а в папке prod может храниться конфигурация промышленной среды.

• Предусмотреть для каждой среды разные хранилища с разными механизмами аутентификации и управления доступом (предположим, у каждого окружения может быть отдельная учетная запись AWS со своим бакетом S3 в качестве хранилища).

Благодаря применению отдельных папок вам будет намного легче понять, в какой среде происходит развертывание, а использование отдельных файлов состояния с отдельными механизмами аутентификации значительно уменьшает вероятность того, что случайная оплошность в одной среде будет иметь какое-либо влияние на другие.

Концепцию изоляции лучше опустить на уровень ниже, вплоть до компонентов — наборов связанных между собой ресурсов, которые обычно развертываются совместно. Например, после настройки базовой сетевой топологии для своей инфраструктуры (в терминологии AWS это виртуальное частное облако (VPC) и все связанные с ним подсети, правила маршрутизации, VPN и сетевые списки доступа) вы будете менять ее не чаще раза в месяц. Но новые версии серверов могут развертываться по нескольку раз в день. Управляя инфраструктурой VPC и веб-сервера в одной и той же конфигурации Terraform, вы постоянно рискуете нарушить работу всей своей сетевой топологии (скажем, из-за простой опечатки в коде или случайного выполнения не той команды) без причины.

В связи с этим я рекомендую использовать отдельные папки Terraform (и, следовательно, отдельные файлы состояния) для каждого окружения (тестового, промышленного и т. д.) и каждого компонента (VPC, сервисов, баз данных). Чтобы посмотреть, как это выглядит на практике, разберем рекомендуемую структуру файлов и каталогов для проектов Terraform.

На рис. 3.7 показана типичная структура файлов в моих проектах.

На самом верхнем уровне находятся отдельные папки для каждого «окружения». Набор окружений зависит от проекта, но обычно он включает в себя следующее.

•stage — среда для предпромышленной нагрузки (тестирование).

• prod — среда для промышленной нагрузки (приложения, взаимодействующие с пользователями).

• mgmt — среда для инструментария DevOps (например, узел-бастион, Jenkins).

•global — папка с ресурсами, которые используются во всех средах (скажем, S3, IAM).

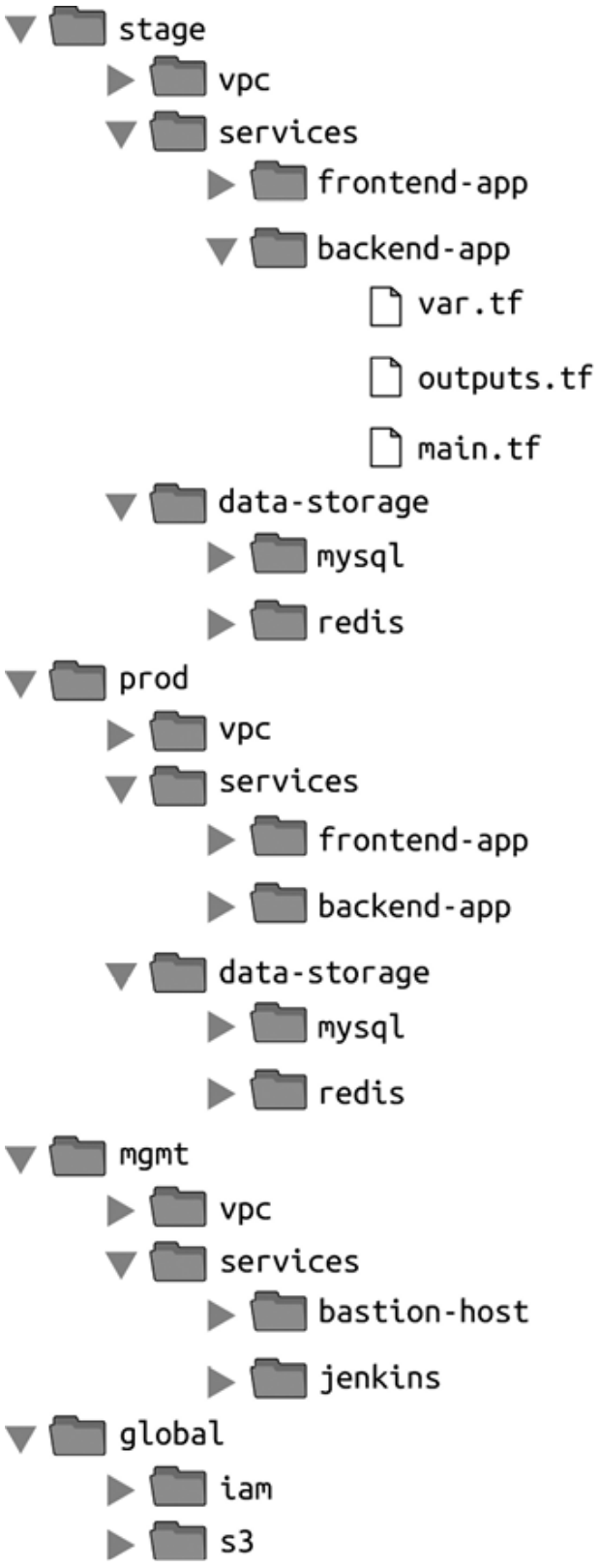


Рис. 3.7. Типичная структура файлов для проекта Terraform

Внутри каждого окружения есть отдельные папки для каждого компонента. Компоненты зависят от проекта, но обычно в их число входят следующие.

•vpc — сетевая топология для этой среды.

•services — приложения или микросервисы, которые запускаются в этой среде, например Ruby on Rails на стороне клиента или Scala на стороне сервера. Вы можете даже изолировать все приложения, поместив их в отдельные папки.

•data-storage — хранилища данных для этой среды, такие как MySQL или Redis. Хранилища можно изолировать друг от друга, разместив их в отдельных папках.

Внутри каждого компонента находятся конфигурационные файлы Terraform, которые названы по следующему принципу.

•variables.tf — входные переменные.

• outputs.tf — выходные переменные.

•main.tf — ресурсы.

При запуске Terraform просто ищет в текущей папке файлы с расширением .tf, поэтому вы можете называть свои файлы как угодно. Но, несмотря на это, вашим коллегам не все равно, какие названия файлов вы используете. Если задать последовательные и предсказуемые названия, это упростит просмотр кода. Вы всегда будете знать, где искать переменные, вывод или ресурсы. Если какой-то файл (особенно main.tf) становится слишком крупным, можете свободно вынести из него часть функциональности (и назвать ее, к примеру, iam.tf, s3.tf или database.tf), но это также может быть признаком того, что вам следует разбить свой код на более мелкие модули. Эту тему мы подробно рассмотрим в главе 4.

Как избежать дублирования кода

Структура файлов и каталогов, описанная в этом разделе, имеет много повторяющихся элементов. Например, одни и те же приложения frontend-app и backend-app находятся сразу в двух папках: stage и prod. Но не волнуйтесь, вам не придется копировать и вставлять весь этот код! В главе 4 вы увидите, как избегать дублирования и соблюдать принцип DRY с помощью модулей Terraform.

Возьмем код кластера с веб-сервером, который вы написали в главе 2, объединим его с кодом для Amazon S3 и DynamoDB из этой главы и

организуем это все в виде структуры каталогов, представленной на рис. 3.8.

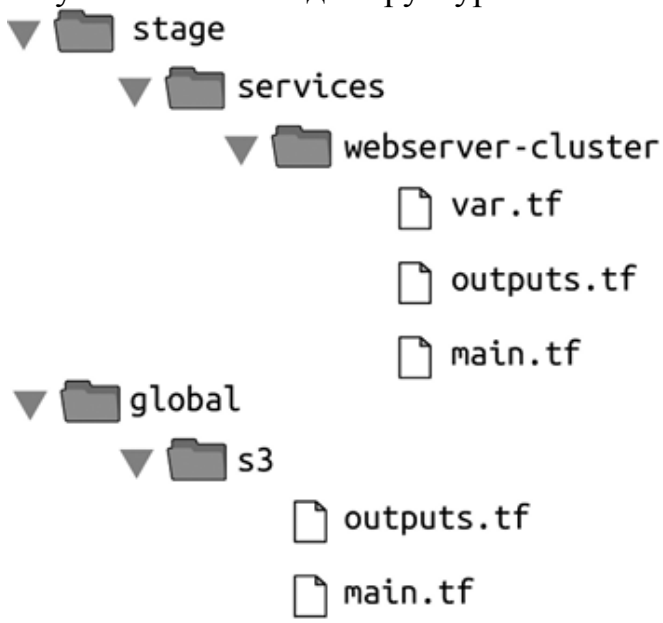


Рис. 3.8. Структура файлов для кода кластера с веб-сервером

Бакет S3, созданный вами в этой главе, необходимо переместить в папку global/s3. Поместите выходные переменные (s3_bucket_arn и dynamodb_table_name) в файл outputs.tf. При перемещении файлов в новое место убедитесь, что вы не забыли скопировать (скрытую) папку .terraform, иначе придется заново все инициализировать.

Кластер веб-серверов, который вы создали в главе 2, следует поместить в папку stage/services/webserver-cluster (своего рода тестовая версия этого кластера; промышленную мы добавим в следующей главе). Не забудьте скопировать папку .terraform и поместить входные и выходные переменные в файлы variables.tf и соответственно outputs.tf.

Вам также необходимо обновить кластер веб-серверов, чтобы он использовал S3 в качестве хранилища. Вы можете просто скопировать и вставить раздел backend из файла global/s3/main.tf без особых изменений, но не забудьте присвоить параметру key путь к папке, в которой находится код веб-сервера: stage/services/webserver-cluster/terraform.tfstate. Таким образом, код Terraform в системе управления версиями и структура ваших файлов состояния будут полностью совпадать и вам будет очевидно то, как они между собой связаны. Модуль s3 использует этот же принцип для установки параметра key.

Такая структура файлов и каталогов упрощает просмотр кода и помогает понять, какие именно компоненты развернуты в той или иной среде. Кроме того, она обеспечивает хорошую степень изоляции между окружениями и между компонентами внутри одного окружения. Благодаря

этому, если что-то пойдет не так, это коснется лишь небольшой части вашей инфраструктуры.

Конечно, это свойство в каком-то смысле является и недостатком: разделение компонентов на отдельные папки не дает случайной оплошности нарушить работу всей инфраструктуры, но при этом лишает вас возможности развертывать всю инфраструктуру одной командой. Если бы все компоненты для одного окружения были описаны в одном конфигурационном файле Terraform, вы бы могли запустить все это окружение с помощью единого вызова terraformapply. Но если все компоненты находятся в отдельных папках, нужно выполнить terraformapply в каждой из них. Стоит отметить, что в случае применения Terragrunt вы можете автоматизировать этот процесс с помощью команды apply-all39.

У этой структуры файлов есть еще одна проблема: она усложняет использование зависимостей ресурсов. Приложение имеет прямой доступ к атрибутам базы данных (то есть может сослаться на адрес базы данных через aws_db_instance.foo.address), если код этих двух компонентов находится в одних и тех же конфигурационных файлах Terraform. Но если код приложения и базы данных размещен в разных папках, как я советовал, эта возможность теряется. К счастью, Terraform предлагает решение этой проблемы: источник данных terraform_remote_state.

Источник данных terraform_remote_state

В главе 2 вы использовали источники данных для извлечения из AWS информации, доступной только для чтения. Например, источник aws_subnet_ids возвращал список подсетей вашего облака VPC. Но существует другой источник, terraform_remote_state, который особенно полезен при работе с состоянием. С его помощью можно извлечь файл состояния, который является частью другой конфигурации Terraform, и сделать это сугубо для чтения.

Рассмотрим пример. Представьте, что вашему кластеру веб-серверов необходимо взаимодействовать с базой данных MySQL. Обслуживание масштабируемой, безопасной, устойчивой и высокодоступной БД требует много усилий. Вы можете позволить Amazon позаботиться об этом с помощью сервиса RDS (Relational Database Service), как показано на рис. 3.9. RDS поддерживает разнообразные базы данных, включая MySQL, PostgreSQL, SQL Server и Oracle.

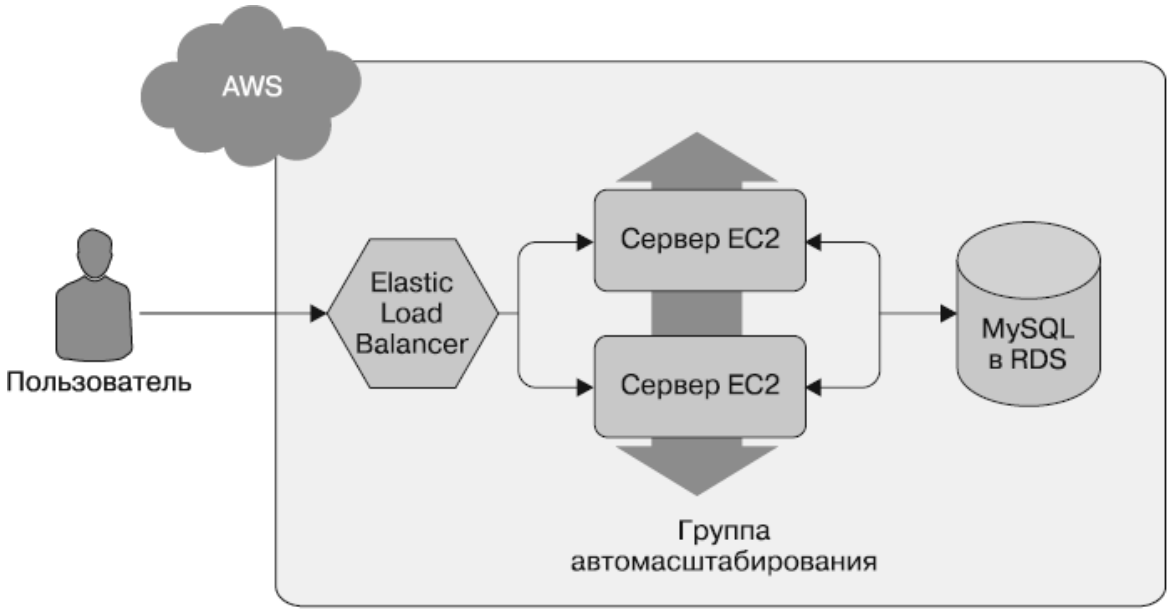


Рис. 3.9. Кластер веб-серверов взаимодействует с базой данных MySQL, развернутой поверх Amazon RDS

Базу данных MySQL лучше не объявлять в том же наборе конфигурационных файлов, что и кластер веб-серверов, потому что последний обновляется значительно чаще и вам вряд ли захочется рисковать при каждом таком обновлении. Первое, что вы должны сделать, — это создать новую папку stage/data-stores/mysql и поместить в нее три основных файла Terraform (main.tf, variables.tf, outputs.tf), как показано на рис. 3.10.

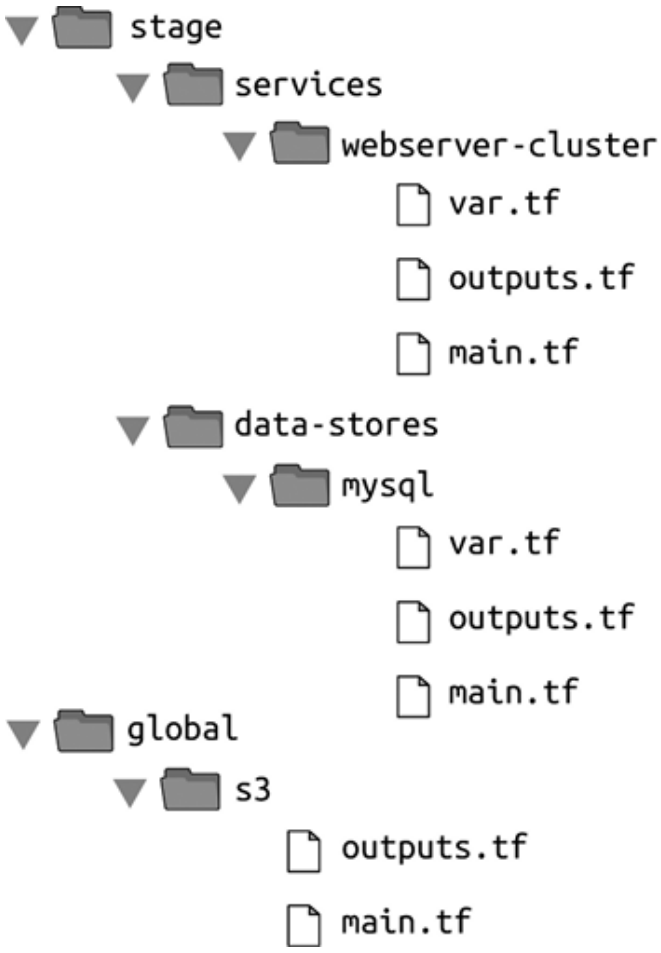


Рис. 3.10. Код базы данных в папке stage/data-stores

Вслед за этим создайте ресурс базы данных в файле stage/data-stores/mysql/main.tf:

```
provider "aws" {
  region = "us-east-2"
}
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  name                = "example_database"
  username            = "admin"
  # Как нам задать пароль?
  password            = "???"
}
```

Сразу под стандартным ресурсом provider в верхней части файла находится новый: aws_db_instance. Он создает базу данных в RDS.

Параметры в этом коде настраивают RDS для запуска MySQL с хранилищем размером 10 Гбайт на сервере db.t2.micro, который имеет один виртуальный процессор, 1 Гбайт памяти и входит в бесплатный тариф AWS.

Обратите внимание, что одним из параметров, которые вы должны передать ресурсу aws_db_instance, является главный пароль к базе данных. Поскольку он конфиденциален, его нельзя прописывать прямо в коде в виде обычного текста! Вместо этого можно воспользоваться одним из двух способов передачи конфиденциальных данных в ресурсы Terraform.

Первый способ для работы с конфиденциальными данными заключается в применении источника Terraform, который считывает их из секретного хранилища. Например, вы можете размещать такую информацию, как пароли к базе данных, в управляемом сервисе AWS Secrets Manager, предназначенном специально для хранения чувствительных данных. Вы можете воспользоваться его графическим интерфейсом, чтобы сохранить свой пароль, и затем прочитать его в своем коде Terraform с помощью источника данных aws_secretsmanager_secret_version:

```
provider "aws" {
  region = "us-east-2"
}
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine            = "mysql"
  allocated_storage = 10
  instance_class    = "db.t2.micro"
  name              = "example_database"
  username          = "admin"
  password =
    data.aws_secretsmanager_secret_version.db_password.secret_string
}
data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = "mysql-master-password-stage"
}
```

Вот несколько поддерживаемых комбинаций секретных хранилищ и источников данных, которые могут вас заинтересовать.

- AWS Secrets Manager и источник данных aws_secretsmanager_secret_version (предыдущий листинг).
- AWS Systems Manager Parameter Store и источник данных aws_ssm_parameter.
- AWS Key Management Service (AWS KMS) и источник данных aws_kms_secrets.

• Google Cloud KMS и источник данных google_kms_secret.
• Azure Key Vault и источник данных azurerm_key_vault_secret.
• HashiCorp Vault и источник данных vault_generic_secret.

Второй способ работы с конфиденциальными данными — полный вынос управления ими за пределы Terraform (например, вы можете делегировать это таким диспетчерам паролей, как 1Password, LastPass или OS X Keychain) и передача их в систему в виде переменных среды. Для этого нужно объявить переменную под названием db_password в файле stage/data-stores/mysql/variables.tf:

```
variable "db_password" {
  description = "The password for the database"
  type        = string
}
```

Обратите внимание на то, что у нее нет параметра default. Это сделано намеренно. Вы не должны хранить свой пароль к базе данных или любую чувствительную информацию в открытом виде. Вместо этого значение следует брать из переменной среды.

Напоминаю, что каждой входной переменной, определенной в конфигурации Terraform (как foo), можно предоставить значение, которое берется из переменной среды (вроде TF_VAR_foo). Для входной переменной db_password нужно установить переменную среды TF_VAR_db_password. Вот как это делается в системах Linux/Unix/OS X:

```
$ export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
$ terraform apply
(...)
```

Стоит отметить, что пробел перед командой export указан не случайно. Он нужен, чтобы ваши конфиденциальные данные не были сохранены на диск в истории bash40. Но есть лучший способ предотвратить случайную запись конфиденциальных данных на диск в открытом виде: хранить их в секретном хранилище, совместимом с командной строкой, таком как pass (https://www.passwordstore.org/), и безопасно считывать их оттуда в переменные среды с помощью дочерней командной оболочки:

```
$ export TF_VAR_db_password=$(pass database-password)
$ terraform apply
(...)
```

Конфиденциальные данные всегда хранятся в состоянии Terraform

Считывание конфиденциальных данных из секретного хранилища или переменных среды — хороший способ предотвратить их хранение в открытом виде внутри вашего кода. Но помните: вне зависимости от того, как вы их читаете, если они передаются в качестве аргумента в ресурс, такой как aws_db_instance, они автоматически сохраняются в состоянии Terraform в виде обычного текста.

Это известное слабое место Terraform, у которого нет эффективных решений. Поэтому будьте максимально бдительны с тем, как вы храните файлы состояния (например, всегда включайте шифрование) и кто имеет к ним доступ (скажем, ограничивайте доступ к своему бакету S3 с помощью привилегий IAM)!

Вслед за конфигурацией пароля нужно сделать так, чтобы модуль хранил свое состояние в бакете S3, который вы создали ранее в файле stage/data-stores/mysql/terraform.tfstate:

```
terraform {
  backend "s3" {
    # Поменяйте это на имя своего бакета!
    bucket         = "terraform-up-and-running-state"
    key            = "stage/data-stores/mysql/terraform.tfstate"
    region         = "us-east-2"
    # Замените это именем своей таблицы DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Выполните команды terraforminit и terraformapply, чтобы создать базу данных. Нужно учитывать, что на инициализацию даже небольшой базы данных в Amazon RDS может уйти около десяти минут, поэтому будьте терпеливы.

Итак, вы создали базу данных. Но как передать ее адрес и порт вашему кластеру веб-серверов? Для начала нужно добавить две выходные переменные в файл stage/data-stores/mysql/outputs.tf:

```
output "address" {
  value       = aws_db_instance.example.address
  description = "Connect to the database at this endpoint"
}
output "port" {
  value       = aws_db_instance.example.port
  description = "The port the database is listening on"
```

```
}
```

Выполните terraformapply еще раз, и в терминале должны появиться ваши выходные переменные:

```
$ terraform apply
(...)
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
address = tf-2016111123.cowu6mts6srx.us-east-2.rds.amazonaws.com
port = 3306
```

Теперь они хранятся и в состоянии Terraform для базы данных, которое находится в вашем бакете S3 в файле stage/data-stores/mysql/terraform.tfstate. Чтобы код вашего кластера веб-серверов прочитал содержимое этого файла состояния, добавьте в файл stage/services/webserver-cluster/main.tf источник данных terraform_remote_state:

```
data "terraform_remote_state" "db" {
  backend = "s3"
  config = {
    bucket = "(YOUR_BUCKET_NAME)"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-2"
  }
}
```

Благодаря этому источнику данных код кластера веб-серверов считывает файл состояния из тех же бакета S3 и папки, где свое состояние хранит база данных (рис. 3.11).

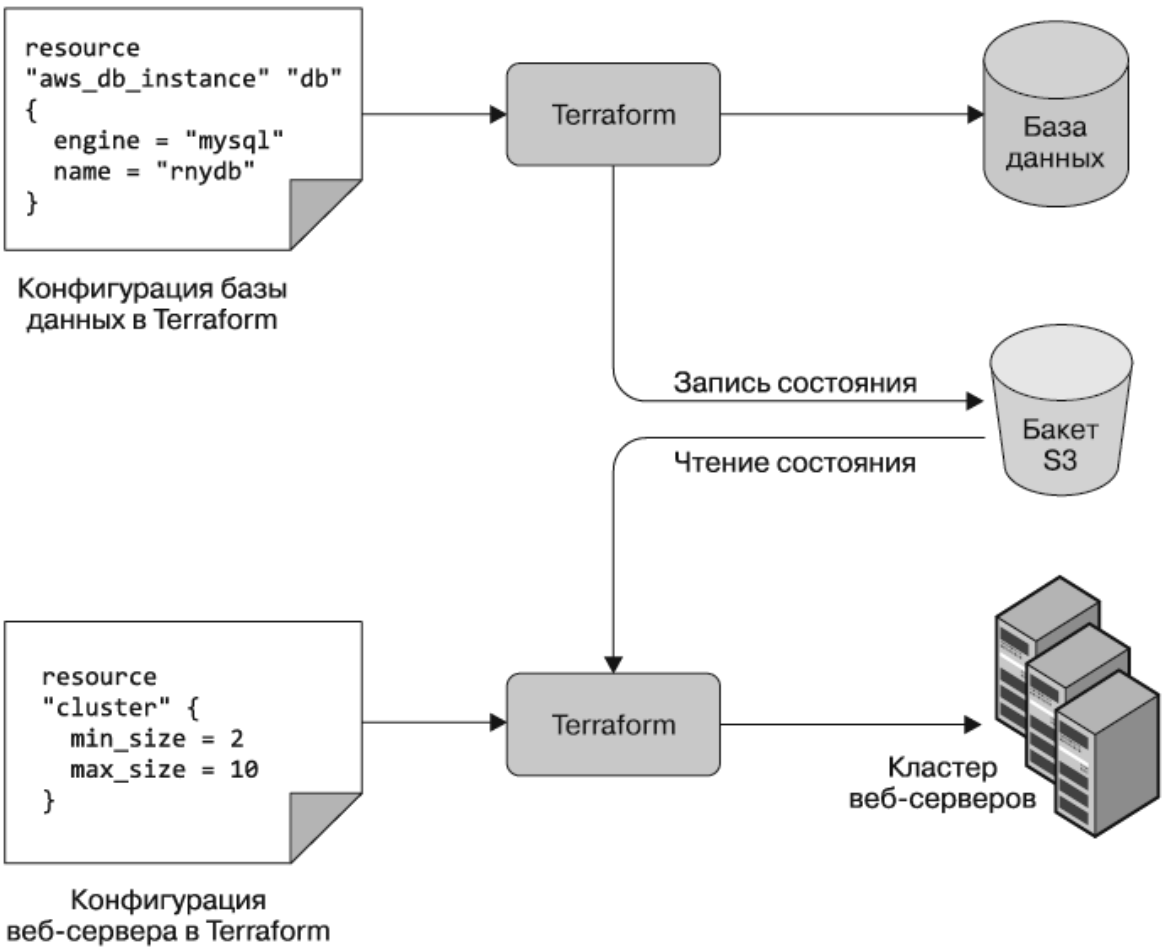


Рис. 3.11. База данных записывает свое состояние в бакет S3 *(вверху)*, а кластер веб-серверов считывает его из того же бакета *(внизу)*

Важно понимать, что, как и все источники данных в Terraform, terraform_remote_state возвращает только доступную для чтения информацию. У вас нет никакой возможности поменять это состояние в коде кластера веб-серверов, поэтому то, что вы извлекаете состояние базы данных, не подвергает ее никакому риску.

Все выходные переменные БД хранятся в файле состояния, и вы можете считывать их из источника данных terraform_remote_state, используя ссылку на атрибут следующего вида:

data.terraform_remote_state.<NAME>.outputs.<ATTRIBUTE>

Например, вот как можно обновить пользовательские данные веб-серверов кластера, чтобы они извлекали из источника terraform_remote_state адрес и порт базы данных и возвращали их в виде HTTP-ответа:

```
user_data = <<EOF
#!/bin/bash
echo "Hello, World" >> index.html
echo "${data.terraform_remote_state.db.outputs.address}" >> index.html
```

```
echo "${data.terraform_remote_state.db.outputs.port}" >> index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

Чем длиннее становится скрипт в параметре user_data, тем более неряшливым выглядит ваш код. Встраивание одного языка программирования (bash) в другой (Terraform) усложняет поддержку обоих, поэтому давайте на секунду остановимся и вынесем bash-скрипт в отдельный файл. Для этого можно использовать встроенную функцию file и источник данных template_file. Рассмотрим их по отдельности.

Terraform включает в себя ряд *встроенных функций*, которые можно выполнять с помощью выражения такого вида:

```
function_name (...)
```

Возьмем для примера функцию format:

```
format(<FMT>, <ARGS>, ...)
```

Эта функция форматирует аргументы в ARGS в соответствии с синтаксисом sprintf, заданным в строке FMT41. Отличным способом поэкспериментировать со встроенными функциями является использование команды terraformconsole. Она создает интерактивную консоль, в которой вы можете попробовать синтаксис Terraform, запросить состояние вашей инфраструктуры и сразу же получить результаты:

```
$ terraform console
> format("%.3f", 3.14159265359)
3.142
```

Стоит отметить, что консоль Terraform предназначена только для чтения, поэтому не нужно волноваться о случайном изменении инфраструктуры или состояния.

Существует целый ряд других встроенных функций для работы со строками, числами, списками и ассоциативными массивами42. К их числу относится функция file:

```
file(<PATH>)
```

Эта функция читает файл с путем PATH и возвращает его содержимое в виде строки. Например, вы могли бы сохранить скрипт пользовательских данных в файл stage/services/webserver-cluster/user-data.sh и загружать его следующим образом:

```
file("user-data.sh")
```

Подвох в том, что скрипту пользовательских данных для кластера веб-серверов необходима определенная динамическая информация из Terraform, включая порт сервера, а также адрес и порт базы данных. Когда скрипт был встроен в код Terraform, он получал эти значения с помощью ссылок и

интерполяции. С функцией file это не сработает. Однако вместо этого вы можете воспользоваться источником данных template_file.

Источник данных template_file принимает два аргумента: template (строка, которую нужно обработать) и vars (ассоциативный массив с переменными, которые должные быть доступны во время обработки). Он содержит одну выходную переменную, rendered, которая является результатом обработки template. Чтобы увидеть это в деле, добавьте следующий код источника template_file в файл stage/services/webserver-cluster/main.tf:

```
data "template_file" "user_data" {
  template = file("user-data.sh")
  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

Как видите, этот код присваивает параметрам template и vars содержимое скрипта user-data.sh и, соответственно, три переменные, которые нужны этому скрипту: порт сервера, адрес и порт базы данных. Для использования этих переменных нужно соответствующим образом обновить скрипт stage/services/webserver-cluster/user-data.sh:

```
#!/bin/bash
cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF
nohup busybox httpd -f -p ${server_port} &
```

Обратите внимание на несколько изменений в этом bash-скрипте по сравнению с оригиналом.

• Он ищет переменные с помощью стандартного синтаксиса интерполяции Terraform. Единственными переменными в данном случае являются те, что находятся в ассоциативном массиве vars источника template_file. Стоит отметить, что для доступа к ним не нужен никакой префикс: например, вместо var.server_port следует писать server_port.

• Теперь в скрипте можно заметить синтаксис HTML (скажем, <h1>), который делает вывод более удобным для чтения в браузере.

Замечание о внешних файлах

Одним из преимуществ выноса скрипта пользовательских данных в отдельный файл является возможность написания для него модульных тестов. Код теста может даже заполнить интерполированные значения с помощью переменных среды, поскольку для поиска последних bash использует тот же синтаксис, который в Terraform применяется для интерполяции. Вы можете написать автоматический тест для user-data.sh примерно такого вида:

```
export db_address=12.34.56.78
export db_port=5555
export server_port=8888
./user-data.sh
output=$(curl "http://localhost:$server_port")
if [[ $output == *"Hello, World"* ]]; then
  echo "Success! Got expected text from server."
else
  echo "Error. Did not get back expected text 'Hello, World'."
fi
```

Заключительным шагом будет обновление параметра user_data в ресурсе aws_launch_configuration. Присвойте ему обработанный выходной атрибут источника данных template_file:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]
  user_data       = data.template_file.user_data.rendered
  # Требуется при использовании группы автомасштабирования
  # в конфигурации запуска.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

Выглядит намного аккуратней, чем встраивание bash-скриптов!

Если развернуть этот кластер с помощью команды terraformapply, подождать, пока серверы не зарегистрируются в ALB, и открыть URL-адрес ALB в браузере, можно увидеть нечто похожее на рис. 3.12.

Ура! Ваш кластер веб-серверов теперь может программно обращаться к адресу и порту базы данных через Terraform. Если вы используете настоящий веб-фреймворк (вроде Ruby on Rails), можете задать адрес и порт в виде переменных среды или записать их в конфигурационный файл, чтобы их могла использовать ваша библиотека для работы с БД (как ActiveRecord).

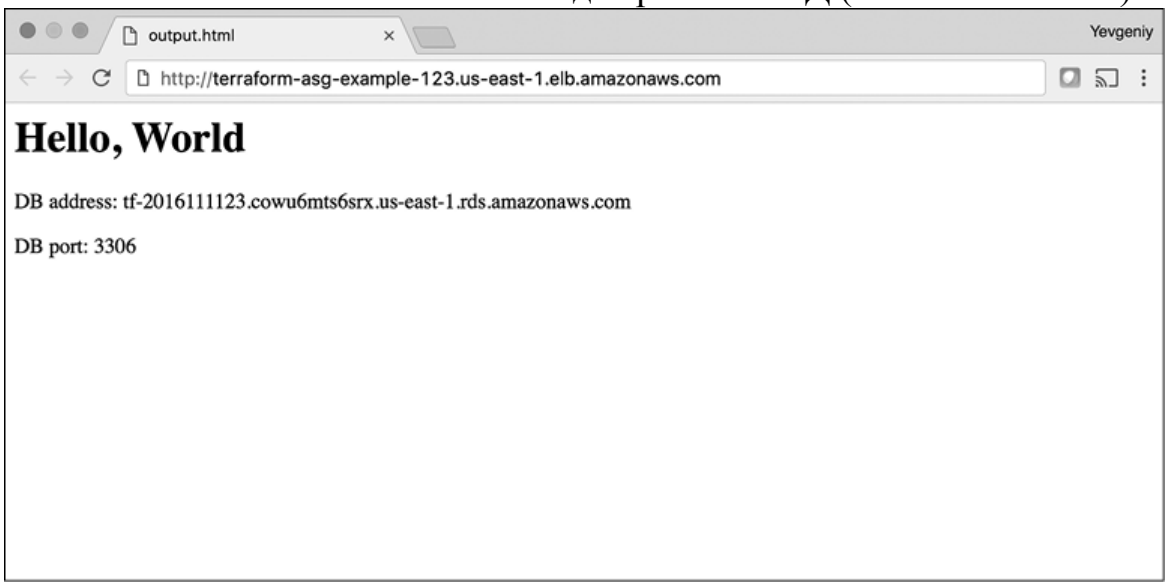


Рис. 3.12. Кластер веб-серверов может программно обращаться к адресу и порту базы данных

Резюме

Причина, по которой вам следует уделять столько внимания изоляции, блокированию и состоянию, заключается в том, что IaC отличается от обычного программирования. При написании кода для типичного приложения большинство ошибок оказываются относительно несущественными и портят только небольшую его часть. Но когда вы пишете код для управления инфраструктурой, программные ошибки имеют более серьезные последствия, поскольку могут затронуть все ваши приложения вместе со всеми источниками данных, топологией сети и практически всем остальным. Поэтому при работе над IaC советую использовать больше «защитных механизмов», чем при написании обычного кода43.

Применение рекомендуемой структуры файлов и каталогов часто приводит к дублированию кода. Если вы хотите запускать кластер веб-серверов как в тестовой, так и в промышленной среде, как избежать

копирования и вставки большого количества фрагментов между stage/services/webserver-cluster и prod/services/webserver-cluster? Ответ: использовать модули Terraform, которым посвящена глава 4.

34 Узнайте больше о гарантиях, которые дает S3, по адресу amzn.to/31ihjAg.

35 Ознакомьтесь с тарифами для S3 по адресу amzn.to/2yTtnw1.

36 Подробнее об именах бакетов S3 можно почитать по адресу bit.ly/2b1s7eh.

37 Ознакомьтесь с тарифами для DynamoDB по адресу amzn.to/2OJiyHp.

38 По адресу bit.ly/2lTsewM представлен яркий пример того, что может случиться, если не изолировать состояние Terraform.

39 Подробнее об этом читайте в документации Terragrunt по адресу bit.ly/2M48S8e.

40 Большинство командных оболочек в Linux/Unix/OS X сохраняют каждую вводимую вами команду в файл истории какого-то рода (например, ~/.bash_history). Но если начать команду с пробела, почти все оболочки не станут ее туда записывать. Имейте в виду, что эта возможность может быть отключена в вашей командной оболочке. Чтобы ее включить, придется присвоить переменной среды HISTCONTROL значение ignoreboth.

41 На странице golang.org/pkg/fmt/ можно найти документацию для синтаксиса sprintf.

42 На странице bit.ly/2GNCxOM представлен полный список встроенных функций.

43 Подробнее о защитных механизмах в ПО можно почитать по адресу bit.ly/2YJuqJb.

4. Повторное использование инфраструктуры с помощью модулей Terraform

В конце главы 3 вы развернули архитектуру, показанную на рис. 4.1.

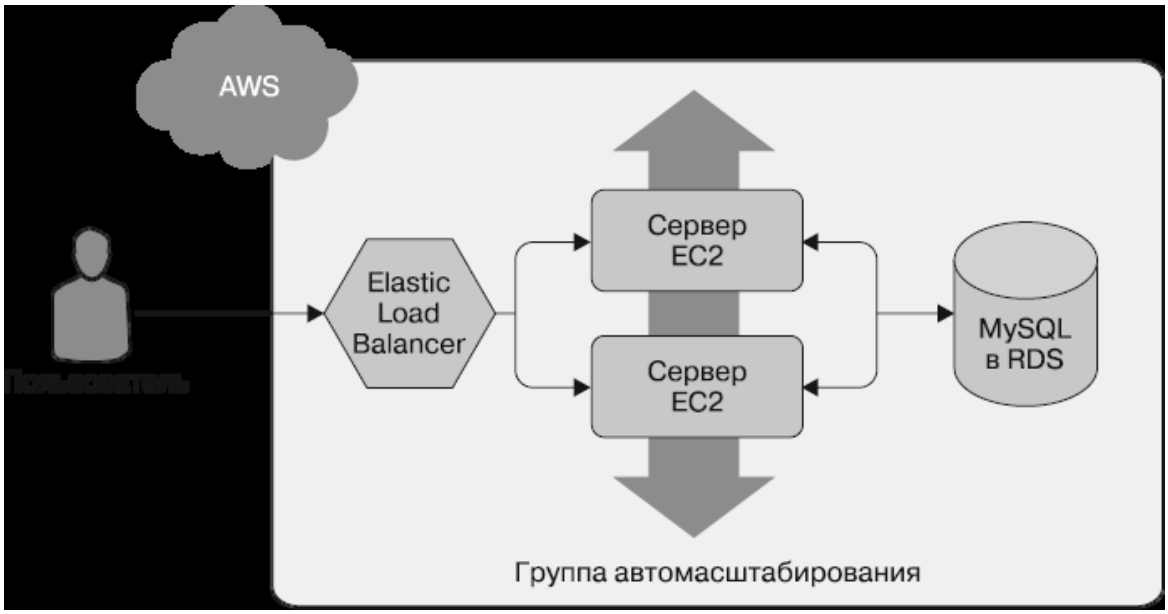


Рис. 4.1. Балансировщик нагрузки, кластер веб-серверов и база данных

Это хорошо работает в качестве первой среды, но их обычно нужно как минимум две: одна для финального тестирования внутри команды, а другая, промышленная, — для обслуживания реальных пользователей (рис. 4.2). В идеале обе среды должны быть почти идентичными, хотя, чтобы сэкономить, для среды финального тестирования важно инициализировать чуть меньше серверов (или серверы меньшего размера).

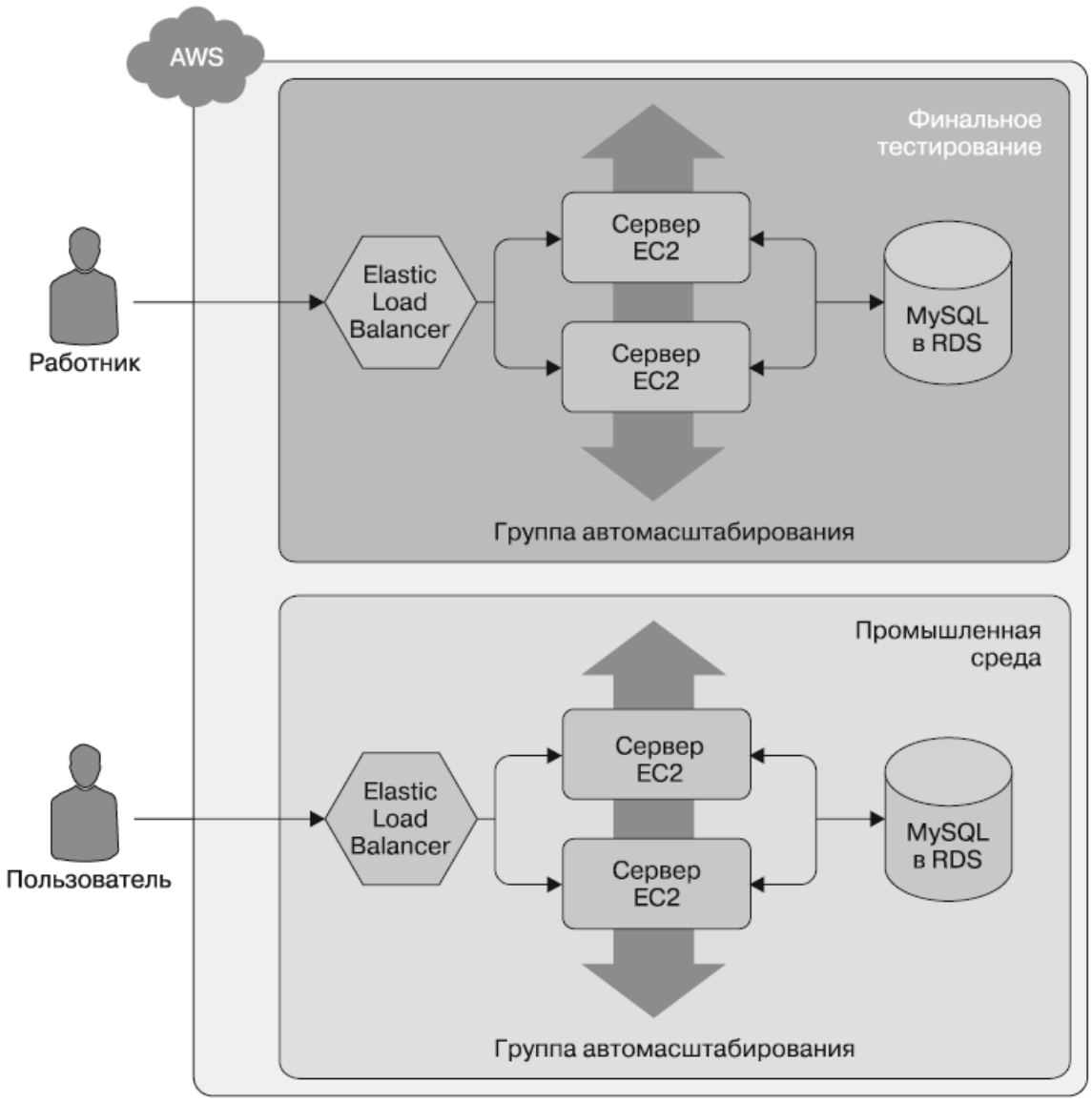


Рис. 4.2. Две среды, каждая со своим балансировщиком нагрузки, кластером веб-серверов и базой данных

Как добавить эту промышленную среду без копирования всего кода из среды тестирования? Например, как избежать дублирования всего содержимого stage/services/webserver-clusteristage/data-stores/mysql в prod/services/webserver-cluster и соответственно prod/data-stores/mysql?

В языках общего назначения, таких как Ruby, код, который копируется и вставляется в нескольких местах, можно оформить в виде функции и использовать в других частях программы:

```
def example_function()
  puts "Hello, World"
end
# Другие участки вашего кода
```

example_function()

Terraform позволяет поместить код внутрь *модуля*, который можно будет повторно применять на разных участках вашей конфигурации. Вместо того чтобы копировать и вставлять код в тестовой и промышленной средах, мы сделаем так, чтобы обе среды использовали код из одного и того же модуля, как показано на рис. 4.3.

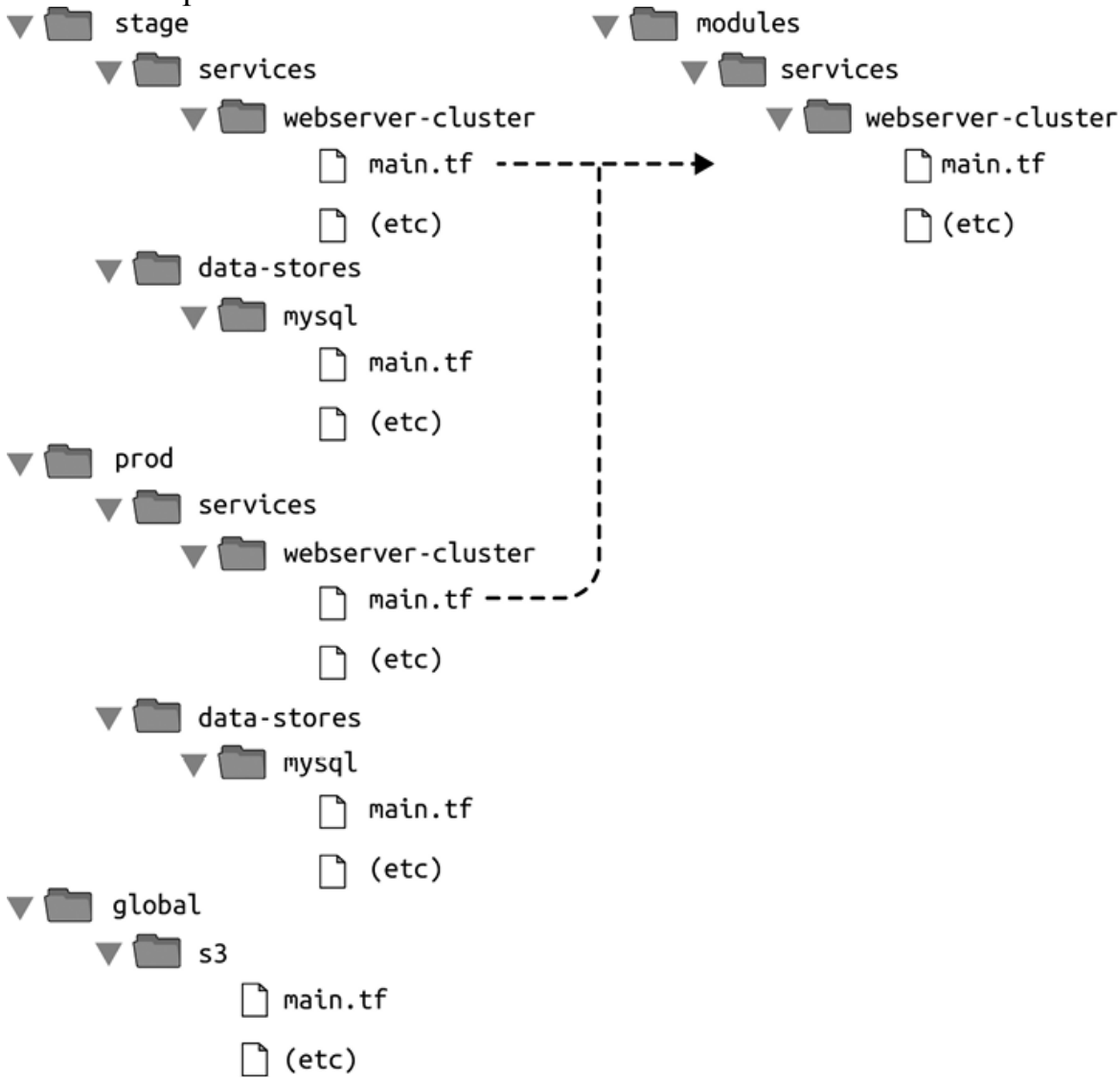


Рис. 4.3. Поместив код в модуль, вы сможете применить его повторно в разных окружениях

Это очень важно. Модули являются ключевым аспектом написания универсального кода Terraform, который легко поддерживать и тестировать. Начав их использовать, вы уже не сможете без них обойтись. Вы начнете оформлять все в виде модулей, объединять их в библиотеки для удобного

использования в компании, загружать сторонние модули из Интернета и воспринимать всю свою инфраструктуру как набор универсальных модулей.

В этой главе я покажу, как создавать и применять модули Terraform. Мы рассмотрим такие темы.

- Основные характеристики модулей.
- Входные параметры модулей.
- Локальные переменные модулей.
- Выходные переменные модулей.
- Подводные камни.
- Управление версиями.

Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

Что такое модуль

В Terraform любой набор конфигурационных файлов, размещенных в одной папке, считается модулем. Вся конфигурация, которую вы уже написали, формально состоит из модулей, хоть и не очень интересных, так как вы развертывали их напрямую (модуль в текущей рабочей папке называется *корневым*). Чтобы увидеть всю их мощь, нужно использовать один модуль из другого.

В качестве примера превратим в универсальный модуль код из stage/services/webserver-cluster, который включает в себя Auto Scaling Group (ASG), Application Load Balancer (ALB), группы безопасности и многие другие ресурсы.

Для начала выполните terraformdestroy в stage/services/webserver-cluster, чтобы удалить все ресурсы, созданные вами ранее. Затем создайте новую папку верхнего уровня под названием modules и переместите все файлы из stage/services/webserver-cluster в modules/services/webserver-cluster. В итоге ваша структура папок должна выглядеть так, как на рис. 4.4.

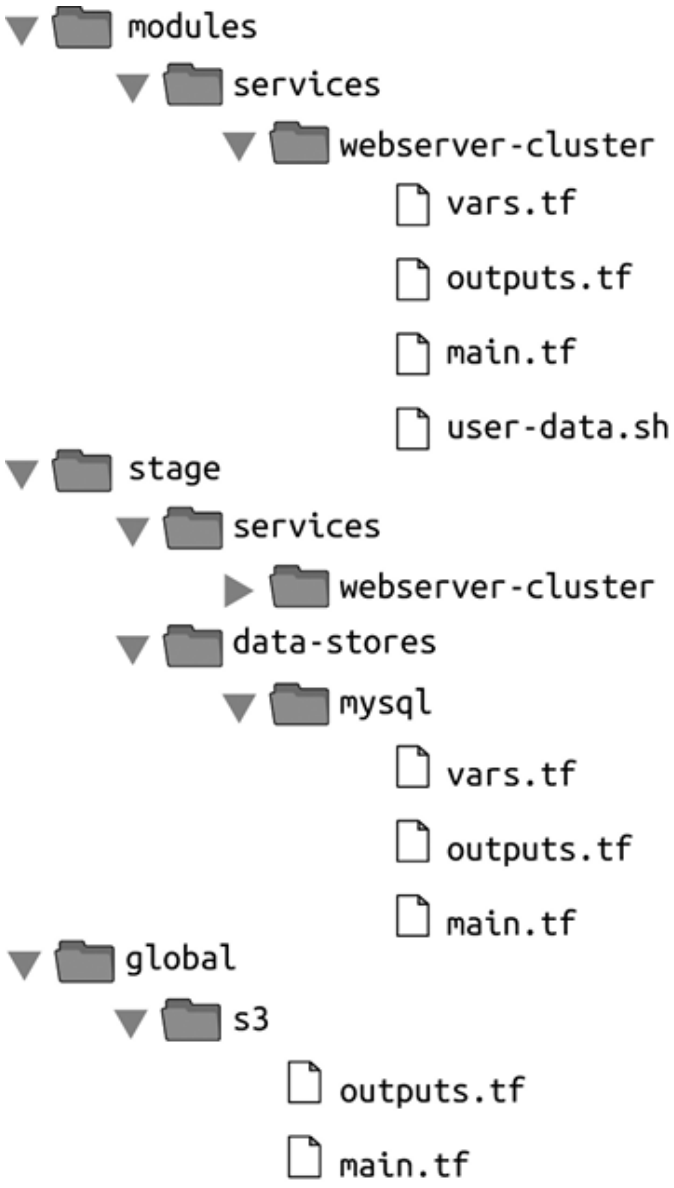


Рис. 4.4. Структура папок с модулем и тестовой средой

Откройте файл main.tf в modules/services/webserver-cluster и уберите из него определение provider. Провайдеры должны настраиваться не самим модулем, а его пользователями.

Теперь можно воспользоваться этим модулем в тестовой среде, используя следующий синтаксис:

```
module "<NAME>" {
  source = "<SOURCE>"
  [CONFIG ...]
}
```

NAME — это идентификатор, который можно использовать в коде Terraform для обращения к модулю (вроде web-service), SOURCE — путь к

коду модуля (скажем, modules/services/webserver-cluster), а CONFIG состоит из одного/нескольких аргументов, предназначенных специально для этого модуля. Например, вы можете создать новый файл stage/services/webserver-cluster/main.tf и использовать в нем модуль webserver-cluster следующим образом:

```
provider "aws" {
  region = "us-east-2"
}
module "webserver_cluster" {
  source = "../../../modules/services/
    webserver-cluster"
}
```

Затем вы можете повторно использовать тот же модуль в промышленной среде, создав новый файл prod/services/webserver-cluster/main.tf следующего содержания:

```
provider "aws" {
  region = "us-east-2"
}
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
}
```

Вот и все: повторное использование кода в разных окружениях с минимальным дублированием. Обратите внимание, что при добавлении модуля в конфигурацию Terraform или изменении параметра модуля source необходимо сначала выполнить команду init, а только потом plan или apply:

```
$ terraform init
Initializing modules...
- webserver_cluster in ../../../modules/services/webserver-cluster
Initializing the backend...
Initializing provider plugins...
Terraform has been successfully initialized!
```

Теперь вы знаете обо всех хитростях в арсенале команды init. Она сама умеет загружать провайдеры и модули, а также конфигурировать ваши хранилища.

Прежде чем применять этот код, нужно упомянуть об одном недостатке модуля webserver-cluster: все имена в нем прописаны вручную. Это касается групп безопасности, ALB и других ресурсов. Таким образом, при попытке повторного использования этого модуля вы получите конфликты имен. Прямо в коде прописаны даже параметры для обращения к базе данных, поскольку файл main.tf, который вы скопировали в

modules/services/webserver-cluster, берет адрес и порт БД из источника данных terraform_remote_state, а тот написан лишь с расчетом на тестовую среду.

Чтобы исправить эти проблемы, необходимо добавить в модуль webserver-cluster конфигурируемые входные параметры. Это позволит ему менять свое поведение в зависимости от окружения.

Входные параметры модуля

В языке программирования общего назначения, таком как Ruby, функцию можно сделать конфигурируемой, передав ей входные параметры:

```
def example_function(param1, param2)
  puts "Hello, #{param1} #{param2}"
end
# Другие участки вашего кода
example_function("foo", "bar")
```

Модули Terraform тоже могут иметь параметры. Для их определения используется уже знакомый вам механизм: входные переменные. Откройте файл modules/services/webserver-cluster/variables.tf и добавьте три новых блока variable:

```
variable "cluster_name" {
  description = "The name to use for all the cluster resources"
  type        = string
}
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the database's remote state"
  type        = string
}
variable "db_remote_state_key" {
  description = "The path for the database's remote state in S3"
  type        = string
}
```

Далее пройдитесь по файлу modules/services/webserver-cluster/main.tf и подставьте var.cluster_name прописанных вручную имен (скажем, "terraform-asgexample"). Например, вот как это сделать в группе безопасности ALB:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
```

```
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Обратите внимание на то, что параметру name присваивается "${var.cluster_name}-alb". Аналогичные изменения нужно внести и в другой ресурс aws_security_group (можете назвать его "${var.cluster_name}-instance"), а также в aws_alb и раздел tag ресурса aws_autoscaling_group.

Вы также должны обновить источник данных terraform_remote_state, чтобы он использовал db_remote_state_bucket и db_remote_state_key в качестве параметров bucket и соответственно key. Это позволит ему считывать файл состояния из правильной среды:

```
data "terraform_remote_state" "db" {
  backend = "s3"
  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}
```

Теперь можете аналогичным образом задать эти новые входные переменные в тестовой среде в файле stage/services/webserver-cluster/main.tf:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
}
```

То же самое нужно сделать для промышленной среды в файле prod/services/webserver-cluster/main.tf:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"
```

}

Промышленная база данных еще не существует. В качестве упражнения попробуйте добавить ее самостоятельно по аналогии с тестовой.

Как видите, для установки входных переменных модуля и аргументов ресурса используется один и тот же синтаксис. Входящие переменные являются API модуля и определяют то, как он себя ведет в разных окружениях. В этом примере мы задаем разные имена для разных сред, но вы можете сделать конфигурируемыми и другие параметры. Предположим, чтобы сэкономить деньги, в тестовой среде можно запускать небольшой кластер веб-серверов, но в промышленных условиях вам понадобится большой кластер, способный справиться с сильными нагрузками. Для этого в файл modules/services/webserver-cluster/variables.tf можно добавить еще три входные переменные:

```
variable "instance_type" {
  description = "The type of EC2 Instances to run (e.g. t2.micro)"
  type        = string
}
variable "min_size" {
  description = "The minimum number of EC2 Instances in the ASG"
  type        = number
}
variable "max_size" {
  description = "The maximum number of EC2 Instances in the ASG"
  type        = number
}
```

Дальше нужно обновить конфигурацию запуска в файле modules/services/webserver-cluster/main.tf, присвоив параметру instance_type новую входную переменную var.instance_type:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data       = data.template_file.user_data.rendered
  # Требуется при использовании группы автомасштабирования
  # в конфигурации запуска.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
```

```
  lifecycle {
    create_before_destroy = true
  }
}
```

Похожим образом следует обновить определение ASG в том же файле. Присвойте параметрам min_size и max_size входные переменные var.min_size и var.max_size соответственно:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"
  min_size = var.min_size
  max_size = var.max_size
  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
}
```

Теперь кластер в тестовой среде (stage/services/webserver-cluster/main.tf) можно сделать поменьше и подешевле, указав "t2.micro" для instance_type и 2 для min_size и max_size:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

В то же время в промышленной среде можно использовать более крупный тип серверов (например, m4.large) с большим количеством серверов и памяти. Имейте в виду, что этот тип *не* входит в бесплатный тариф AWS, поэтому, если кластер вам нужен только в образовательных целях и вы не хотите платить, оставьте в поле instance_type значение "t2.micro". Параметру max_size можно присвоить 10, что позволит кластеру расширяться и сжиматься в зависимости от нагрузки (не волнуйтесь, изначально он запустится с двумя серверами):

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"
  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

Локальные переменные модулей

Определение параметров модуля с помощью входных переменных — отличный подход, но что, если вам нужно определить переменную внутри модуля для каких-то промежуточных вычислений или просто для того, чтобы не дублировать код, но при этом вы не хотите делать ее доступной в качестве конфигурируемого ввода? Например, балансировщик нагрузки в модуле webserver-cluster (modules/services/webserver-cluster/main.tf) прослушивает стандартный для HTTP порт под номером 80. Сейчас нам приходится копировать и вставлять этот номер в разных местах, в том числе и в прослушивателе балансировщика:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"
  # По умолчанию возвращает простую страницу с кодом 404
  default_action {
    type = "fixed-response"
    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

А вот группа безопасности балансировщика:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
  ingress {
    from_port   = 80
    to_port     = 80
```

```
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Значения в этой группе безопасности, включая блок CIDR 0.0.0.0/0 (любые IP-адреса), номер порта 0 (любой порт) и произвольный протокол "-1", тоже копируются и вставляются на нескольких участках модуля. Явное и многократное задание этих «магических» значений усложняет чтение и поддержку кода. Их можно вынести во входные переменные, но в таком случае ваш модуль может (случайно) переопределить эти значения, что может быть нежелательным. Вместо этого можно определить *локальные значения* в блоке locals:

```
locals {
  http_port    = 80
  any_port     = 0
  any_protocol = "-1"
  tcp_protocol = "tcp"
  all_ips      = ["0.0.0.0/0"]
}
```

Локальные значения позволяют назначить любому выражению Terraform имя, которое затем можно использовать в коде модуля. Такие имена видны только в самом модуле, поэтому они не имеют никакого влияния на внешний код, при этом вы не можете перезаписать их извне. Чтобы прочитать локальное значение, нужна *локальная ссылка* со следующим синтаксисом:

```
local.<NAME>
```

Используйте этот синтаксис для обновления прослушивателя балансировщика нагрузки:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"
  # По умолчанию возвращает простую страницу с кодом 404
  default_action {
```

```
    type = "fixed-response"
    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

и всех групп безопасности в модуле, включая ту, которая относится к балансировщику:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
  ingress {
    from_port   = local.http_port
    to_port     = local.http_port
    protocol    = local.tcp_protocol
    cidr_blocks = local.all_ips
  }
  egress {
    from_port   = local.any_port
    to_port     = local.any_port
    protocol    = local.any_protocol
    cidr_blocks = local.all_ips
  }
}
```

Локальные переменные упрощают чтение и поддержку кода, поэтому используйте их как можно чаще.

Выходные переменные модуля

Мощной особенностью групп ASG является возможность сконфигурировать их для увеличения и уменьшения количества запущенных серверов в зависимости от нагрузки. Для этого можно воспользоваться *запланированным действием*, которое будет менять размер кластера в заданное время суток. Например, если ваш кластер испытывает повышенную нагрузку в рабочее время, вы можете запланировать увеличение и уменьшение количества серверов на 9 утра и 5 вечера соответственно.

Запланированное действие, определенное в модуле webserver-cluster, относится как к тестовой, так и к промышленной среде. Поскольку вам не нужно подобного рода масштабирование во время тестирования, можете

пока определить график автомасштабирования прямо в промышленной конфигурации. В главе 5 вы познакомитесь с условным определением ресурсов, что позволит вам переместить запланированное действие в модуль webserver-cluster.

Чтобы определить запланированное действие, добавьте следующих два ресурса aws_autoscaling_schedule в файл prod/services/webserver-cluster/main.tf:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence            = "0 9 * * *"
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
  scheduled_action_name = "scale-in-at-night"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 2
  recurrence            = "0 17 * * *"
}
```

Первый ресурс aws_autoscaling_schedule используется для увеличения количества серверов до десяти в утреннее время (в параметре recurrence используется синтаксис cron, поэтому "09***" означает «в 9 утра каждый день»), а второй уменьшает этот показатель на ночь ("017***" значит «в 5 вечера каждый день»). Однако в обоих случаях не хватает параметра autoscaling_group_name, который задает имя ASG. Сама группа ASG определяется внутри модуля webserver-cluster. Как же получить доступ к ее имени? В языках общего назначения, таких как Ruby, функции могут возвращать значения:

```
def example_function(param1, param2)
  return "Hello, #{param1} #{param2}"
end
# Другие участки вашего кода
return_value = example_function("foo", "bar")
```

В Terraform модули тоже могут возвращать значения. Для этого используется уже знакомый вам механизм: выходные переменные. Вы можете добавить имя ASG в качестве выходной переменной в файле modules/services/webserver-cluster/outputs.tf, как показано ниже:

```
output "asg_name" {
```

```
  value       = aws_autoscaling_group.example.name
  description = "The name of the Auto Scaling Group"
}
```

Для обращения к выходным переменным модуля используется следующий синтаксис:

module.<MODULE_NAME>.<OUTPUT_NAME>

Например:

module.frontend.asg_name

Этот синтаксис можно использовать в файле prod/services/webserver-cluster/main.tf, чтобы установить параметр autoscaling_group_name в каждом из ресурсов aws_autoscaling_schedule:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence            = "0 9 * * *"
  autoscaling_group_name = module.webserver_cluster.asg_name
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
  scheduled_action_name = "scale-in-at-night"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 2
  recurrence            = "0 17 * * *"
  autoscaling_group_name = module.webserver_cluster.asg_name
}
```

Возможно, вам следует сделать доступным еще одно выходное значение в модуле webserver-cluster: доменное имя ALB. Так вы будете знать, какой URL-адрес нужно проверить после развертывания кластера. Для этого в файле /modules/services/webserver-cluster/outputs.tf необходимо еще раз добавить выходную переменную:

```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "The domain name of the load balancer"
}
```

После этого данный вывод можно «пропустить через» файлы stage/services/webserver-cluster/outputs.tf и prod/services/webserver-cluster/outputs.tf:

```
output "alb_dns_name" {
```

```
  value       = module.webserver_cluster.alb_dns_name
  description = "The domain name of the load balancer"
}
```

Ваш кластер веб-серверов почти готов к развертыванию. Осталось только принять во внимание несколько подводных камней.

Подводные камни

При создании модулей обращайте внимание:

• на файловые пути;

• вложенные блоки.

Файловые пути

В главе 3 вы поместили скрипт пользовательских данных для кластера веб-серверов во внешний файл, user-data.sh, и применили встроенную функцию file, чтобы прочитать его с диска. Неочевидный момент функции file состоит в том, что файловый путь, который она использует, должен быть относительным (поскольку Terraform можно запускать на множестве разных компьютеров) — но относительно чего?

По умолчанию Terraform интерпретирует этот путь относительно текущей рабочей папки. Это не вызывает проблем, если вы используете функцию file в конфигурационном файле Terraform в той же папке, из которой выполняется команда terraformapply (то есть если функция file применяется в корневом модуле). Но если сделать то же самое в модуле, размещенном в отдельной папке, ничего не будет работать.

Решить эту проблему можно с помощью выражения, известного как *«ссылка на путь»*, которое имеет вид path.<TYPE>. Terraform поддерживает следующие типы этих ссылок.

• path.module — возвращает путь к модулю, в котором определено выражение.

• path.root — возвращает путь к корневому модулю.

• path.cwd — возвращает путь к текущей рабочей папке. При нормальном использовании Terraform это значение совпадает с path.root, но в некоторых нестандартных случаях Terraform запускается не из папки корневого модуля, что приводит к расхождению этих путей.

Для скрипта пользовательских данных нужен путь, взятый относительно самого модуля, поэтому в источнике данных template_file в файле modules/services/webser-vercluster/main.tf следует применять path.module:

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")
```

```
  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

Вложенные блоки

Конфигурацию некоторых ресурсов Terraform можно определять отдельно или в виде вложенных блоков. При создании модулей всегда следует отдавать предпочтение отдельным ресурсам.

Например, ресурс aws_security_group позволяет определять входящие и исходящие правила в виде вложенных блоков. Вы уже это видели в модуле webserver-cluster (modules/services/webserver-cluster/main.tf):

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
  ingress {
    from_port   = local.http_port
    to_port     = local.http_port
    protocol    = local.tcp_protocol
    cidr_blocks = local.all_ips
  }
  egress {
    from_port   = local.any_port
    to_port     = local.any_port
    protocol    = local.any_protocol
    cidr_blocks = local.all_ips
  }
}
```

Вы должны модифицировать этот модуль так, чтобы те же входящие и исходящие правила определялись в виде отдельных ресурсов aws_security_group_rule (не забудьте сделать это для обеих групп безопасности в данном модуле):

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
}
resource "aws_security_group_rule" "allow_http_inbound" {
  type              = "ingress"
  security_group_id = aws_security_group.alb.id
  from_port   = local.http_port
```

```
  to_port     = local.http_port
  protocol    = local.tcp_protocol
  cidr_blocks = local.all_ips
}
resource "aws_security_group_rule" "allow_all_outbound" {
  type              = "egress"
  security_group_id = aws_security_group.alb.id
  from_port   = local.any_port
  to_port     = local.any_port
  protocol    = local.any_protocol
  cidr_blocks = local.all_ips
}
```

При попытке *одновременного* использования вложенных блоков и отдельных ресурсов вы получите ошибки, когда правила маршрутизации конфликтуют и переопределяют друг друга. Поэтому вы должны выбрать что-то одно. В связи с этим ограничением при создании модуля всегда следует использовать отдельные ресурсы вместо вложенных блоков. В противном случае ваш модуль получится менее гибким и конфигурируемым.

Например, если все входящие и исходящие правила в модуле webserver-cluster определены в виде отдельных ресурсов aws_security_group_rule, вы сможете сделать этот модуль достаточно гибким для того, чтобы разрешить пользователям добавлять собственные правила за его пределами. Для этого идентификатор aws_security_group нужно экспортировать в виде выходной переменной внутри modules/services/webserver-cluster/outputs.tf:

```
output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "The ID of the Security Group attached to the load balancer"
}
```

Теперь представьте, что вам нужно сделать доступным извне дополнительный порт, сугубо для тестирования. Теперь это легко сделать: нужно лишь добавить в файл stage/services/webserver-cluster/main.tf ресурс aws_security_group_rule:

```
resource "aws_security_group_rule" "allow_testing_inbound" {
  type              = "ingress"
  security_group_id = module.webserver_cluster.alb_security_group_id
  from_port   = 12345
  to_port     = 12345
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
```

}

Если бы вы определили входящее или исходящее правило в виде вложенного блока, этот код был бы нерабочим. Стоит отметить, что эта же проблема характерна для целого ряда ресурсов Terraform, включая следующие:

•aws_security_group и aws_security_group_rule;

• aws_route_table и aws_route;

•aws_network_acl и aws_network_acl_rule.

Теперь вы готовы развернуть свой кластер веб-серверов сразу в тестовой и промышленной средах. Выполните terraformapply и наслаждайтесь работой с двумя отдельными копиями своей инфраструктуры.

Сетевая изоляция

Показанные в этой главе примеры создают две среды, изолированные как в вашем коде Terraform, так и с точки зрения инфраструктуры: у них есть отдельные балансировщики нагрузки, серверы и базы данных. Но, несмотря на это, они не изолированы на уровне сети. Чтобы не усложнять примеры кода, все ресурсы в этой книге развертываются в одно и то же виртуальное частное облако (VPC). Это означает, что серверы в тестовой и промышленной среде могут взаимодействовать между собой.

В реальных сценариях использования запуск двух окружений в одном облаке VPC чреват сразу двумя рисками. Во-первых, ошибка, допущенная в одной среде, может повлиять на другую. Например, если при внесении изменений в тестовом окружении вы случайно сломаете правила в таблице маршрутизации, это скажется на перенаправлении трафика и в промышленных условиях. Во-вторых, если злоумышленник получит доступ к одной среде, он сможет проникнуть и в другую. Если вы активно меняете код в тестовом окружении и случайно оставите открытым какой-нибудь порт, любой взломщик, проникший внутрь, сможет завладеть не только тестовыми, но и промышленными данными.

В связи с этим, если не считать простые примеры и эксперименты, вы должны размещать каждую среду в отдельном облаке VPC. Для пущей уверенности окружения можно даже разнести по разным учетным записям AWS.

Управление версиями

Если ваши тестовая и промышленная среды ссылаются на папку с одним и тем же модулем, любое изменение в этой папке коснется и той и другой при следующем же развертывании. Такого рода связывание усложняет тестирование изменений в изоляции от промышленного окружения. Вместо этого лучше использовать разные *версии модулей*: например, v0.0.2 для тестовой среды и v0.0.1 для промышленной, как показано на рис. 4.5.

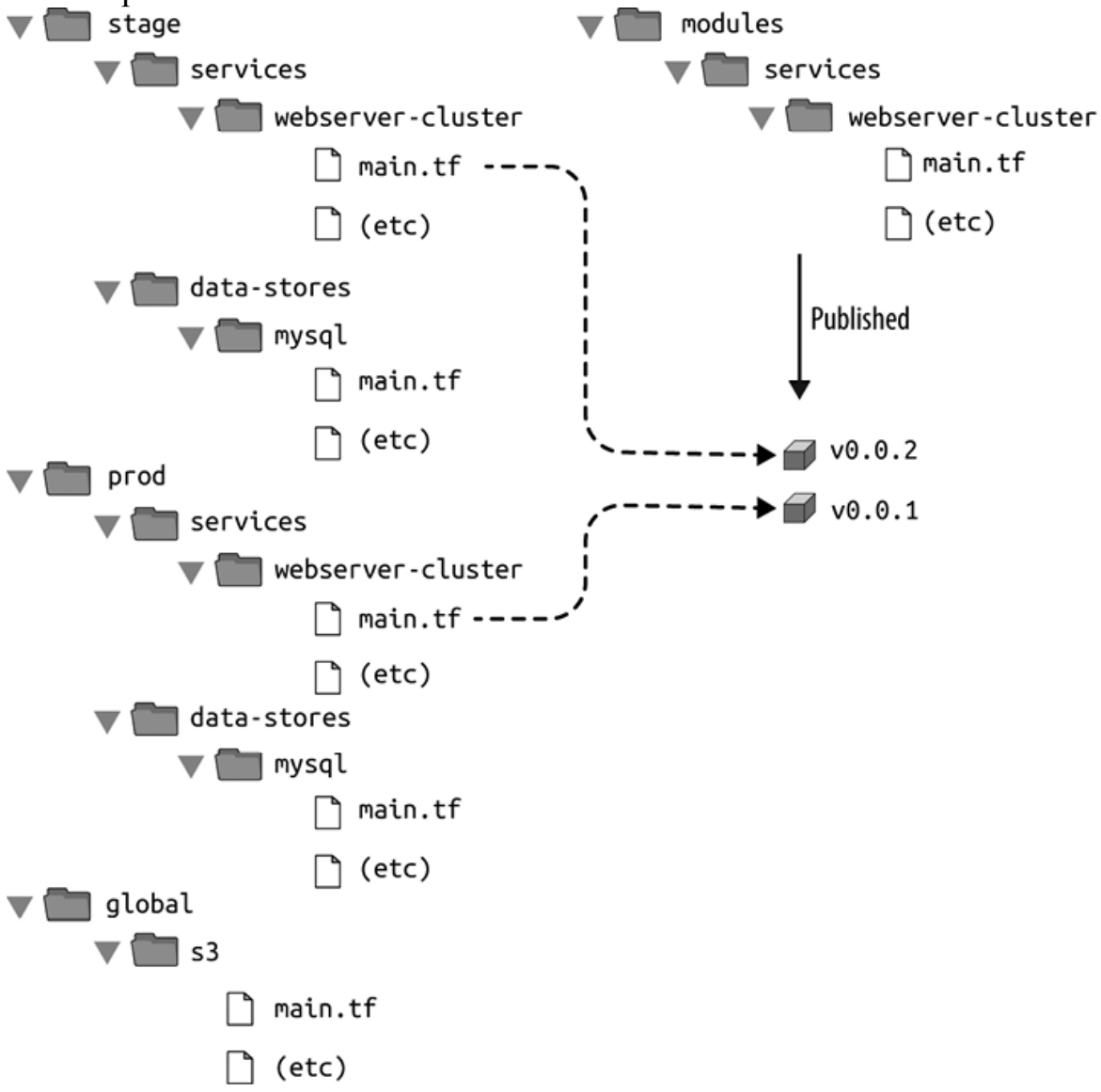


Рис. 4.5. Применение разных версий модуля в разных окружениях

Во всех примерах с модулями, которые вы видели до сих пор, параметр source содержал локальный файловый путь. Но, помимо файловых путей, модули Terraform поддерживают и другие виды источников, такие как URL-адреса Git/Mercurial и произвольные URL44. Самый простой способ

управления версиями модуля — размещение его кода в отдельном Git-репозитории, URL-адрес которого затем прописывается в параметре source. Это означает, что код Terraform будет распределен (как минимум) по двум репозиториям.

•modules — в этом репозитории находятся универсальные модули. Каждый модуль — своего рода «чертеж», который описывает определенную часть вашей инфраструктуры.

•live — репозиторий, который содержит текущую инфраструктуру, развернутую в каждом окружении (stage, prod, mgmt и т. д.). Это такие «здания», которые вы строите по «чертежам», взятым из репозитория modules.

Обновленная структура папок для вашего кода Terraform будет выглядеть примерно так, как на рис. 4.6.

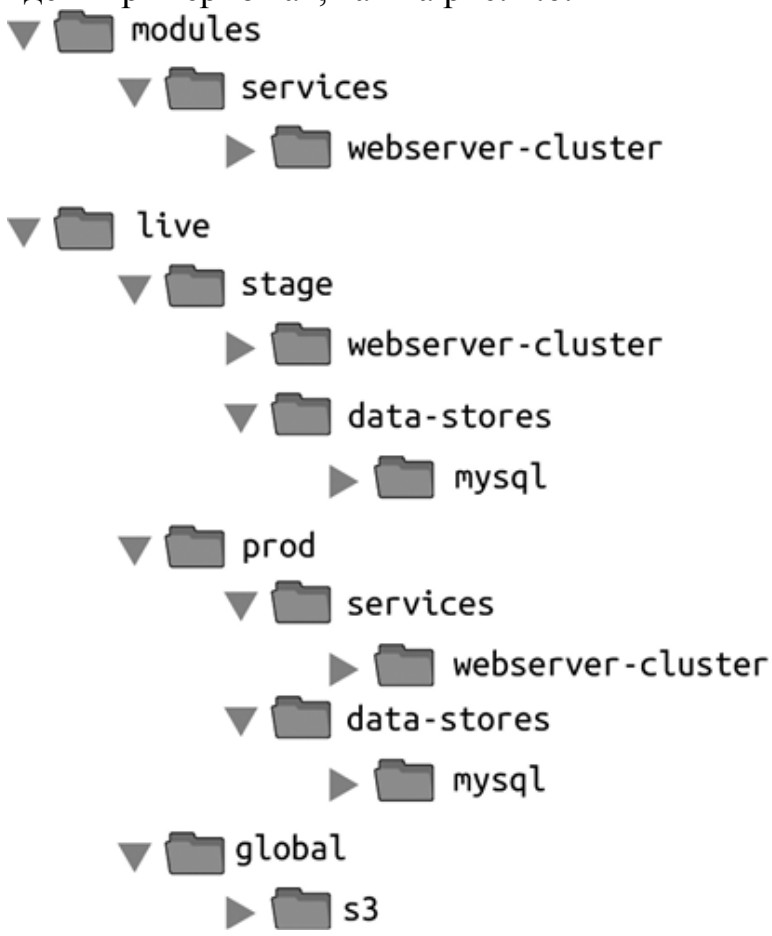


Рис. 4.6. Структура файлов и каталогов с несколькими репозиториями

Чтобы организовать код таким образом, сначала нужно переместить папки stage, prod и global в папку под названием live. Затем вы должны разнести папки live и modules по отдельным Git-репозиториям. Вот пример того, как это делается с папкой modules:

```
$ cd modules
$ git init
$ git add .
$ git commit -m "Initial commit of modules repo"
$ git remote add origin "(URL OF REMOTE GIT REPOSITORY)"
$ git push origin master
```

Репозиторию modules можно также назначить тег, который будет использоваться в качестве номера версии. Если вы работаете с сервисом GitHub, это можно сделать в его пользовательском интерфейсе: создайте выпуск (bit.ly/2Yv8kPg), который автоматически создаст тег. Если вы не используете GitHub, можете применить утилиту командной строки Git:

```
$ git tag -a "v0.0.1" -m "First release of webserver-cluster module"
$ git push --follow-tags
```

Теперь вы можете работать с разными версиями модуля в тестовой и промышленной средах, указав URL-адрес Git в параметре source. Вот как это будет выглядеть в файле live/stage/services/webserver-cluster/main.tf, если ваш репозиторий modules находится в GitHub по адресу github.com/foo/modules (имейте в виду, что двойная косая черта в URL-адресе Git является обязательной):

```
module "webserver_cluster" {
  source = "github.com/foo/modules//webserver-cluster?ref=v0.0.1"
  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

Если вы хотите использовать разные версии модулей без возни с Git-репозиториями, можете загрузить модуль из архива с кодом для этой книги (https://github.com/brikis98/terraform-up-and-running-code). Мне пришлось разбить этот адрес на части, чтобы он поместился на странице. Его следует записывать одной строкой:

```
source = "github.com/brikis98/terraform-up-and-running-code//
  code/terraform/04-terraform-module/module-example/modules/
  services/webserver-cluster?ref=v0.1.0"
```

Параметр ref позволяет указать определенную фиксацию Git по ее хешу SHA1, имя ветки или, как в данном примере, конкретный тег Git. В целом я советую использовать в качестве версий модулей теги. Имена веток нестабильны, так как вы всегда получаете последнюю фиксацию в заданной

ветке, которая может меняться при каждом выполнении команды init, а хеши SHA1 выглядят малопонятными. Теги Git такие же стабильные, как и фиксации (на самом деле это просто указатели на фиксации), но при этом они позволяют применять удобные и разборчивые названия.

Особенно полезной схемой именования тегов является *семантическое версионирование* (http://semver.org). Это система управления версиями в формате MAJOR.MINOR.PATCH (например, 1.0.4) с отдельными правилами относительно того, как следует инкрементировать каждый элемент номера версии. Вы должны инкрементировать:

• версию MAJOR при внесении несовместимых изменений в API;

• версию MINOR при добавлении возможностей с соблюдением обратной совместимости;

• версию PATCH при исправлении ошибок с соблюдением обратной совместимости.

Семантическое версионирование позволяет донести до пользователей модуля, какого рода изменения вы внесли и как это сказывается на процессе обновления.

Поскольку вы обновили свою конфигурацию с использованием разных URL-адресов для разных версий вашего модуля, нужно заново выполнить команду terraforminit, чтобы загрузить его код:

```
$ terraform init
Initializing modules...
Downloading git@github.com:brikis98/terraform-up-and-running-code.git?ref=v0.1.0
for webserver_cluster...
(...)
```

На этот раз вы можете видеть, что Terraform загружает код модуля из Git, а не из вашей локальной файловой системы. Когда все будет готово, вы сможете выполнить команду apply как обычно.

Закрытые Git-репозитории

Если ваш модуль находится в закрытом Git-репозитории, чтобы применять этот репозиторий в качестве источника модуля, нужно позволить Terraform в нем аутентифицироваться. Рекомендую использовать аутентификацию SSH, чтобы не пришлось хранить учетные данные для доступа к репозиторию в самом коде. Каждый разработчик сможет создать

SSH-ключ и привязать его к пользователю Git. После добавления ключа в ssh-agent Terraform будет автоматически использовать его для аутентификации, если в качестве URL источника указан SSH45.

URL-адрес источника должен выглядеть так:

git@github.com:<OWNER>/<REPO>.git//<PATH>?ref=<VERSION>

Например:

git@github.com:acme/modules.git//example?ref=v0.1.2

Чтобы проверить, корректно ли вы отформатировали URL, попробуйте клонировать базовый адрес в терминале с помощью git clone:

$ git clone git@github.com:<OWNER>/<REPO>.git

Если эта команда выполнится успешно, Terraform тоже сможет использовать ваш приватный репозиторий.

Теперь пройдемся по процессу внесения изменений в проекте с разными версиями модулей. Допустим, вы модифицировали модуль webserver-cluster и хотите проверить его в тестовой среде. Для начала изменения нужно зафиксировать в репозитории modules:

$ cd modules
$ git add .
$ git commit -m "Made some changes to webserver-cluster"
$ git push origin master

Затем в том же репозитории нужно создать новый тег:

$ git tag -a "v0.0.2" -m "Second release of webserver-cluster"
$ git push --follow-tags

Теперь вы можете перевести на новую версию *только* тот URL-адрес, который используется в тестовой среде (live/stage/services/webserver-cluster/main.tf):

```
module "webserver_cluster" {
  source = "git@github.com:foo/modules.git//webserver-cluster?ref=v0.0.2"
  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

В промышленной среде (live/prod/services/webserver-cluster/main.tf) можно по-прежнему использовать версию v0.0.1 без всяких изменений:

```
module "webserver_cluster" {
  source = "git@github.com:foo/modules.git//webserver-cluster?ref=v0.0.1"
  cluster_name           = "webservers-prod"
```

```
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"
  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

После того как вы тщательно проверили версию v0.0.2 и убедились в ее корректности в тестовой среде, можете обновить и промышленное окружение. Но если в v0.0.2 обнаружится ошибка, это не составит большой проблемы, поскольку пользователи реальной системы не будут затронуты. Исправьте ошибку, выпустите новую версию и повторите весь процесс заново, пока ваш модуль не станет достаточно стабильным для промышленного применения.

Разработка модулей

Управление версиями модулей отлично подходит, когда развертывание происходит в общем окружении (тестовом или промышленном), но, если вы просто занимаетесь тестированием на собственном компьютере, лучше использовать локальные файловые пути. Это ускорит разработку, поскольку после внесения изменений в код модулей вы сможете сразу же выполнить команду apply в активных папках, без фиксации своего кода и публикации новой версии.

Цель этой книги — сделать процесс изучения и экспериментирования с Terraform максимально быстрым, поэтому в остальных примерах модули будут использовать локальные файловые пути.

Резюме

Описывая IaC в виде модулей, вы получаете возможность использовать в своей инфраструктуре разнообразные рекомендуемые методики программирования: разбирать и тестировать каждое изменение, вносимое в модуль; создавать для каждого модуля выпуски с семантическими версиями; безопасно экспериментировать с разными версиями модулей в разных окружениях и в случае какой-то проблемы откатиться к предыдущему выпуску.

Все это может существенно помочь в построении инфраструктуры быстрым и надежным образом, поскольку разработчики смогут повторно

использовать ее компоненты, которые были как следует проверены и задокументированы. Например, вы можете создать канонический модуль, который описывает процесс развертывания одного микросервиса, включая то, как запускать кластер, масштабировать его в зависимости от нагрузки и распределять запросы между серверами. Затем каждая команда сможет применять этот модуль для управления собственными микросервисами, и для этого будет достаточно лишь нескольких строчек кода.

Чтобы такой модуль подошел сразу нескольким командам, его код должен быть гибким и конфигурируемым. Например, одна команда может использовать его для развертывания одного экземпляра своего микросервиса без балансировщика нагрузки, а другой может понадобиться десяток экземпляров с распределением трафика между ними. Как в Terraform записать условные выражения? Можно ли выполнить цикл for? Можно ли с помощью Terraform выкатывать изменения в микросервисы без простоя? Этим углубленным аспектам синтаксиса Terraform посвящена глава 5.

44 Все подробности о URL-адресах источников можно найти на странице bit.ly/2TaXmZF.

45 Хорошее руководство по работе с SSH-ключами можно найти по адресу bit.ly/2ZFLJwe.

5. Работа с Terraform: циклы, условные выражения, развертывание и подводные камни

Terraform — это декларативный язык. Как уже обсуждалось в главе 1, по сравнению с процедурными языками декларативные обычно дают более точное представление о том, что на самом деле развернуто в IaC. Благодаря этому код остается компактным и в нем легче разобраться. Однако с некоторыми видами задач сложнее справиться в декларативном стиле.

Например, как повторить какой-то элемент бизнес-логики, в частности создание нескольких похожих ресурсов, без дублирования кода, учитывая, что у декларативных языков обычно нет цикла for? И если декларативный язык не поддерживает выражения if, как сконфигурировать ресурсы условным образом: скажем, написать модуль Terraform, который умеет создавать определенные ресурсы только для некоторых пользователей? И как в декларативном языке выразить сугубо процедурную концепцию, такую как развертывание с нулевым временем простоя?

К счастью, Terraform предоставляет несколько элементов языка, которые позволят вам выполнять определенные виды циклов, условных выражений и развертываний. Речь идет о метапараметре count, выражениях for_each и for, блоке жизненного цикла под названием create_before_destroy,

тернарном операторе и большом количестве функций. Эта глава охватывает следующие темы.

• Циклы.

• Условные выражения.

• Развертывание с нулевым временем простоя;

• Подводные камни Terraform.

Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

Циклы

Terraform предоставляет несколько разных циклических конструкций с немного разными сценариями использования.

• Параметр count для циклического перебора ресурсов.

• Выражение for_each для циклического перебора ресурсов и их вложенных блоков.

• Выражение for для циклического перебора списков и ассоциативных массивов.

• Строковая директива for для циклического перебора списков и ассоциативных массивов внутри строк.

Рассмотрим их одну за другой.

Циклы с параметром count

В главе 2 с помощью консоли AWS вы создали учетную запись AWS и пользователя Access Management (IAM). Теперь с помощью этого пользователя вы можете создавать и администрировать всех будущих пользователей IAM прямо в коде Terraform. Рассмотрим следующий код, который должен находиться в файле live/global/iam/main.tf:

```
provider "aws" {
  region = "us-east-2"
}
resource "aws_iam_user" "example" {
  name = "neo"
}
```

Здесь используется ресурс aws_iam_user для создания одного нового пользователя IAM. Но если необходимо создать трех пользователей? В языке программирования общего назначения вы бы применили цикл for:

```
# Это просто псевдокод. Он не будет работать в Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo"
  }
}
```

В языке Terraform нет встроенной поддержки циклов for и другой традиционной процедурной логики, поэтому такой синтаксис работать не будет. Однако у каждого ресурса Terraform есть метапараметр под названием count. Это самая старая, простая ограниченная разновидность итератора в Terraform: она просто определяет, сколько копий ресурса нужно создать. Вот как с помощью этого параметра создать трех пользователей IAM:

```
resource "aws_iam_user" "example" {
  count = 3
  name  = "neo"
}
```

У этого кода есть одна проблема: у всех трех пользователей IAM будет одно и то же имя. Это приведет к ошибке, так как имена пользователей должны быть уникальными. Если бы у вас был доступ к стандартному циклу for, вы могли бы использовать индекс i, чтобы изменить каждое имя:

```
# Это просто псевдокод. Он не будет работать в Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo.${i}"
  }
}
```

Чтобы добиться того же в Terraform и получить индекс каждой итерации в цикле, можно воспользоваться ссылкой count.index:

```
resource "aws_iam_user" "example" {
  count = 3
  name  = "neo.${count.index}"
}
```

Если выполнить команду plan для представленного выше кода, можно увидеть, что Terraform собирается создать трех пользователей IAM с разными именами ("neo.0", "neo.1", "neo.2"):

```
Terraform will perform the following actions:
  # aws_iam_user.example[0] will be created
```

```
  + resource "aws_iam_user" "example" {
    + arn           = (known after apply)
    + force_destroy = false
    + id            = (known after apply)
    + name          = "neo.0"
    + path          = "/"
    + unique_id     = (known after apply)
  }
  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
    + arn           = (known after apply)
    + force_destroy = false
    + id            = (known after apply)
    + name          = "neo.1"
    + path          = "/"
    + unique_id     = (known after apply)
  }
  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
    + arn           = (known after apply)
    + force_destroy = false
    + id            = (known after apply)
    + name          = "neo.2"
    + path          = "/"
    + unique_id     = (known after apply)
  }
Plan: 3 to add, 0 to change, 0 to destroy.
```

Конечно, такое имя, как "neo.0", будет не очень полезным. Но если совместить count.index с некоторыми встроенными в Terraform функциями, каждую итерацию этого цикла можно изменить еще сильнее.

Например, все нужные вам имена пользователей IAM можно перечислить во входной переменной внутри live/global/iam/variables.tf:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

В языке программирования общего назначения с циклами и массивами вы бы назначили каждому пользователю IAM отдельное имя путем поиска значений в массиве var.user_names по индексу i:

```
# Это просто псевдокод. Он не будет работать в Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = vars.user_names[i]
  }
}
```

В Terraform то же самое можно сделать с помощью count в сочетании:

• с *синтаксисом доступа к массиву по индексу*, который похож на синтаксис большинства других языков:

```
ARRAY[<INDEX>]
```

Например, вот как взять из массива var.user_names элемент с индексом 1:

```
var.user_names[1]
```

• с *функцией* length. У Terraform есть встроенная функция под названием length, которая имеет следующий синтаксис:

```
length(<ARRAY>)
```

Как вы уже догадались, функция length возвращает количество элементов в заданном массиве. Она также работает со строками и ассоциативными массивами.

Если все это объединить, получится следующее:

```
resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}
```

Теперь, если выполнить команду plan, можно увидеть, что Terraform собирается создать трех пользователей IAM с уникальными именами:

```
Terraform will perform the following actions:
  # aws_iam_user.example[0] will be created
  + resource "aws_iam_user" "example" {
    + arn           = (known after apply)
    + force_destroy = false
    + id            = (known after apply)
    + name          = "neo"
    + path          = "/"
    + unique_id     = (known after apply)
  }
  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
    + arn           = (known after apply)
    + force_destroy = false
```

```
  + id           = (known after apply)
  + name         = "trinity"
  + path         = "/"
  + unique_id    = (known after apply)
 }
 # aws_iam_user.example[2] will be created
 + resource "aws_iam_user" "example" {
  + arn          = (known after apply)
  + force_destroy = false
  + id           = (known after apply)
  + name         = "morpheus"
  + path         = "/"
  + unique_id    = (known after apply)
 }
Plan: 3 to add, 0 to change, 0 to destroy.
```

Обратите внимание: если в ресурсе используется параметр count, он превращается в массив ресурсов. Поскольку aws_iam_user.example теперь является массивом пользователей IAM, вместо стандартного синтаксиса для чтения атрибутов (<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>) необходимо указывать, какой именно пользователь вас интересует. Для этого применяется тот же синтаксис доступа к элементам массива по его индексу:

<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>

Например, если вы хотите предоставить в качестве выходной переменной ARN одного из пользователей IAM, нужно сделать следующее:

```
output "neo_arn" {
  value       = aws_iam_user.example[0].arn
  description = "The ARN for user Neo"
}
```

Если вам нужны ARN всех пользователей IAM, вы должны указать символ * вместо индекса:

```
output "all_arns" {
  value       = aws_iam_user.example[*].arn
  description = "The ARNs for all users"
}
```

Если выполнить команду apply, вывод neo_arn будет содержать только ARN пользователя Neo, тогда как all_arns выведет список всех ARN:

```
$ terraform apply
(...)
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:
```

```
neo_arn = arn:aws:iam::123456789012:user/neo
all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
```

К сожалению, у параметра count есть два ограничения, которые делают его куда менее полезным. Во-первых, с помощью count можно пройтись по всему ресурсу, но при этом нельзя перебирать его вложенные блоки. *Вложенный блок* — это аргумент, который устанавливается внутри ресурса в следующем формате:

```
resource "xxx" "yyy" {
  <NAME> {
    [CONFIG...]
  }
}
```

NAME — это имя вложенного блока (например, tag), а CONFIG состоит из одного или нескольких аргументов, предназначенных специально для него (вроде key и value). Посмотрите, как устанавливаются теги в ресурсе aws_autoscaling_group:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"
  min_size = var.min_size
  max_size = var.max_size
  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
}
```

Каждый тег требует создания нового вложенного блока со значениями для key, value и propagate_at_launch. В предыдущем листинге вручную указан единственный тег, но можно разрешить пользователям передавать собственные. У вас может появиться соблазн воспользоваться параметром count для циклического перебора этих тегов и генерации динамических вложенных блоков tag, но, к сожалению, применение count внутри вложенного блока не поддерживается.

Второе ограничение параметра count даст о себе знать, когда вы попытаетесь его изменить. Рассмотрим созданный ранее список пользователей IAM:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

Представьте, что вы убрали из этого списка "trinity". Что произойдет при выполнении terraformplan?

```
$ terraform plan
(...)
Terraform will perform the following actions:
  # aws_iam_user.example[1] will be updated in-place
  ~ resource "aws_iam_user" "example" {
        id            = "trinity"
      ~ name          = "trinity" -> "morpheus"
    }
  # aws_iam_user.example[2] will be destroyed
  - resource "aws_iam_user" "example" {
      - id            = "morpheus" -> null
      - name          = "morpheus" -> null
    }
Plan: 0 to add, 1 to change, 1 to destroy.
```

Постойте, это не совсем то, чего мы ожидали! Вместо простого удаления пользователя IAM "trinity" вывод plan говорит о том, что Terraform собирается переименовать его в "morpheus" и затем удалить оригинального пользователя с этим именем. Что происходит?

Ресурс, в котором указан параметр count, превращается в список или массив ресурсов. К сожалению, Terraform определяет каждый элемент массива по его позиции (индексу). То есть после первого выполнения apply с именами трех пользователей внутреннее представление массива в Terraform выглядит примерно так:

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: trinity
aws_iam_user.example[2]: morpheus
```

Если удалить элемент посреди массива, все остальные элементы, которые шли за ним, смещаются назад на позицию. Поэтому после выполнения plan с именами двух пользователей внутреннее представление будет таким:

aws_iam_user.example[0]: neo
aws_iam_user.example[1]: morpheus

Обратите внимание на то, что у имени "morpheus" теперь индекс 1, а не 2. Terraform воспринимает индекс в качестве идентификатора ресурса, поэтому данное изменение можно перефразировать так: «переименовать элемент с индексом 1 в morpheus и удалить элемент с индексом 2». Иными словами, каждый раз, когда вы удаляете находящийся внутри списка ресурс, Terraform удаляет все ресурсы, которые следуют за ним, и воссоздает их заново, с нуля. Ох. Конечно, итоговым результатом будет именно то, о чем вы просили (то есть два пользователя IAM с именами "morpheus" и "neo"), но вряд ли вам хотелось бы достичь этого за счет удаления и изменения ресурсов.

Чтобы вы могли обойти эти два ограничения, в Terraform 0.12 появились выражения for_each.

Циклы с выражениями for_each

Выражение for_each позволяет выполнять циклический перебор списков, множеств и ассоциативных массивов с последующим созданием множественных копий либо всего ресурса, либо вложенного в него блока. Сначала рассмотрим первый вариант. Синтаксис выглядит так:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  for_each = <COLLECTION>
   [CONFIG ...]
}
```

PROVIDER — это имя провайдера (например, aws), TYPE — тип ресурса, который нужно создать в этом провайдере (скажем, instance), NAME — идентификатор, с помощью которого можно ссылаться на ресурс на разных участках кода Terraform (вроде my_instance), COLLECTION — множество или ассоциативный массив, который нужно перебрать в цикле (при использовании for_each в сочетании с ресурсом списки не поддерживаются), а CONFIG состоит из одного или нескольких аргументов, предназначенных специально для этого ресурса. Внутри CONFIG можно применять ссылки each.key и each.value для доступа к ключу и значению текущего элемента COLLECTION.

Например, так можно создать тех же трех пользователей IAM с помощью for_each:

```
resource "aws_iam_user" "example" {
  for_each = toset(var.user_names)
  name     = each.value
}
```

Обратите внимание на функцию toset, которая превращает список var.user_names во множество. Дело в том, что выражение for_each поддерживает множества и ассоциативные массивы только для ресурсов. При переборе этого множества оно предоставляет имя каждого пользователя в виде each.value. То же самое значение будет доступно и в each.key, хотя эта ссылка обычно используется только в ассоциативных массивах с ключами и значениями.

Ресурс, к которому применяется for_each, становится ассоциативным массивом ресурсов (а не обычным массивом, как в случае с count). Чтобы это продемонстрировать, заменим оригинальные выходные переменные all_arns и neo_arn новой, all_users:

```
output "all_users" {
  value = aws_iam_user.example
}
```

Вот что произойдет, если выполнить terraformapply:

```
$ terraform apply
(...)
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:
all_users = {
  "morpheus" = {
    "arn" = "arn:aws:iam::123456789012:user/morpheus"
    "force_destroy" = false
    "id" = "morpheus"
    "name" = "morpheus"
    "path" = "/"
    "tags" = {}
  }
  "neo" = {
    "arn" = "arn:aws:iam::123456789012:user/neo"
    "force_destroy" = false
    "id" = "neo"
    "name" = "neo"
    "path" = "/"
    "tags" = {}
  }
  "trinity" = {
    "arn" = "arn:aws:iam::123456789012:user/trinity"
    "force_destroy" = false
    "id" = "trinity"
```

```
    "name" = "trinity"
    "path" = "/"
    "tags" = {}
  }
}
```

Как видите, Terraform создает трех пользователей IAM, а выходная переменная all_users содержит ассоциативный массив, ключи которого (в данном случае имена пользователей) используются в for_each, а значения служат выходными переменными этого ресурса. Если хотите вернуть выходную переменную all_arns, нужно приложить дополнительные усилия, чтобы извлечь соответствующие ARN, добавив встроенную функцию values (которая возвращает только значения ассоциативного массива) и символ *:

```
output "all_arns" {
  value = values(aws_iam_user.example)[*].arn
}
```

Мы получим нужный нам вывод:

```
$ terraform apply
(...)
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
all_arns = [
  "arn:aws:iam::123456789012:user/morpheus",
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
]
```

Важно, что у вас теперь есть ассоциативный массив ресурсов с for_each вместо обычного массива ресурсов с count, так как теперь вы можете безопасно удалять его элементы. Например, если опять удалить "trinity" из списка var.user_names и выполнить terraformplan, результат будет следующим:

```
$ terraform plan
Terraform will perform the following actions:
  # aws_iam_user.example["trinity"] will be destroyed
  - resource "aws_iam_user" "example" {
      - arn           = "arn:aws:iam::123456789012:user/trinity" -> null
      - name          = "trinity" -> null
    }
Plan: 0 to add, 0 to change, 1 to destroy.
```

Вот так-то лучше! Теперь вы удаляете только тот ресурс, который вам нужен, не смещая все вокруг. Вот почему для создания множественных

копий ресурса почти всегда следует выбирать for_each вместо count.

Рассмотрим еще одно преимущество выражения for_each: его способность создавать множественные вложенные блоки внутри ресурса. Например, с его помощью можно динамически сгенерировать вложенные блоки tag для ASG в модуле webserver-cluster. Чтобы пользователь мог указывать собственные теги, добавим в файл modules/services/webserver-cluster/variables.tf новую входную переменную с ассоциативным массивом под названием custom_tags:

```
variable "custom_tags" {
  description = "Custom tags to set on the Instances in the ASG"
  type        = map(string)
  default     = {}
}
```

Далее установим некоторые пользовательские теги в промышленной среде в файле live/prod/services/webserver-cluster/main.tf:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"
  instance_type      = "m4.large"
  min_size           = 2
  max_size           = 10
  custom_tags = {
    Owner      = "team-foo"
    DeployedBy = "terraform"
  }
}
```

В этом листинге устанавливается несколько полезных тегов: Owner определяет, какой команде принадлежит ASG, а DeployedBy сигнализирует, что данная инфраструктура была развернута с помощью Terraform (это говорит о том, что ее не следует редактировать вручную; см. подраздел «Даже хороший план может оказаться неудачным» на с. 207). Обычно имеет смысл выработать систему тегов внутри своей команды и воплотить ее в виде кода с помощью модулей Terraform.

Итак, вы указали свои теги. Но как назначить их ресурсу aws_autoscaling_group? Для этого нужно циклически перебрать var.custom_tags. Пример этого показан в следующем псевдокоде:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
```

```
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"
  min_size = var.min_size
  max_size = var.max_size
  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
  # Это просто псевдокод. Он не будет работать в Terraform.
  for (tag in var.custom_tags) {
    tag {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

Псевдокод, приведенный выше, работать не будет. Вместо него мы воспользуемся выражением for_each. Так выглядит синтаксис динамической генерации вложенных блоков:

```
dynamic "<VAR_NAME>" {
  for_each = <COLLECTION>
  content {
    [CONFIG...]
  }
}
```

VAR_NAME — имя переменной, которая будет хранить значение каждой итерации (вместо each), COLLECTION — список или ассоциативный массив, который нужно перебрать, а блок content — это то, что генерируется при каждом проходе. Внутри блока content можно использовать ссылки <VAR_NAME>.key и <VAR_NAME>.value для доступа к ключу и соответственно к значению текущего элемента COLLECTION. Стоит отметить, что, когда вы применяете for_each в сочетании со списком, key содержит индекс, value — элемент с этим индексом. В случае с ассоциативным массивом key и value представляют собой одну из его пар типа «ключ — значение».

Соберем все это вместе и динамически сгенерируем блоки tag в ресурсе aws_autoscaling_group с помощью for_each:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"
  min_size = var.min_size
  max_size = var.max_size
  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
  dynamic "tag" {
    for_each = var.custom_tags
    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

Теперь, если выполнить terraformapply, план действий будет выглядеть примерно так:

```
$ terraform apply
Terraform will perform the following actions:
  # aws_autoscaling_group.example will be updated in-place
  ~ resource "aws_autoscaling_group" "example" {
        (...)
        tag {
            key                 = "Name"
            propagate_at_launch = true
            value               = "webservers-prod"
        }
      + tag {
          + key                 = "Owner"
          + propagate_at_launch = true
          + value               = "team-foo"
        }
      + tag {
          + key                 = "DeployedBy"
```

```
      + propagate_at_launch = true
      + value               = "terraform"
    }
  }
```

```
Plan: 0 to add, 1 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:
```

Введите yes, чтобы развернуть изменения. В веб-консоли EC2 должны появиться новые теги, как показано на рис. 5.1.

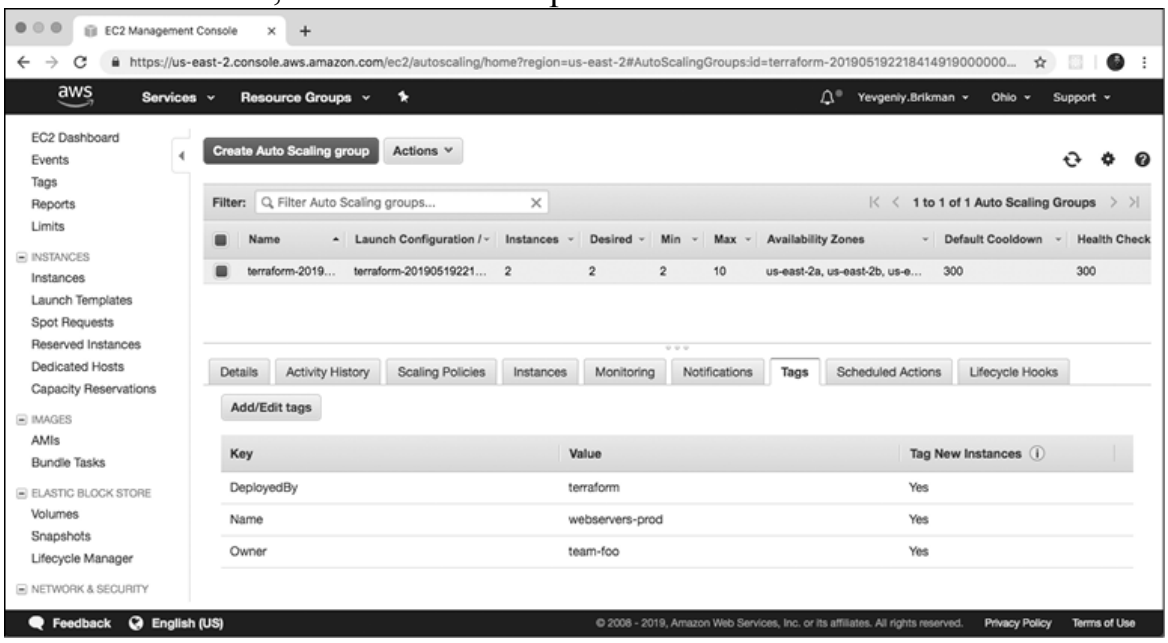


Рис. 5.1. Динамические теги группы автомасштабирования

Циклы на основе выражений for

Вы уже знаете, как циклически перебирать ресурсы и вложенные блоки. Но если с помощью цикла нужно сгенерировать лишь одно значение? Немного отвлечемся и рассмотрим некоторые примеры, не связанные с кластером веб-серверов. Представьте, что вы написали код Terraform, который принимает на вход список имен:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

Как бы вы перевели все эти имена в верхний регистр? В языке программирования общего назначения, таком как Python, можно было бы написать следующий цикл:

```
names = ["neo", "trinity", "morpheus"]
upper_case_names = []
for name in names:
   upper_case_names.append(name.upper())
print upper_case_names
# Выводит: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python позволяет выразить точно такой же код одной строчкой, используя синтаксис под названием *«абстракция списков» (list comprehension)*:

```
names = ["neo", "trinity", "morpheus"]
upper_case_names = [name.upper() for name in names]
print upper_case_names
# Выводит: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python также позволяет отфильтровать итоговый список по заданному выражению:

```
names = ["neo", "trinity", "morpheus"]
short_upper_case_names = [name.upper() for name in names if len(name) < 5]
print short_upper_case_names
# Выводит: ['NEO']
```

Terraform предлагает похожие возможности в виде выражения for (не путать с выражением for_each из предыдущего раздела). У него следующий базовый синтаксис:

```
[for <ITEM> in <LIST> : <OUTPUT>]
```

LIST — это список, который нужно перебрать, ITEM — имя локальной переменной, которое будет назначено каждому элементу списка, а OUTPUT — выражение, которое каким-то образом преобразует ITEM. Например, вот код Terraform для перевода списка имен в var.names в верхний регистр:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
output "upper_names" {
  value = [for name in var.names : upper(name)]
}
```

Если выполнить для этого кода команду terraformapply, получится следующий вывод:

```
$ terraform apply
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
upper_names = [
  "NEO",
  "TRINITY",
  "MORPHEUS",
]
```

Как и с абстракциями списков в Python, вы можете задать выражение для фильтрации полученного результата:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
output "short_upper_names" {
  value = [for name in var.names : upper(name) if length(name) < 5]
}
```

Выполнив terraformapply для этого кода, вы получите следующее:

```
short_upper_names = [
  "NEO",
]
```

Выражение for в Terraform также поддерживает циклический перебор ассоциативных массивов с использованием такого синтаксиса:

[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]

MAP — это ассоциативный массив, который нужно перебрать, KEY и VALUE — имена локальных переменных, которые назначаются каждой паре «ключ — значение» в MAP, а OUTPUT — выражение, которое каким-то образом преобразует KEY и VALUE. Например:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
  }
}
```

```
output "bios" {
  value = [for name, role in var.hero_thousand_faces : "${name} is the ${role}"]
}
```

Если выполнить terraformapply для этого кода, получится следующее:

```
map_example = [
  "morpheus is the mentor",
  "neo is the hero",
  "trinity is the love interest",
]
```

Выражение for может вернуть ассоциативный массив вместо списка, используя следующий синтаксис:

```
# Циклический перебор списков
[for <ITEM> in <MAP> : <OUTPUT_KEY> => <OUTPUT_VALUE>]
# Циклический перебор ассоциативных массивов
{for <KEY>, <VALUE> in <MAP> : <OUTPUT_KEY> => <OUTPUT_VALUE>}
```

Разница лишь в том, что: а) выражение помещается в фигурные скобки вместо прямоугольных; б) на каждой итерации выводится не только значение, но еще и ключ, отделенный от него стрелкой. Например, так можно перевести в верхний регистр все ключи и значения ассоциативного массива:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
  }
}
output "upper_roles" {
  value = {for name, role in var.hero_thousand_faces : upper(name) => upper(role)}
}
```

При выполнении этот код возвращает такой вывод:

```
upper_roles = {
  "MORPHEUS" = "MENTOR"
  "NEO" = "HERO"
  "TRINITY" = "LOVE INTEREST"
```

```
}
```

Циклы с использованием строковой директивы for

Ранее в этой книге вы узнали о строковой интерполяции, которая позволяет ссылаться на код Terraform внутри строки:

```
"Hello, ${var.name}"
```

С помощью *строковых директив* подобный синтаксис можно использовать и для управляющих конструкций (вроде циклов for и выражения if), но вместо знака доллара (${…}) перед фигурными скобками указывается знак процента (%{…}).

Строковая директива for имеет следующий синтаксис:

```
%{ for <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

COLLECTION — список или ассоциативный массив, который нужно перебрать, ITEM — имя локальной переменной, которое назначается каждому элементу COLLECTION, а BODY — это то, что выводится на каждой итерации (здесь можно ссылаться на ITEM). Например:

```
variable "names" {
  description = "Names to render"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
output "for_directive" {
  value = <<EOF
%{ for name in var.names }
  ${name}
%{ endfor }
EOF
}
```

Выполнив terraformapply, вы получите следующий вывод:

```
$ terraform apply
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
for_directive =
  neo
  trinity
  morpheus
```

Обратите внимание на дополнительные символы перехода на новую строку. В строковой директиве можно указать маркер ~, чтобы удалить все пробельные символы (пробелы и перенос строки) перед ней (если маркер

находится в начале директивы) или после нее (если маркер находится в конце директивы):

```
output "for_directive_strip_marker" {
  value = <<EOF
%{~ for name in var.names }
  ${name}
%{~ endfor }
EOF
}
```

Эта обновленная версия дает такой вывод:

```
for_directive_strip_marker =
  neo
  trinity
  morpheus
```

Условные выражения

Как и циклы, условные выражения в Terraform бывают нескольких видов, каждый из которых рассчитан немного на другой сценарий использования.

•*Параметр* count для условных ресурсов.

• *Выражения*for_each*и*for для условных ресурсов и их вложенных блоков.

• *Строковая директива* if для условных выражений внутри строк.

Рассмотрим их все по очереди.

Условные выражения с использованием параметра count

Параметр count, который вы видели ранее, позволяет создавать простые циклы. Но если проявить смекалку, тот же механизм можно использовать и для условных выражений. Мы начнем с рассмотрения конструкций if в пункте «Выражения if с использованием параметра count» ниже, а затем перейдем к выражениям if-else.

Выражения if с использованием параметра count

В главе 4 вы написали модуль Terraform, который можно применить в качестве «чертежа» для развертывания кластеров с веб-серверами. Этот модуль создавал группу автомасштабирования (ASG), балансировщик нагрузки (ALB), группы безопасности и ряд других ресурсов. Чего он *не* создавал, так это запланированного действия. Поскольку кластер нужно масштабировать только в промышленных условиях, вы определили ресурсы aws_autoscaling_schedule непосредственно в промышленной конфигурации в

файле live/prod/services/webserver-cluster/main.tf. Можно ли их определить в модуле webserver-cluster и затем создавать только для определенных пользователей?

Попробуем это сделать. Для начала добавим булеву входную переменную в файл modules/services/webserver-cluster/variables.tf, чтобы иметь возможность включать и выключать автомасштабирование в этом модуле:

```
variable "enable_autoscaling" {
  description = "If set to true, enable auto scaling"
  type        = bool
}
```

Теперь, если бы вы использовали язык программирования общего назначения, вы бы могли применить эту входную переменную в выражении if:

```
# Это просто псевдокод. Он не будет работать в Terraform.
if var.enable_autoscaling {
  resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
        scheduled_action_name     =  "${var.cluster_name}-scale-out-during-business-hours"
    min_size              = 2
    max_size              = 10
    desired_capacity      = 10
    recurrence            = "0 9 * * *"
    autoscaling_group_name = aws_autoscaling_group.example.name
  }
  resource "aws_autoscaling_schedule" "scale_in_at_night" {
    scheduled_action_name  = "${var.cluster_name}-scale-in-at-night"
    min_size              = 2
    max_size              = 10
    desired_capacity      = 2
    recurrence            = "0 17 * * *"
    autoscaling_group_name = aws_autoscaling_group.example.name
  }
}
```

Terraform не поддерживает выражения if, поэтому данный код работать не будет. Но того же результата можно добиться с помощью параметра count и двух особенностей языка.

• Если внутри ресурса параметру count присвоить значение 1, вы получите копию этого ресурса; если указать 0, этот ресурс вообще не будет создан.

• Terraform поддерживает *условные выражения* в формате <CONDITION>?<TRUE_VAL>:<FALSE_VAL>. Это *тернарный синтаксис*, с которым вы можете быть знакомы по другим языкам программирования. Он проверит булеву логику в CONDITION и, если результат равен true, вернет TRUE_VAL; в противном случае возвращается FALSE_VAL.

Объединив эти две идеи, мы можем обновить модуль webserver-cluster следующим образом:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  count = var.enable_autoscaling ? 1 : 0
      scheduled_action_name      =   "${var.cluster_name}-scale-out-during-
business-hours"
  min_size             = 2
  max_size             = 10
  desired_capacity     = 10
  recurrence           = "0 9 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}
resource "aws_autoscaling_schedule" "scale_in_at_night" {
  count = var.enable_autoscaling ? 1 : 0
  scheduled_action_name  = "${var.cluster_name}-scale-in-at-night"
  min_size             = 2
  max_size             = 10
  desired_capacity     = 2
  recurrence           = "0 17 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}
```

Если var.enable_autoscaling равно true, параметру count для каждого из ресурсов aws_autoscaling_schedule будет присвоено значение 1, поэтому оба они будут созданы в единственном экземпляре. Если var.enable_autoscaling равно false, параметру count для каждого из ресурсов aws_autoscaling_schedule будет присвоено значение 0, поэтому ни один из них создан не будет. Это именно та условная логика, которая нам нужна!

Теперь мы можем обновить использование этого модуля в тестовой среде (в файле live/stage/services/webserver-cluster/main.tf): выключим масштабирование, присвоив enable_autoscaling значение false:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name          = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
```

```
  instance_type       = "t2.micro"
  min_size            = 2
  max_size            = 2
  enable_autoscaling  = false
}
```

Аналогичным образом обновим использование этого модуля в промышленной среде (в файле live/prod/services/webserver-cluster/main.tf). Включим масштабирование, присвоив enable_autoscaling значение true (не забудьте также убрать пользовательские ресурсы aws_autoscaling_schedule, которые остались в промышленной среде после выполнения примеров главы 4):

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"
  instance_type       = "m4.large"
  min_size            = 2
  max_size            = 10
  enable_autoscaling  = true
  custom_tags = {
    Owner     = "team-foo"
    DeployedBy = "terraform"
  }
}
```

Этот подход хорошо работает в случае, если пользователь передает вашему модулю явное булево значение. Но если вместо этого передается результат более сложного сравнения, такого как проверка равенства строк? Рассмотрим более замысловатый пример.

Представьте, что вы хотите создать в рамках модуля webserver-cluster набор *оповещений CloudWatch*. Оповещение CloudWatch может доставляться с помощью разных механизмов (скажем, в виде электронного письма или текстового сообщения), если достигается заранее заданный порог. Например, ниже мы используем ресурс aws_cloudwatch_metric_alarm в файле modules/services/webserver-cluster/main.tf, чтобы создать оповещение, которое срабатывает, если загруженность процессора превышает 90 % на протяжении пяти минут:

```
resource "aws_cloudwatch_metric_alarm" "high_cpu_utilization" {
  alarm_name  = "${var.cluster_name}-high-cpu-utilization"
  namespace   = "AWS/EC2"
```

```
  metric_name = "CPUUtilization"
  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Average"
  threshold           = 90
  unit                = "Percent"
}
```

Это хорошо работает для показателя CPUUtilization. Но если нужно добавить еще одно оповещение, которое срабатывает, когда заканчиваются кредиты для процессора?46 Ниже это демонстрируется на примере нашего кластера веб-серверов:

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  alarm_name  = "${var.cluster_name}-low-cpu-credit-balance"
  namespace   = "AWS/EC2"
  metric_name = "CPUCreditBalance"
  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }
  comparison_operator = "LessThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Minimum"
  threshold           = 10
  unit                = "Count"
}
```

Но есть одна загвоздка: кредиты для процессора распространяются только на серверы типа tXXX (как t2.micro, t2.medium и т. д.). Более крупные типы серверов (вроде m4.large) эти кредиты не поддерживают и не отчитываются о показателе CPUCreditBalance. Поэтому, если вы создадите подобное оповещение для таких серверов, оно никогда не выйдет из состояния INSUFFICIENT_DATA. Возможно ли создавать оповещения только в случае, если var.instance_type начинается с буквы t?

Вы могли бы создать новую булеву входную переменную с именем var.is_t2_instance, но тогда бы она дублировала var.instance_type, а вы, скорее всего, забудете обновлять их вместе. Лучшая альтернатива — использование условного выражения:

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  count = format("%.1s", var.instance_type) == "t" ? 1 : 0
  alarm_name  = "${var.cluster_name}-low-cpu-credit-balance"
  namespace   = "AWS/EC2"
  metric_name = "CPUCreditBalance"
  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }
  comparison_operator = "LessThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Minimum"
  threshold           = 10
  unit                = "Count"
}
```

Код оповещения остается прежним, если не считать относительно сложного параметра count:

```
count = format("%.1s", var.instance_type) == "t" ? 1 : 0
```

Здесь используется функция format, которая извлекает первый символ из var.instance_type. Если это символ t (как в случае с t2.micro), параметру count присваивается значение 1; в противном случае параметр count будет равен 0. Таким образом, оповещение создается только для серверов, у которых действительно есть показатель CPUCreditBalance.

Выражения if-else с использованием параметра count

Теперь вы знаете, как создавать выражения if. Но что насчет if-else?

Ранее в этой главе вы создали несколько пользователей IAM с правом на чтение ресурсов EC2. Представьте, что вы хотите дать одному из них, Neo, еще и доступ к CloudWatch, но будет этот доступ только на чтение или еще и на запись, должен решать тот, кто применяет конфигурацию Terraform. Этот пример немного надуманный, но он позволяет легко продемонстрировать простую разновидность выражения if-else, в которой существенно лишь то, какая из веток, if или else, будет выполнена. В то же время остальному коду Terraform не нужно ничего об этом знать.

Вот правило IAM, которое разрешает доступ на чтение к CloudWatch:

```
resource "aws_iam_policy" "cloudwatch_read_only" {
  name   = "cloudwatch-read-only"
  policy = data.aws_iam_policy_document.cloudwatch_read_only.json
}
data "aws_iam_policy_document" "cloudwatch_read_only" {
```

```
  statement {
    effect    = "Allow"
    actions   = [
      "cloudwatch:Describe*",
      "cloudwatch:Get*",
      "cloudwatch:List*"
    ]
    resources = ["*"]
  }
}
```

А вот правило IAM, которое выдает полный доступ к CloudWatch (на чтение и запись):

```
resource "aws_iam_policy" "cloudwatch_full_access" {
  name   = "cloudwatch-full-access"
  policy = data.aws_iam_policy_document.cloudwatch_full_access.json
}
data "aws_iam_policy_document" "cloudwatch_full_access" {
  statement {
    effect    = "Allow"
    actions   = ["cloudwatch:*"]
    resources = ["*"]
  }
}
```

Наша цель — назначить одно из этих правил IAM пользователю neo с учетом значения новой входной переменной под названием give_neo_cloudwatch_full_access:

```
variable "give_neo_cloudwatch_full_access" {
  description = "If true, neo gets full access to CloudWatch"
  type        = bool
}
```

Если бы вы использовали язык программирования общего назначения, выражение if-else можно было бы написать в таком виде:

```
# Это просто псевдокод. Он не будет работать в Terraform.
if var.give_neo_cloudwatch_full_access {
                                resource                "aws_iam_user_policy_attachment"
"neo_cloudwatch_full_access" {
    user       = aws_iam_user.example[0].name
    policy_arn = aws_iam_policy.cloudwatch_full_access.arn
  }
} else {
```

```
  resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only"
{
    user       = aws_iam_user.example[0].name
    policy_arn = aws_iam_policy.cloudwatch_read_only.arn
  }
}
```

В Terraform для этого можно воспользоваться параметром count и условным выражением для каждого из ресурсов:

```
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access"
{
  count = var.give_neo_cloudwatch_full_access ? 1 : 0
  user       = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_full_access.arn
}
resource  "aws_iam_user_policy_attachment"  "neo_cloudwatch_read_only"
{
  count = var.give_neo_cloudwatch_full_access ? 0 : 1
  user       = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_read_only.arn
}
```

Этот код содержит два ресурса aws_iam_user_policy_attachment. У первого, который выдает полный доступ к CloudWatch, есть условное выражение. Если var.give_neo_cloudwatch_full_access равно true, оно возвращает 1, если нет — 0 (это частица if). Условное выражение второго ресурса, который выдает доступ на чтение, делает все наоборот: если var.give_neo_cloudwatch_full_access равно true, оно возвращает 0, если нет — 1 (это частица else).

Этот подход хорошо работает в ситуациях, когда вашему коду Terraform не нужно знать о том, какая из веток (if или else) на самом деле выполняется. Но если нужно обратиться к какому-нибудь выходному атрибуту ресурса, который возвращается из if или else? Представьте, к примеру, что вы хотите предложить пользователю на выбор два разных скрипта в разделе user_data модуля webserver-cluster. В настоящее время модуль webserver-cluster загружает скрипт user-data.sh из источника данных template_file:

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")
  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
```

```
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

Скрипт user-data.sh сейчас выглядит так:

```
#!/bin/bash
cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF
nohup busybox httpd -f -p ${server_port} &
```

Теперь представьте, что вы хотите позволить некоторым из своих кластеров использовать следующую, более короткую альтернативу под названием user-data-new.sh:

```
#!/bin/bash
echo "Hello, World, v2" > index.html
nohup busybox httpd -f -p ${server_port} &
```

Для загрузки этого скрипта вам понадобится новый источник данных template_file:

```
data "template_file" "user_data_new" {
  template = file("${path.module}/user-data-new.sh")
  vars = {
    server_port = var.server_port
  }
}
```

Вопрос в том, как разрешить пользователю модуля webserver-cluster выбрать один из этих скриптов user_data? Для начала в файл modules/services/webserver-cluster/variables.tf можно добавить новую булеву входную переменную:

```
variable "enable_new_user_data" {
  description = "If set to true, use the new User Data script"
  type        = bool
}
```

Если бы вы использовали язык программирования общего назначения, вы могли бы разместить в конфигурации запуска выражение if-else, которое выбирает между двумя вариантами template_file в user_data:

```
# Это просто псевдокод. Он не будет работать в Terraform.
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = var.instance_type
```

```
  security_groups = [aws_security_group.instance.id]
  if var.enable_new_user_data {
    user_data = data.template_file.user_data_new.rendered
  } else {
    user_data = data.template_file.user_data.rendered
  }
}
```

Чтобы это заработало в настоящем коде Terraform, сначала нужно воспользоваться приемом симуляции выражения if-else, который мы рассмотрели ранее, чтобы в итоге создавался только один источник данных template_file:

```
data "template_file" "user_data" {
  count = var.enable_new_user_data ? 0 : 1
  template = file("${path.module}/user-data.sh")
  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
data "template_file" "user_data_new" {
  count = var.enable_new_user_data ? 1 : 0
  template = file("${path.module}/user-data-new.sh")
  vars = {
    server_port = var.server_port
  }
}
```

Если атрибут var.enable_new_user_data равен true, будет создан источник data.template_file.user_data_new, но не data.template_file.user_data. Если он равен false, все будет наоборот. Вам остается лишь присвоить параметру user_data ресурса aws_launch_configuration источник данных template_file, который на самом деле существует. Для этого можно воспользоваться еще одним условным выражением:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data = (
    length(data.template_file.user_data[*]) > 0
      ? data.template_file.user_data[0].rendered
```

```
    : data.template_file.user_data_new[0].rendered
  )
  # Требуется при использовании группы автомасштабирования
  # в конфигурации запуска.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

Разберем по частям большое значение параметра user_data. Вначале взгляните на проверку булева условия:

```
length(data.template_file.user_data[*]) > 0
```

Обратите внимание, что оба источника данных используют параметр count и, следовательно, являются массивами, поэтому для работы с ними нужно использовать соответствующий синтаксис. Однако один из них имеет длину 1, а другой — 0, поэтому вы не можете напрямую обратиться по заданному индексу (например, data.template_file.user_data[0]), так как массив может оказаться пустым. В качестве решения можно воспользоваться выражением *, которое всегда возвращает массив (хоть и потенциально пустой), и затем проверить его длину.

Затем, учитывая длину массива, мы можем выбрать одно из следующих выражений:

```
? data.template_file.user_data[0].rendered
: data.template_file.user_data_new[0].rendered
```

Terraform выполняет отложенное вычисление условных результатов, поэтому значение true будет получено, только если условие истинно. В противном случае значение равно false. Таким образом, обращение к элементам user_data и user_data_new с индексом 0 будет безопасным, поскольку мы знаем, что вычислению подлежит только выражение с непустым массивом.

Можете теперь попробовать новый скрипт пользовательских данных в тестовой среде. Для этого присвойте параметру enable_new_user_data в файле live/stage/services/webserver-cluster/main.tf значение true:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
  instance_type       = "t2.micro"
  min_size            = 2
```

```
  max_size            = 2
  enable_autoscaling   = false
  enable_new_user_data = true
}
```

В промышленной среде можно оставить старую версию скрипта, установив параметру enable_new_user_data в файле live/prod/services/webserver-cluster/main.tf значение false:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"
  instance_type       = "m4.large"
  min_size            = 2
  max_size            = 10
  enable_autoscaling   = true
  enable_new_user_data = false
  custom_tags = {
    Owner      = "team-foo"
    DeployedBy = "terraform"
  }
}
```

Применение параметра count и встроенных функций для симуляции выражения if-else сродни грязному трюку, но этот подход работает достаточно хорошо. Как вы можете видеть в приведенном здесь коде, он позволяет скрыть от пользователей излишнюю сложность, чтобы они имели дело с простым и понятным API.

Условная логика с использованием выражений for_each и for

Теперь вы знаете, как применять к ресурсам условную логику с помощью параметра count, и догадываетесь, что похожая стратегия возможна и при использовании выражения for_each. Если передать for_each пустую коллекцию, получится ноль ресурсов и ноль вложенных блоков. Но если коллекция непустая, будет создан один или несколько ресурсов или вложенных блоков. Вопрос только в том, как определить, должна коллекция быть пустой или нет?

В качестве ответа можно объединить выражения for_each и for. Например, вспомните, как модуль webserver-cluster в файле modules/services/webserver-cluster/main.tf устанавливает теги:

```
dynamic "tag" {
```

```
  for_each = var.custom_tags
  content {
    key                 = tag.key
    value               = tag.value
    propagate_at_launch = true
  }
}
```

Если список var.custom_tags пустой, выражению for_each нечего перебирать, поэтому не будет задано ни одного тега. Иными словами, здесь у нас уже есть какая-то условная логика. Но мы можем пойти дальше и добавить к for_each выражение for:

```
dynamic "tag" {
  for_each = {
    for key, value in var.custom_tags:
    key => upper(value)
    if key != "Name"
  }
  content {
    key                 = tag.key
    value               = tag.value
    propagate_at_launch = true
  }
}
```

Вложенное выражение for циклически перебирает var.custom_tags, переводя каждое значение в верхний регистр (например, для однородности), и использует условную логику, чтобы отфильтровать любой параметр key, равный Name, поскольку модуль устанавливает свой собственный тег Name. Фильтрация значений в выражении for позволяет реализовать произвольную условную логику.

Выражение for_each почти всегда более предпочтительно для создания множественных копий ресурса по сравнению с параметром count, однако следует отметить, что с точки зрения условной логики присваивание count значений 0 или 1 обычно оказывается более простым, чем назначение for_each пустой/непустой коллекции. В связи с этим параметр count следует добавлять для условного создания ресурсов, тогда как for_each лучше подходит для любых других видов циклов и условных выражений.

Условные выражения с использованием строковой директивы if

Ранее в этой главе мы указывали строковую директиву для выполнения циклов внутри строк. Теперь рассмотрим еще одну:

%{ if <CONDITION> }<TRUEVAL>%{ endif }

CONDITION — это любое выражение, возвращающее булево значение, а TRUEVAL — выражение, которое нужно вывести, если CONDITION равно true. При желании можно также добавить блок else:

%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif }

FALSEVAL — это выражение, которое выводится, если CONDITION равно false. Например:

```
variable "name" {
  description = "A name to render"
  type        = string
}
output "if_else_directive" {
  value = "Hello, %{ if var.name != "" }${var.name}%{ else }(unnamed)%{ endif }"
}
```

Если выполнить команду terraformapply, присвоив World переменной name, получится следующее:

```
$ terraform apply -var name="World"
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
if_else_directive = Hello, World
```

Если выполнить команду terraformapply, присвоив переменной name пустую строку, результат будет таким:

```
$ terraform apply -var name=""
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
if_else_directive = Hello, (unnamed)
```

Развертывание с нулевым временем простоя

Итак, у ваших модулей есть простой и понятный API для развертывания кластера веб-серверов. Теперь возникает важный вопрос: как вы будете обновлять этот кластер? То есть как вы развернете в нем новый образ AMI (Amazon Machine Image) после внесения изменений в свой код? И как это сделать таким образом, чтобы пользователи не ощутили перебои в работе?

Для начала образ AMI нужно сделать доступным в виде входной переменной в файле modules/services/webser-vercluster/variables.tf. В реальных сценариях использования этого было бы достаточно, поскольку код самого веб-сервера находился бы в AMI. Но в наших упрощенных примерах весь код веб-сервера размещен в скрипте пользовательских

данных, а в качестве AMI применяется стандартный образ Ubuntu. Переход на другую версию Ubuntu будет не очень хорошей демонстрацией, поэтому, помимо новой входной переменной с AMI, можно также добавить входную переменную для изменения текста, который скрипт пользовательских данных возвращает из своего однострочного HTTP-сервера:

```
variable "ami" {
  description = "The AMI to run in the cluster"
  default     = "ami-0c55b159cbfafe1f0"
  type        = string
}
variable "server_text" {
  description = "The text the web server should return"
  default     = "Hello, World"
  type        = string
}
```

Когда мы упражнялись с выражениями if-else, вы создали два скрипта пользовательских данных. Вновь их объединим, чтобы не усложнять нашу задачу. Во-первых, удалите входную переменную enable_new_user_data из файла modules/services/webserver-cluster/variables.tf. Во-вторых, удалите из файла modules/services/webserver-cluster/main.tf ресурс template_file под названием user_data_new. В-третьих, оставаясь в том же файле, обновите другой ресурс template_file с именем user_data, чтобы больше не использовать входную переменную enable_new_user_data, и добавьте в его блок vars новую входную переменную server_text:

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")
  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
    server_text = var.server_text
  }
}
```

Теперь нужно сделать так, чтобы bash-скрипт modules/services/webserver-cluster/user-data.sh использовал эту переменную server_text в теге <h1>, который он возвращает:

```
#!/bin/bash
cat > index.html <<EOF
<h1>${server_text}</h1>
<p>DB address: ${db_address}</p>
```

```
<p>DB port: ${db_port}</p>
EOF
nohup busybox httpd -f -p ${server_port} &
```

Наконец, найдите конфигурацию запуска в файле modules/services/webserver-cluster/main.tf, присвойте параметру user_data оставшийся источник template_file (тот, что с именем user_data) и установите новую входную переменную ami параметру image_id:

```
resource "aws_launch_configuration" "example" {
  image_id        = var.ami
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data = data.template_file.user_data.rendered
    # Требуется при использовании группы автомасштабирования в конфигурации запуска.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

Теперь в тестовой среде (live/stage/services/webserver-cluster/main.tf) можно установить новые параметры, ami и server_text, и удалить enable_new_user_data:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-cluster"
  ami         = "ami-0c55b159cbfafe1f0"
  server_text = "New server text"
  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

В этом коде используется тот же AMI-образ Ubuntu, но у server_text теперь новое значение. Если выполнить команду plan, должен получиться такой результат:

```
Terraform will perform the following actions:
  # module.webserver_cluster.aws_autoscaling_group.ex will be updated in-place
```

```
  ~ resource "aws_autoscaling_group" "example" {
      id                  = "webservers-stage-terraform-20190516"
    ~ launch_configuration = "terraform-20190516" -> (known after apply)
      (...)
  }
  # module.webserver_cluster.aws_launch_configuration.ex must be replaced
+/- resource "aws_launch_configuration" "example" {
  ~ id                   = "terraform-20190516" -> (known after apply)
    image_id             = "ami-0c55b159cbfafe1f0"
    instance_type        = "t2.micro"
  ~ name                 = "terraform-20190516" -> (known after apply)
  ~ user_data            = "bd7c0a6" -> "4919a13" # forces replacement
    (...)
  }
Plan: 1 to add, 1 to change, 1 to destroy.
```

Как видите, Terraform хочет внести два изменения: заменить старую конфигурацию запуска новой с обновленным полем user_data и модифицировать уже имеющуюся группу автомасштабирования так, чтобы она ссылалась на новую конфигурацию запуска. Проблема в том, что во втором случае изменения не вступят в силу, пока ASG не запустит новые серверы EC2. Как же заставить ASG их развернуть?

Как вариант, вы можете уничтожить группу ASG (например, с помощью команды terraformdestroy) и затем создать ее заново (скажем, выполнив terraformapply). Но проблема этого способа в том, что после удаления старого экземпляра ASG и до загрузки нового ваши пользователи будут испытывать перебои в работе. Вместо этого лучше сделать *развертывание с нулевым временем простоя*. Для этого нужно сначала создать новую группу ASG и затем удалить старую. Оказывается, именно это делает параметр жизненного цикла create_before_destroy, с которым вы впервые столкнулись в главе 2. Рассмотрим, как организовать развертывание с нулевым временем простоя, используя этот параметр жизненного цикла47.

1. Сконфигурируйте параметр name для ASG так, чтобы он напрямую зависел от имени конфигурации запуска. При каждом изменении этой конфигурации (которое происходит в результате обновления AMI или пользовательских данных) будет меняться и ее название, а вместе с ним и имя ASG. Это заставит Terraform заменить группу автомасштабирования.

2. Присвойте true параметру create_before_destroy группы ASG, чтобы каждый раз, когда ее нужно заменить, система Terraform сначала создавала ее замену, и только потом удаляла оригинал.

3. Присвойте параметру min_elb_capacity группы ASG значение min_size, принадлежащее кластеру. Благодаря этому, прежде чем уничтожать оригинал, Terraform будет ждать, пока как минимум min_size серверов из новой группы ASG не пройдут проверку работоспособности в ALB.

Так должен выглядеть обновленный ресурс aws_autoscaling_group в файле modules/services/webserver-cluster/main.tf:

```
resource "aws_autoscaling_group" "example" {
  # Создаем явную зависимость от имени конфигурации запуска,
  # чтобы вместе с ней заменялась и группа ASG
                                                    name                 =
"${var.cluster_name}-${aws_launch_configuration.example.name}"
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"
  min_size = var.min_size
  max_size = var.max_size
  # Ждем, пока проверку работоспособности не пройдет как минимум
   # столько серверов, прежде чем считать завершенным развертывание
ASG
  min_elb_capacity = var.min_size
  # При замене этой группы ASG сначала создаем ее новую версию
  # и только потом удаляем старую
  lifecycle {
    create_before_destroy = true
  }
  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
  dynamic "tag" {
    for_each = {
      for key, value in var.custom_tags:
       key => upper(value)
      if key != "Name"
    }
    content {
      key                 = tag.key
      value               = tag.value
```

```
      propagate_at_launch = true
    }
  }
}
```

Если снова выполнить команду plan, результат будет примерно таким:

```
Terraform will perform the following actions:
    # module.webserver_cluster.aws_autoscaling_group.example must be replaced
+/- resource "aws_autoscaling_group" "example" {
    ~ id     = "example-2019" -> (known after apply)
    ~ name   = "example-2019" -> (known after apply) # forces replacement
      (...)
    }
    # module.webserver_cluster.aws_launch_configuration.example must be replaced
+/- resource "aws_launch_configuration" "example" {
    ~ id              = "terraform-2019" -> (known after apply)
      image_id        = "ami-0c55b159cbfafe1f0"
      instance_type   = "t2.micro"
    ~ name            = "terraform-2019" -> (known after apply)
    ~ user_data       = "bd7c0a" -> "4919a" # forces replacement
      (...)
    }
    (...)
Plan: 2 to add, 2 to change, 2 to destroy.
```

Главное, на что следует обратить внимание, — это строка forcesreplacement напротив параметра name ресурса aws_autoscaling_group. Это означает, что Terraform заменит этот ресурс новой группой ASG с новым образом AMI или новыми пользовательскими данными. Выполните команду apply, чтобы инициировать развертывание, и проследите за тем, как этот процесс работает.

Сначала у нас запущена оригинальная группа ASG, скажем, версии v1 (рис. 5.2).

Вы обновляете некоторые аспекты конфигурации запуска (например, переходите на образ AMI с кодом версии v2) и выполняете команду apply. Это заставляет Terraform начать процесс развертывания нового экземпляра ASG с кодом версии v2 (рис. 5.3).

После 1–2 минут серверы в новой группе ASG завершили загрузку, подключились к базе данных, зарегистрировались в ALB и начали проходить проверку работоспособности. На этом этапе обе версии вашего

приложения, v1 и v2, работают параллельно, и то, какую из них видит пользователь, зависит от того, куда ALB направил его запрос (рис. 5.4).

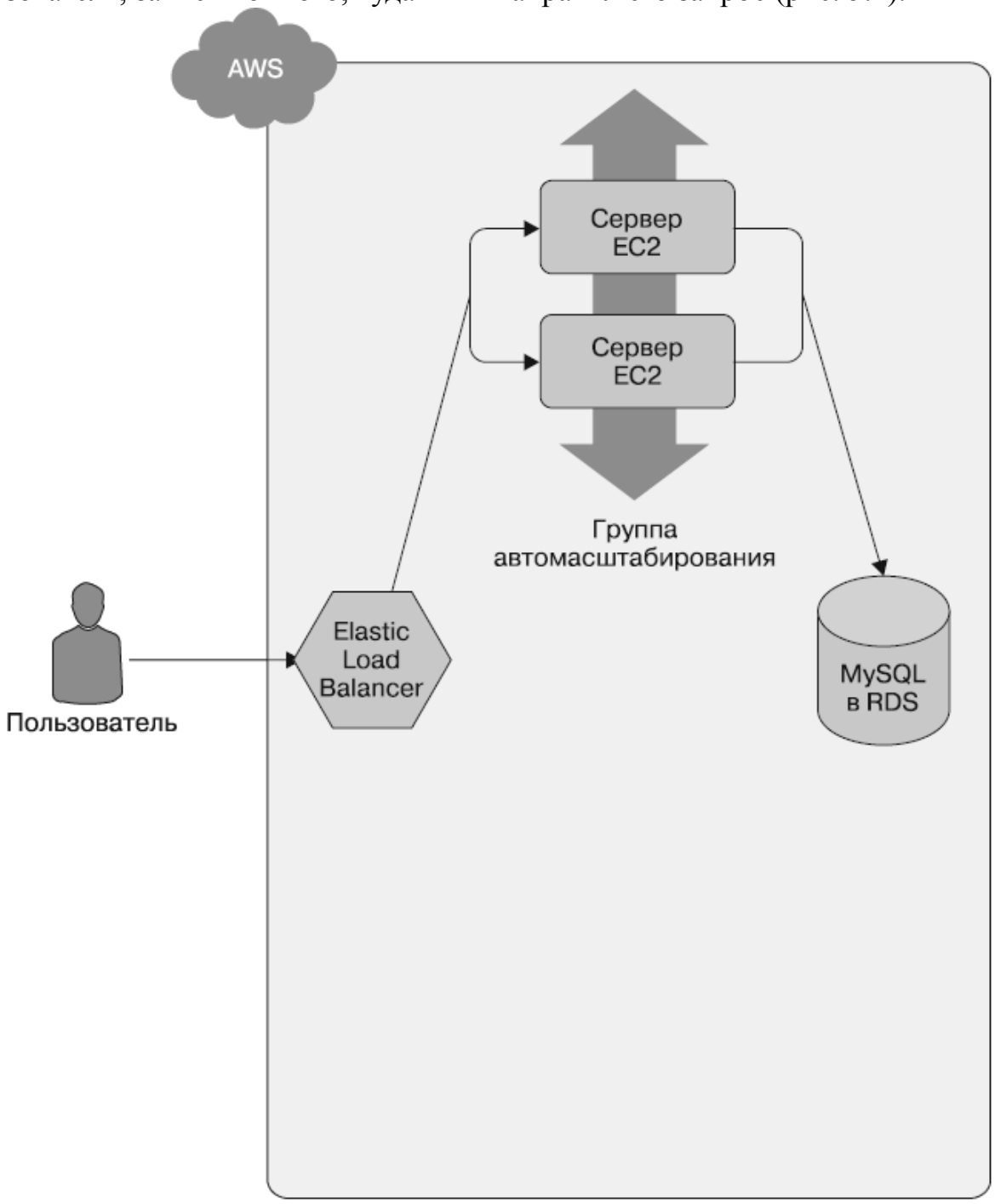


Рис. 5.2. Сначала у вас есть исходная группа ASG, выполняющая код версии v1

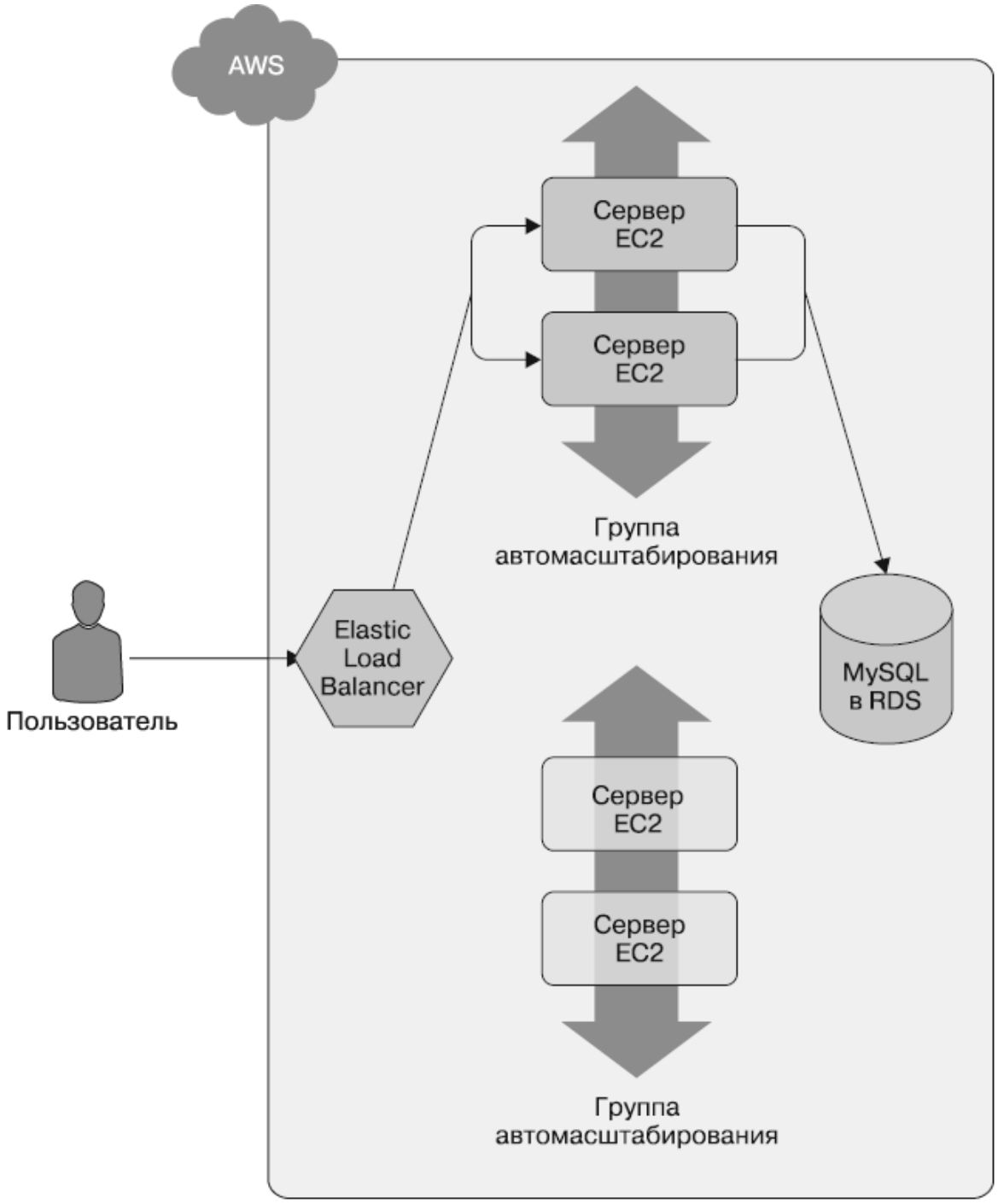


Рис. 5.3. Terraform начинает развертывание нового экземпляра ASG с кодом версии v2

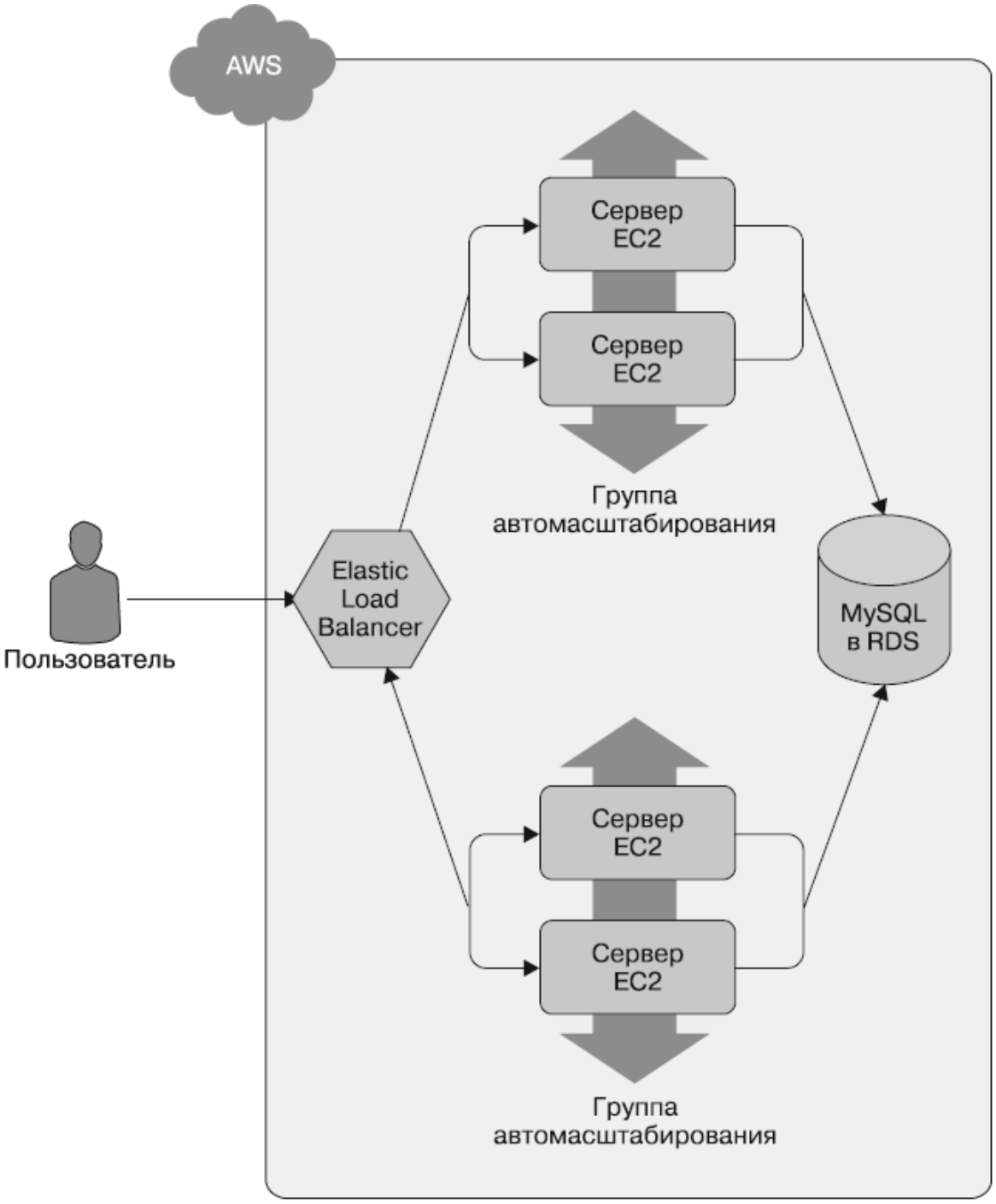


Рис. 5.4. Серверы в новой группе ASG загрузились, подключились к БД, зарегистрировались в ALB и начали обслуживать запросы

После того, как min_elb_capacity серверов из кластера ASG версии v2 зарегистрировалось в ALB, Terraform начинает удалять старую группу ASG. Сначала отменяется их регистрация в ALB, а затем они выключаются (рис. 5.5).

Через 1–2 минуты старая группа ASG исчезнет, и у вас останется только версия v2 вашего приложения в новом экземпляре ASG (рис. 5.6).

Весь процесс у вас всегда будут оставаться рабочие серверы, обслуживающие запросы от ALB, поэтому простоя не наблюдается. Открыв URL-адрес ALB в своем браузере, вы должны увидеть что-то похожее на рис. 5.7.

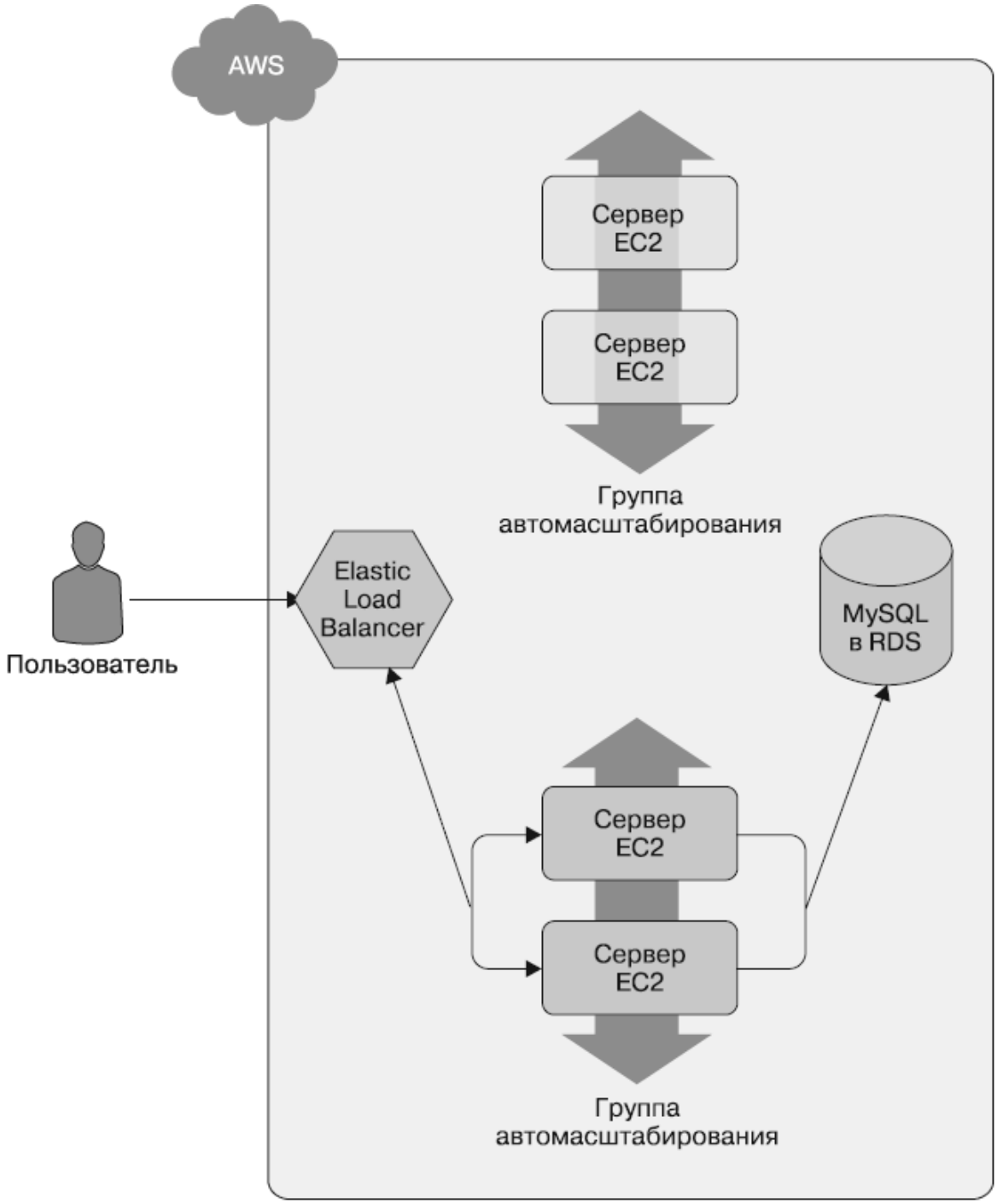


Рис. 5.5. Серверы из старой группы ASG начинают выключаться

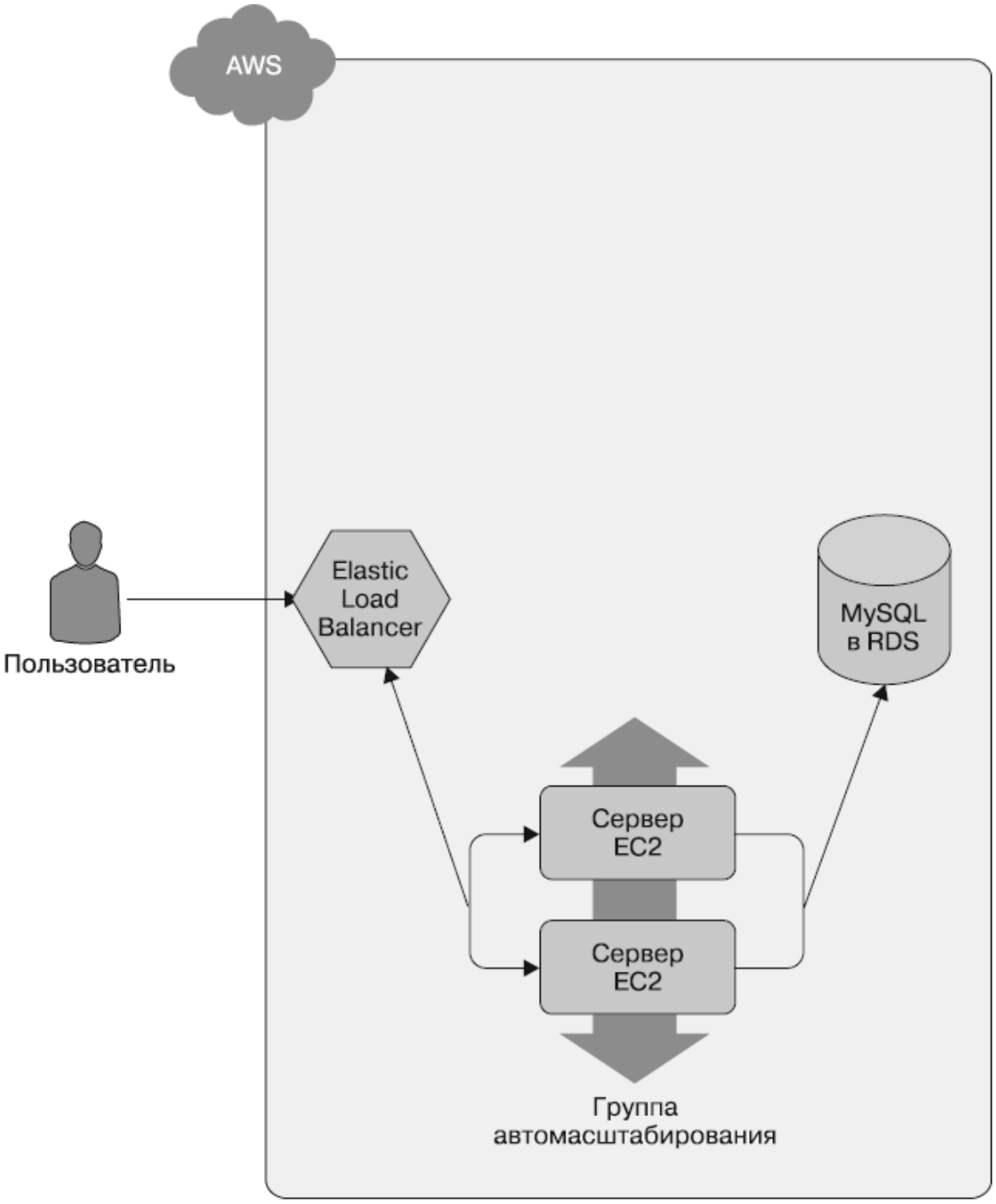


Рис. 5.6. Теперь остается только новая группа ASG, которая выполняет код версии v2

Получилось! Сервер с новым сообщением был развернут. Можете провести увлекательный эксперимент: внесите еще одно изменение в параметр server_text (например, поменяйте текст на foobar) и выполните команду apply. Если вы работаете в Linux/Unix/OS X, можете открыть отдельную вкладку терминала и запустить однострочный bash-скрипт,

который будет циклически вызывать curl, обращаясь к ALB раз в секунду. Это наглядно продемонстрирует, как происходит развертывание с нулевым временем простоя:

$ while true; do curl http://<load_balancer_url>; sleep 1; done

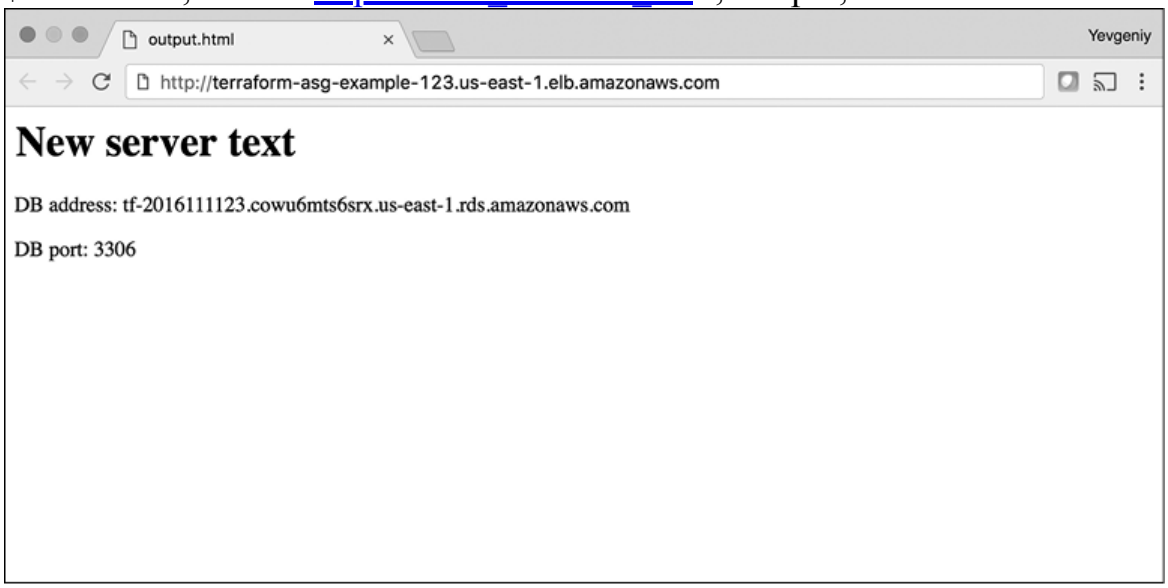


Рис. 5.7. Новый код уже развернут

Где-то на протяжении первой минуты ответ должен оставаться прежним: Newservertext. Затем вы заметите чередование Newservertext и foobar. Значит, новые серверы зарегистрировались в ALB и прошли проверку работоспособности. Еще через минуту сообщение Newservertext исчезнет, и вы будете видеть только foobar. Это означает, что старая группа ASG была отключена. Вывод будет выглядеть примерно так (для ясности я вывожу только содержимое тегов <h1>):

New server text
New server text
New server text
New server text
New server text
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text

foo bar
New server text
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar

У этого подхода есть еще одно преимущество: если во время развертывания что-то пойдет не так, Terraform автоматически откатит все назад. Например, если в версии v2 вашего приложения допущена ошибка, из-за которой оно не может загрузиться, серверы из новой группы ASG не будут зарегистрированы в ALB. Terraform будет ждать регистрации min_elb_capacity серверов из ASG v2 на протяжении отрезка времени длиной wait_for_capacity_timeout (по умолчанию десять минут). После этого развертывание считается неудачным, серверы ASG v2 удаляются, а Terraform завершает работу с ошибкой (тем временем версия v1 вашего приложения продолжает нормально работать в оригинальной группе ASG).

Подводные камни Terraform

После рассмотрения всех этих советов и приемов стоит сделать шаг назад и выделить несколько подводных камней, включая те, что связаны с циклами, выражениями if и методиками развертывания, а также с более общими проблемами, которые касаются Terraform в целом:

- параметры count и for_each имеют ограничения;
- ограничения развертываний с нулевым временем простоя;
- даже хороший план может оказаться неудачным;
- рефакторинг может иметь свои подвохи;
- отложенная согласованность согласуется… с отлагательством.

Параметры count и for_each имеют ограничения

В примерах этой главы параметр count и выражение for_each активно применяются в циклах и условной логике. Они хорошо себя показывают, но у них есть два важных ограничения, о которых необходимо знать.

- В count и for_each нельзя ссылаться ни на какие выходные переменные ресурса.
- count и for_each нельзя использовать в конфигурации модуля.

Рассмотрим их по очереди.

В count и for_each нельзя ссылаться ни на какие выходные переменные ресурса

Представьте, что нужно развернуть несколько серверов EC2 и по какой-то причине вы не хотите использовать ASG. Ваш код может быть таким:

```
resource "aws_instance" "example_1" {
  count         = 3
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Поскольку параметру count присвоено статическое значение, этот код заработает без проблем: когда вы выполните команду apply, он создаст три сервера EC2. Но если вам захотелось развернуть по одному серверу в каждой зоне доступности (Availability Zone или AZ) в рамках текущего региона AWS? Вы можете сделать так, чтобы ваш код загрузил список зон из источника данных aws_availability_zones и затем «циклически» прошелся по каждой из них и создал в ней сервер EC2, используя параметр count и доступ к массиву по индексу:

```
resource "aws_instance" "example_2" {
  count             = length(data.aws_availability_zones.all.names)
  availability_zone = data.aws_availability_zones.all.names[count.index]
  ami               = "ami-0c55b159cbfafe1f0"
  instance_type     = "t2.micro"
}
data "aws_availability_zones" "all" {}
```

Этот код тоже будет прекрасно работать, поскольку параметр count может без проблем ссылаться на источники данных. Но что произойдет, если количество серверов, которые вам нужно создать, зависит от вывода какого-то ресурса? Чтобы это продемонстрировать, проще всего взять ресурс random_integer, который, как можно догадаться по названию, возвращает случайное целое число:

```
resource "random_integer" "num_instances" {
  min = 1
  max = 3
}
```

Этот код генерирует случайное число от 1 до 3. Посмотрим, что случится, если мы попытаемся использовать вывод result этого ресурса в параметре count ресурса aws_instance:

```
resource "aws_instance" "example_3" {
  count         = random_integer.num_instances.result
  ami           = "ami-0c55b159cbfafe1f0"
```

```
  instance_type = "t2.micro"
}
```

Если выполнить для этого кода terraformplan, получится следующая ошибка:

```
Error: Invalid count argument
  on main.tf line 30, in resource "aws_instance" "example_3":
  30: count = random_integer.num_instances.result
The "count" value depends on resource attributes that cannot be determined
until apply, so Terraform cannot predict how many instances will be created.
To work around this, use the -target argument to first apply only the
resources that the count depends on.
```

Terraform требует, чтобы count и for_each вычислялись на этапе планирования, до создания или изменения каких-либо ресурсов. Это означает, что count и for_each могут ссылаться на литералы, переменные, источники данных и даже списки ресурсов (при условии, что их длину можно определить во время планирования), но не на вычисляемые выходные переменные ресурса.

count и for_each нельзя использовать в конфигурации модуля

Когда-нибудь у вас может появиться соблазн добавить параметр count в конфигурации модуля:

```
module "count_example" {
  source = "../../../../modules/services/webserver-cluster"
  count = 3
  cluster_name  = "terraform-up-and-running-example"
  server_port   = 8080
  instance_type = "t2.micro"
}
```

Этот код пытается использовать count внутри модуля, чтобы создать три копии ресурса webserver-cluster. Или, возможно, вам захочется сделать подключение модуля опциональным в зависимости от какого-нибудь булева условия, присвоив его параметру count значение 0. Такой код будет выглядеть вполне разумно, однако в результате выполнения terraformplan вы получите такую ошибку:

```
Error: Reserved argument name in module block
  on main.tf line 13, in module "count_example":
  13: count = 3
The name "count" is reserved for use in a future version of Terraform.
```

К сожалению, на момент выхода Terraform 0.12.6 использование count или for_each в ресурсе module не поддерживается. Согласно заметкам о

выпуске Terraform 0.12 (http://bit.ly/3257bv4) компания HashiCorp планирует добавить эту возможность в будущем, поэтому, в зависимости от того, когда вы читаете эту книгу, она уже может быть доступна. Чтобы узнать наверняка, почитайте журнал изменений Terraform по адресу https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md.

Ограничения развертываний с нулевым временем простоя

Использование блока create_before_destroy в сочетании с ASG является отличным решением для организации развертываний с нулевым временем простоя, если не считать один нюанс: правила автомасштабирования при этом не поддерживаются. Или, если быть более точным, это сбрасывает размер ASG обратно к min_size при каждом развертывании, что может стать проблемой, если вы использовали правила автомасштабирования для увеличения количества запущенных серверов.

Например, модуль webserver-cluster содержит пару ресурсов aws_autoscaling_schedule, которые в 9 утра увеличивают количество серверов в кластере с двух до десяти. Если выполнить развертывание, скажем, в 11 утра, новая группа ASG загрузится не с десятью, а всего с двумя серверами и будет оставаться в таком состоянии до 9 утра следующего дня.

Это ограничение можно обойти несколькими путями.

• Поменять параметр recurrence в aws_autoscaling_schedule с 09*** («запускать в 9 утра») на что-то вроде 0-599-17*** («запускать каждую минуту с 9 утра до 5 вечера»). Если в ASG уже есть десять серверов, повторное выполнение этого правила автомасштабирования ничего не изменит, что нам и нужно. Но если группа ASG развернута совсем недавно, это правило гарантирует, что максимум через минуту количество ее серверов достигнет десяти. Это не совсем элегантный подход, и большие скачки с десяти до двух серверов и обратно тоже могут вызвать проблемы у пользователей.

• Создать пользовательский скрипт, который применяет API AWS для определения количества активных серверов в ASG, вызвать его с помощью внешнего источника данных (см. пункт «Внешний источник данных» на с. 249) и присвоить параметру desired_capacity группы ASG значение, возвращенное этим скриптом. Таким образом, каждый новый экземпляр ASG всегда будет запускаться с той же емкостью, что и старый. Недостаток в том, что применение пользовательских скриптов ухудшает переносимость вашего кода Terraform и усложняет его обслуживание.

Конечно, в идеале в Terraform должна быть встроенная поддержка развертываний с нулевым временем простоя, но по состоянию на май 2019

года команда HashiCorp не планировала добавлять эту функциональность (подробности — по адресу github.com/hashicorp/terraform/issues/1552).

Корректный план может быть неудачно реализован

Иногда при выполнении команды plan получается вполне корректный план развертывания, однако команда apply возвращает ошибку. Попробуйте, к примеру, добавить ресурс aws_iam_user с тем же именем, которое вы использовали для пользователя IAM, созданного вами ранее в главе 2:

```
resource "aws_iam_user" "existing_user" {
  # Подставьте сюда имя уже существующего пользователя IAM,
  # чтобы попрактиковаться в использовании команды terraform import
  name = "yevgeniy.brikman"
}
```

Теперь, если выполнить команду plan, Terraform выведет на первый взгляд вполне разумный план развертывания:

```
Terraform will perform the following actions:
  # aws_iam_user.existing_user will be created
  + resource "aws_iam_user" "existing_user" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "yevgeniy.brikman"
      + path          = "/"
      + unique_id     = (known after apply)
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

Если выполнить команду apply, получится следующая ошибка:

```
Error: Error creating IAM User yevgeniy.brikman: EntityAlreadyExists:
User with name yevgeniy.brikman already exists.
  on main.tf line 10, in resource "aws_iam_user" "existing_user":
  10: resource "aws_iam_user" "existing_user" {
```

Проблема, конечно, в том, что пользователь IAM с таким именем уже существует. И это может случиться не только с пользователями IAM, но и почти с любым ресурсом. Возможно, кто-то создал этот ресурс вручную или с помощью командной строки, но, как бы то ни было, совпадение идентификаторов приводит к конфликтам. У этой ошибки существует множество разновидностей, которые часто застают врасплох новичков в Terraform.

Ключевым моментом является то, что команда terraformplan учитывает только те ресурсы, которые указаны в файле состояния Terraform. Если

ресурсы созданы каким-то другим способом (например, вручную, щелчком кнопкой мыши на консоли AWS), они не попадут в файл состояния и, следовательно, Terraform не будет их учитывать при выполнении команды plan. В итоге корректный на первый взгляд план окажется неудачным.

Из этого можно извлечь два урока.

•*Если вы уже начали работать с Terraform, не используйте ничего другого*. Если часть вашей инфраструктуры управляется с помощью Terraform, больше нельзя изменять ее вручную. В противном случае вы не только рискуете получить странные ошибки Terraform, но также сводите на нет многие преимущества IaC, так как код больше не будет точным представлением вашей инфраструктуры.

• *Если у вас уже есть какая-то инфраструктура, используйте команду* import. Если вы начинаете использовать Terraform с уже существующей инфраструктурой, ее можно добавить в файл состояния с помощью команды terraformimport. Так Terraform будет знать, какой инфраструктурой нужно управлять. Команда import принимает два аргумента. Первым служит адрес ресурса в ваших конфигурационных файлах. Здесь тот же синтаксис, что и в ссылках на ресурсы: <PROVIDER>_<TYPE>.<NAME> (вроде aws_iam_user.existing_user). Второй аргумент — это идентификатор ресурса, который нужно импортировать. Скажем, в качестве ID ресурса aws_iam_user выступает имя пользователя (например, yevgeniy.brikman), а ID ресурса aws_instance будет идентификатор сервера EC2 (вроде i-190e22e5). То, как импортировать ресурс, обычно указывается в документации внизу его страницы.

Ниже показана команда import, позволяющая синхронизировать ресурс aws_iam_user, который вы добавили в свою конфигурацию Terraform вместе с пользователем IAM в главе 2 (естественно, вместо yevgeniy.brikman нужно подставить ваше имя):

$ terraform import aws_iam_user.existing_user yevgeniy.brikman

Terraform обратится к API AWS, чтобы найти вашего пользователя IAM и создать в файле состояния связь между ним и ресурсом aws_iam_user.existing_user в вашей конфигурации Terraform. С этого момента при выполнении команды plan Terraform будет знать, что пользователь IAM уже существует, и не станет пытаться создать его еще раз.

Следует отметить, что, если у вас уже есть много ресурсов, которые вы хотите импортировать в Terraform, ручное написание кода и импорт каждого из них по очереди может оказаться хлопотным занятием. Поэтому стоит обратить внимание на такой инструмент, как Terraforming (http://terraforming.dtan4.net/), который может автоматически импортировать из учетной записи AWS код и состояние.

Рефакторинг может иметь свои подвохи

Рефакторинг — распространенная практика в программировании, когда вы меняете внутреннюю структуру кода, оставляя внешнее поведение без изменения. Это нужно, чтобы сделать код более понятным, опрятным и простым в обслуживании. Рефакторинг — это незаменимая методика, которую следует регулярно применять. Но, когда речь идет о Terraform или любом другом средстве IaC, следует крайне осторожно относиться к тому, что имеется в виду под «внешним поведением» участка кода, иначе возникнут непредвиденные проблемы.

Например, распространенный вид рефакторинга — замена названий переменных или функций более понятными. Многие IDE имеют встроенную поддержку рефакторинга и могут автоматически переименовать переменные и функции в пределах всего проекта. В языках программирования общего назначения это тривиальная процедура, о которой можно не задумываться, однако в Terraform с этим следует быть крайне осторожными, иначе можно столкнуться с перебоями в работе.

К примеру, у модуля webserver-cluster есть входная переменная cluster_name:

```
variable "cluster_name" {
  description = "The name to use for all the cluster resources"
  type        = string
}
```

Представьте, что вы начали использовать этот модуль для развертывания микросервиса с названием foo. Позже вам захотелось переименовать свой сервис в bar. Это изменение может показаться тривиальным, но в реальности из-за него могут возникнуть перебои в работе.

Дело в том, что модуль webserver-cluster использует переменную cluster_name в целом ряде ресурсов, включая параметр name двух групп безопасности и ALB:

```
resource "aws_lb" "example" {
  name               = var.cluster_name
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
  security_groups    = [aws_security_group.alb.id]
}
```

Если поменять параметр name в каком-то ресурсе, Terraform удалит старую версию этого ресурса и создаст вместо него новую. Но если таким ресурсом является ALB, в период между его удалением и загрузкой новой

версии у вас не будет механизма для перенаправления трафика к вашему веб-серверу. Точно так же, если удаляется группа безопасности, ваши серверы начнут отклонять любой сетевой трафик, пока не будет создана новая группа.

Еще одним видом рефакторинга, который вас может заинтересовать, является изменение идентификатора Terraform. Возьмем в качестве примера ресурс aws_security_group в модуле webserver-cluster:

```
resource "aws_security_group" "instance" {
  # (...)
}
```

Идентификатор этого ресурса называется instance. Представьте, что во время рефакторинга вы решили поменять его на более понятное (по вашему мнению) имя cluster_instance:

```
resource "aws_security_group" "cluster_instance" {
  # (...)
}
```

Что в итоге случится? Правильно: перебой в работе.

Terraform связывает ID каждого ресурса с идентификатором облачного провайдера. Например, iam_user привязывается к идентификатору пользователя IAM в AWS, а aws_instance — к ID сервера AWS EC2. Если изменить идентификатор ресурса (скажем, с instance на cluster_instance, как в случае с aws_security_group), для Terraform это будет выглядеть так, будто вы удалили старый ресурс и добавили новый. Если применить эти изменения, Terraform удалит старую группу безопасности и создаст другую, а между тем ваши серверы начнут отклонять любой сетевой трафик.

Вот четыре основных урока, которые вы должны извлечь из этого обсуждения.

•*Всегда используйте команду* plan. Ею можно выявить все эти загвоздки. Тщательно просматривайте ее вывод и обращайте внимание на ситуации, когда Terraform планирует удалить ресурсы, которые, скорее всего, удалять не стоит.

• *Создавайте, прежде чем удалять*. Если вы хотите заменить ресурс, хорошенько подумайте, нужно ли создавать замену до удаления оригинала. Если ответ положительный, в этом может помочь create_before_destroy. Того же результата можно добиться вручную, выполнив два шага: сначала добавить в конфигурацию новый ресурс и запустить команду apply, а затем удалить из конфигурации старый ресурс и воспользоваться командой apply еще раз.

• *Изменение идентификаторов требует изменения состояния*. Если вы хотите поменять идентификатор, связанный с ресурсом (например,

переименовать aws_security_group с instance на cluster_instance), избегая при этом удаления ресурса и создания его новой версии, необходимо соответствующим образом обновить файл состояния Terraform. Никогда не делайте этого вручную — используйте вместо этого команду terraformstate. При переименовании идентификаторов следует выполнить команду terraformstatemv, которая имеет следующий синтаксис:

terraform state mv <ORIGINAL_REFERENCE> <NEW_REFERENCE>

ORIGINAL_REFERENCE — это выражение, ссылающееся на ресурс в его текущем виде, а NEW_REFERENCE — то место, куда вы хотите его переместить. Например, при переименовании группы aws_security_group с instance на cluster_instance нужно выполнить следующую команду:

```
$ terraform state mv \
  aws_security_group.instance \
  aws_security_group.cluster_instance
```

Так вы сообщите Terraform, что состояние, которое ранее относилось к aws_security_group.instance, теперь должно быть связано с aws_security_group.cluster_instance. Если после переименования и запуска этой команды terraformplan не покажет никаких изменений, значит, вы все сделали правильно.

• *Некоторые параметры нельзя изменять*. Параметры многих ресурсов неизменяемые. Если попытаться их изменить, Terraform удалит старый ресурс и создаст вместо него новый. На странице каждого ресурса обычно указывается, что происходит при изменении того или иного параметра, поэтому не забывайте сверяться с документацией. Всегда используйте команду plan и рассматривайте целесообразность применения стратегии create_before_destroy.

Отложенная согласованность согласуется… с отлагательством

API некоторых облачных провайдеров, таких как AWS, асинхронные и имеют отложенную согласованность. *Асинхронность* означает, что интерфейс может сразу же вернуть ответ, не дожидаясь завершения запрошенного действия. *Отложенная согласованность* значит, что для распространения изменений по всей системе может понадобиться время; пока это происходит, ваши ответы могут быть несогласованными и зависеть от того, какая реплика источника данных отвечает на ваши API-вызовы.

Представьте, к примеру, что вы делаете API-вызов к AWS с просьбой создать сервер EC2. API вернет «успешный» ответ (201Created) практически мгновенно, не дожидаясь создания самого сервера. Если вы сразу же попытаетесь к нему подключиться, почти наверняка ничего не получится, поскольку в этот момент AWS все еще инициализирует ресурсы или, как

вариант, сервер еще не загрузился. Более того, если вы сделаете еще один вызов, чтобы получить информацию об этом сервере, может прийти ошибка (404NotFound). Дело в том, что сведения об этом сервере EC2 все еще могут распространяться по AWS, чтобы они стали доступными везде, придется подождать несколько секунд.

При каждом использовании асинхронного API с отложенной согласованностью вы должны периодически повторять свой запрос, пока действие не завершится и не распространится по системе. К сожалению, AWS SDK не предоставляет для этого никаких хороших инструментов, и проект Terraform раньше страдал от множества ошибок вроде 6813 (https://github.com/hashicorp/terraform/issues/6813):

$ terraform apply

aws_subnet.private-persistence.2: InvalidSubnetID.NotFound:

The subnet ID 'subnet-xxxxxxx' does not exist

Иными словами, вы создаете ресурс (например, подсеть) и затем пытаетесь получить о нем какие-то сведения (вроде ID только что созданной подсети), а Terraform не может их найти. Большинство из таких ошибок (включая 6813) уже исправлены, но время от времени они все еще проявляются, особенно когда в Terraform добавляют поддержку нового типа ресурсов. Это раздражает, но в большинстве случаев не несет никакого вреда. При повторном выполнении terraformapply все должно заработать, поскольку к этому моменту информация уже распространится по системе.

Резюме

Несмотря на свою декларативную сущность, Terraform включает целый ряд конструкций, которые делают его на удивление гибким и выразительным. Это, к примеру, касается переменных и модулей (см. главу 4), параметра count, выражений for_each, for и create_before_destroy, а также встроенных функций, которые вы могли видеть в этой главе. Вы познакомились с множеством приемов симуляции условных выражений, поэтому потратьте некоторое время на чтение документации по функциям (https://www.terraform.io/docs/configuration/functions.html) и позвольте разыграться своему «внутреннему хакеру». Главное, не переусердствуйте, так как кто-то все равно должен поддерживать ваш код; просто постарайтесь научиться создавать аккуратные и изящные API для своих модулей.

Теперь перейдем к главе 6, в которой важный акцент делается на готовности к промышленному использованию. Речь пойдет о модулях, на которые ваша компания может положиться.

46 О кредитах для процессора можно почитать на сайте EC2 по адресу amzn.to/2lTuvs5.

47 Автором этой методики является Пол Хинзе (bit.ly/2lksQgv).

6. Код Terraform промышленного уровня

Построение инфраструктуры промышленного уровня — сложный и напряженный процесс, который отнимает много времени. Под *инфраструктурой промышленного уровня* я имею в виду серверы, хранилища данных, балансировщики нагрузки, механизмы безопасности, средства мониторинга и оповещения, процедуры сборки и любые другие составляющие вашей технологии, без которых не может обойтись ваша компания. На вас полагаются с надеждой на то, что ваша инфраструктура справится с возрастающей нагрузкой, не потеряет данные во время перебоев в работе и не скомпрометирует эти данные, если кто-то попытается ее взломать. Если надежда не оправдается, ваша компания может обанкротиться. При обсуждении инфраструктуры промышленного уровня в этой главе помните, что стоит на кону.

Я имел возможность работать с сотнями компаний. Исходя из этого опыта, при оценке времени, которое может уйти на разработку проекта промышленного уровня, нужно учитывать следующее.

• Если вы хотите развернуть сервис, который будет полностью управляться третьим лицом (например, запустить MySQL в AWS Relational Database Service (RDS)), на его подготовку к промышленному использованию может уйти от одной до двух недель.

• Если вы хотите сами запускать свое распределенное приложение, которое не хранит состояние, как в случае с кластером Node.js без каких-либо локальных данных (например, данные могут храниться в RDS), запущенным поверх группы автомасштабирования AWS (ASG), для его подготовки к промышленному использованию понадобится где-то вдвое больше времени, или около четырех недель.

• Если вы хотите сами запускать свое распределенное приложение с сохранением состояния, как в случае с кластером Amazon Elasticsearch (Amazon ES) поверх ASG, который хранит данные на локальных дисках, вам потребуется на порядок больше времени — от двух до четырех месяцев.

• Если вы хотите разработать целую архитектуру, включая все свои приложения, хранилища данных, балансировщики нагрузки, мониторинг, механизм оповещения, безопасность и т. д., необходимое время увеличивается еще на 1–2 порядка — примерно от 6 до 36 месяцев работы. В случае с мелкими компаниями этот срок приближается к шести месяцам, а у крупных организаций на это обычно уходят годы.

Эти данные собраны в табл. 6.1.

Таблица 6.1. Сколько займет построение инфраструктуры промышленного уровня с нуля

Тип инфраструктуры	**Пример**	**Примерные сроки**
Управляемый сервис	Amazon RDS	1–2 недели
Распределенная система с самостоятельным размещением (без состояния)	Кластер приложений Node.js	2–4 недели
Распределенная система с самостоятельным размещением (с состоянием)	Amazon ES	2–4 месяца
Целая архитектура	Приложения, хранилища данных, балансировщики нагрузки, мониторинг и т. д.	6–36 месяцев

Если вы еще не проходили через процесс построения инфраструктуры промышленного уровня, эти цифры могут вас удивить. Я часто встречаю реакции наподобие следующих.

• «Так долго? Как такое возможно?»

• «Я могу развернуть сервер на <облако> за пару минут. На то, чтобы сделать все остальное, точно не могут уйти месяцы!»

• И повсеместно от слишком самоуверенных инженеров: «Не сомневаюсь, что эти цифры справедливы для других людей, но я смогу сделать это за несколько дней».

Тем не менее человек с опытом большой облачной миграции или построения совершенно новой инфраструктуры с нуля знает, что эти цифры не то что реальные, но еще и оптимистичные, — на самом деле это идеальный случай. Если в вашей команде нет людей с глубокими знаниями в области построения инфраструктуры промышленного уровня или вам приходится заниматься сразу десятком разных направлений и вы не успеваете сосредоточиться на каждом из них, этот процесс может затянуться еще сильнее.

В этой главе я объясню, почему построение инфраструктуры промышленного уровня занимает столько времени, что собой представляет этот промышленный уровень и какие методики лучше всего подходят для

создания универсальных модулей, готовых к использованию в реальных условиях.

• Почему построение инфраструктуры промышленного уровня требует так много времени.

• В чем состоят требования к инфраструктуре промышленного уровня.

Мы также рассмотрим инфраструктурные модули промышленного уровня:

• мелкие модули;

• компонуемые модули;

• тестируемые модули;

• модули, готовые к выпуску;

• модули вне Terraform.

Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

Почему построение инфраструктуры промышленного уровня требует так много времени

Как известно, оценка сроков разработки программных проектов очень приблизительна. И это вдвойне касается проектов DevOps. Небольшое исправление, которое, как можно было бы ожидать, требует не более пяти минут, на самом деле занимает целый день. Мелкая функция, на реализацию которой, по вашим прикидкам, нужно потратить день работы, растягивается на две недели. Приложение, которое, как вы надеялись, должно быть выпущено за две недели, все еще не готово спустя полгода. Проекты, связанные с инфраструктурой и DevOps, идеально иллюстрируют закон Хофштадтера48:

Закон Хофштадтера: работа всегда занимает больше времени, чем вы ожидаете, даже с учетом закона Хофштадтера.

Даглас Хофштадтер

Мне кажется, у этого есть три основные причины. Прежде всего, DevOps как индустрия все еще находится в каменном веке. Я не пытаюсь никого обидеть, а просто хочу указать на незрелость этих технологий. Термины «облачные вычисления», «инфраструктура как код» и DevOps появились лишь во второй половине 2000-х годов, и инструменты вроде

Terraform, Docker, Packer и Kubernetes впервые были выпущены в середине или конце 2010-х. Все эти средства и методики относительно новые и быстро развивающиеся. Значит, они не совсем зрелые и глубокими знаниями в этой области обладает не так уж много людей, поэтому неудивительно, что на проекты уходит больше ожидаемого времени.

Вторая причина — DevOps, похоже, особенно подвержен эффекту *стрижки быка (yak shaving)*. Если вам незнаком этот термин, уверяю, вы его полюбите (и возненавидите). Лучшее его определение, которое мне встречалось, опубликовано в блоге Сета Година49:

«Я хочу сегодня помыть машину».

«Ох, шланг треснул еще зимой. Придется купить новый в Home Depot».

«Но Home Depot на другом конце моста, а проезд по нему платный, и без платежной карточки будет дорого».

«Ой, точно! Я могу одолжить карточку у своего соседа…»

«Но Боб не даст мне свою карточку, пока я не верну ему подушку-обнимушку, которую взял мой сын».

«А не вернули мы ее потому, что из нее вылезла часть набивки, и, чтобы набить ее снова, мне нужна бычья шерсть».

Таким образом, вы докатились до того, что стрижете быка в зоопарке — и все для того, чтобы помыть свою машину.

Сет Годин

Стрижка быка состоит из всех этих мелких и, на первый взгляд, не связанных между собой задач, которые нужно выполнить до того, как приступать к изначально запланированной работе. Если вы разрабатываете программное обеспечение и особенно если вы работаете в сфере DevOps, подобного рода ситуации вам встречались тысячу раз. Вы беретесь за развертывание исправления для небольшой опечатки, чем внезапно провоцируете ошибку в конфигурации приложения. После нескольких часов на StackOverflow вы решили проблему с TLS и теперь пробуете развернуть свой код еще раз, но тут у вас начинаются проблемы с системой развертывания. Чтобы в них разобраться, вы тратите еще несколько часов и обнаруживаете, что причина — в устаревшей версии Linux. Не успев оглянуться, вы беретесь за обновление операционной системы на целой армии серверов — и все для того, чтобы «быстро» развернуть исправление небольшой опечатки.

DevOps, по всей видимости, особенно подвержен подобного рода конфузам со стрижкой быков. Отчасти из-за незрелости данной технологии и современных подходов к проектированию систем, которые часто требуют жесткого связывание и дублирования инфраструктуры. Любое изменение, которое вы делаете в мире DevOps, сродни попытке выдернуть один USB-

кабель из коробки с запутанными проводами — обычно таким образом вы вытягиваете все содержимое коробки. Но в какой-то мере это связано с тем, что термин DevOps охватывает поразительно широкий спектр тем: от сборки до развертывания, обеспечения безопасности и т. д.

Это подводит нас к третьей причине, почему работа в сфере DevOps занимает столько времени. Первые две причины (DevOps в каменном веке и стрижка быка) можно классифицировать как ненужные сложности. *Ненужные сложности* — это проблемы, связанные с определенными инструментами и процессами, которые вы сами выбрали. Их противоположность — *имманентные сложности* — проблемы, присущие тому, над чем вы работаете50. Например, если вы используете C++ для написания алгоритмов биржевых торгов, проблемы с ошибками выделения памяти будут необязательными трудностями — если бы вы выбрали другой язык программирования с автоматическим управлением памятью, этих проблем у вас бы вообще не было. С другой стороны, подбор алгоритма, способного преуспеть на рынке, является имманентной сложностью — эту проблему придется решать независимо от того, какой язык программирования был выбран.

Третья причина, почему DevOps занимает столько времени (имманентная сложность этой проблемы), состоит в том, что для подготовки инфраструктуры к промышленному использованию необходимо решить довольно длинный список задач. Проблема в том, что множество разработчиков не догадываются о большей части пунктов в этом списке, поэтому при оценке сроков выполнения проекта они забывают об огромном количестве критически важных деталей, которые отнимают много времени. Этому списку посвящен следующий раздел.

Требования к инфраструктуре промышленного уровня

Попробуйте такой забавный эксперимент: пройдитесь по своей компании и поспрашивайте, каковы требования для перехода в промышленную среду. Чаще всего первые пять человек дадут пять разных ответов. Один из них упомянет необходимость в измерении показателей и оповещениях; другой расскажет о планировании емкости и высокой доступности; кто-то начнет разглагольствовать об автоматических тестах и разборе кода; а кто-то затронет тему шифрования, аутентификации и укрепления серверов. Если вам повезет, один из респондентов может вспомнить о резервном копировании данных и агрегации журнальных записей. У большинства компаний нет четкого списка требований к выпуску промышленной системы. Значит, каждый элемент инфраструктуры

развертывается немного по-своему и ему может недоставать каких-то критически важных функций.

Чтобы улучшить эту ситуацию, хочу поделиться с вами контрольным списком задач для подготовки инфраструктуры промышленного уровня (табл. 6.2). Этот список охватывает большинство ключевых моментов, которые необходимо учитывать при развертывании инфраструктуры в промышленной среде.

Таблица 6.2. Список задач для подготовки инфраструктуры промышленного уровня

Задача	**Описание**	**Примеры инструментов**
Установка	Установите исполняемые файлы ПО и все зависимости	Bash, Chef, Ansible, Puppet
Конфигурация	Сконфигурируйте ПО на этапе выполнения, включая настройки портов, сертификаты TLS, обнаружение сервисов, выбор центральных и дополнительных серверов, репликацию и т. д.	Bash, Chef, Ansible, Puppet
Инициализация	Инициализируйте инфраструктуру, включая серверы, балансировщики нагрузки, сетевую конфигурацию, параметры брандмауэра, права доступа IAM и т. д.	Terraform, CloudFormation
Развертывание	Разверните сервис поверх инфраструктуры. Выкатывайте обновления с нулевым временем простоя, включая «сине-зеленые», скользящие и канареечные развертывания	Terraform, CloudFormation, Kubernetes, ECS
Высокая доступность	Система должна выдерживать перебои в работе отдельных процессов, серверов, сервисов, вычислительных центров и регионов	Несколько вычислительных центров и регионов, репликация, автомасштабирование, распределение нагрузки
Масштабируемость	Масштабируйте систему в зависимости от нагрузки. Это касается как горизонтального (больше серверов), так и вертикального масштабирования (более крупные серверы)	

Автомасштабирование, репликация, сегментирование, кэширование, стратегия «разделяй и властвуй»

Производительность

Оптимизируйте использование процессора, памяти, диска, сети и графического адаптера. Это относится к ускорению запросов, эталонному тестированию, нагрузочному тестированию и профилированию

Dynatrace, valgrind, VisualVM, ab, Jmeter

Сеть

Сконфигурируйте статические и динамические IP-адреса, порты, обнаружение сервисов, брандмауэры, DNS, а также доступ по SSH и VPN

Облака VPC, брандмауэры, маршрутизаторы, регистраторы DNS, OpenVPN

Безопасность

Шифрование при передаче (TLS) и на диске, аутентификация, авторизация, управление конфиденциальными данными, укрепление серверов

ACM, Let’s Encrypt, KMS, Cognito, Vault, CIS

Показатели

Показатели доступности, бизнес-показатели, показатели приложения, показатели серверов, события, наблюдаемость, трассировка и оповещения

CloudWatch, DataDog, New Relic, Honeycomb

Журнальные записи

Организуйте чередование журнальных файлов на диске. Агрегируйте журнальные данные в центральном месте

CloudWatch Logs, ELK, Sumo Logic, Papertrail

Резервное копирование и восстановление

Проводите плановое резервное копирование БД, кэшей и других данных. Реплицируйте данные для разделения регионов или учетных записей

RDS, ElastiCache, репликация

Оптимизация расходов

Выбирайте подходящие типы серверов, используйте прерываемые и резервируемые серверы, применяйте автомасштабирование и избавляйтесь от ненужных ресурсов

Автомасштабирование, прерываемые и резервируемые серверы

Документация

Документируйте свой код, архитектуру и практикуемые методики. Создавайте плейбуки на случай разных происшествий

README, вики, Slack

Тесты

Пишите для своего инфраструктурного кода автоматические тесты. Выполняйте их после каждой фиксации кода и по ночам

Terratest, inspec, serverspec, kitchen-terraform

Большинству разработчиков известно о первых нескольких задачах: установке, конфигурации, инициализации и развертывании. А вот то, что идет дальше, застает людей врасплох. Например, подумали ли вы об устойчивости своего сервиса и о том, что произойдет в случае поломки сервера? А если выйдет из строя балансировщик нагрузки или весь вычислительный центр? Сетевые задачи тоже славятся своими подводными камнями: VPC, VPN, обнаружение сервисов и доступ по SSH — все это неотъемлемые элементы инфраструктуры, на подготовку которых могут уйти месяцы; но, несмотря на это, их часто полностью игнорируют при планировании проектов об оценке сроков. О задачах безопасности, таких как шифрование данных при передаче с помощью TLS, настройка аутентификации и выработка механизма хранения конфиденциальных данных, тоже часто вспоминают в последний момент.

Каждый раз, когда вы начинаете работать над новым участком инфраструктуры, не забывайте пройтись по этому списку. Не все пункты обязательные в каждом конкретном случае, но вы должны сознательно и явно документировать, какие компоненты вы реализовали, а какие решили пропустить и почему.

Инфраструктурные модули промышленного уровня

Теперь вы знаете, какие задачи необходимо выполнить для каждого элемента инфраструктуры. Поговорим о рекомендуемых подходах к построению универсальных модулей для реализации этих задач. Мы рассмотрим такие темы.

- Мелкие модули.
- Компонуемые модули.
- Тестируемые модули.
- Модули, готовые к выпуску.
- Модули вне Terraform.

Небольшие модули

Разработчики, которые только знакомятся с Terraform и IaC в целом, часто описывают всю свою инфраструктуру для всех окружений (Dev, Stage, Prod и т. д.) в едином файле или модуле. Как уже обсуждалось в разделе «Изоляция файлов состояния» на с. 112, это плохая идея. Я на этом не останавливаюсь и утверждаю следующее: большие модули, которые содержат более нескольких сотен строчек кода или развертывают больше

нескольких тесно связанных между собой элементов инфраструктуры, должны считаться вредными.

Вот лишь некоторые из их недостатков.

•*Большие модули медленны*. Если вся ваша инфраструктура описана в одном модуле Terraform, выполнение любой команды будет занимать много времени. Мне встречались модули такого размера, что на выполнение команды terraformplan уходило 5–6 минут!

• *Большие модули небезопасны*. Если вся ваша инфраструктура описана в одном большом модуле, любое изменение потребует доступа ко всему коду. Это означает, что почти любой пользователь должен иметь права администратора, что противоречит *принципу минимальных привилегий*.

• *Большие модули несут в себе риски*. Если сложить все яйца в одну корзину, ошибка в любом месте может сломать все на свете. Например, при внесении небольшого изменения в клиентское приложение вы можете допустить опечатку или запустить не ту команду, в результате чего будет удалена ваша промышленная база данных.

• *Большие модули сложно понять*. Чем больше кода вы размещаете в одном месте, тем сложнее одному человеку его постичь целиком. А без понимания инфраструктуры, с которой вы работаете, можно допустить большую ошибку.

• *Большие модули трудно разбирать*. Разбор модуля, состоящего из нескольких десятков строк кода, не составляет труда. Разбор модуля, который состоит из нескольких тысяч строк кода, практически невозможен. Более того, это не только замедляет работу команды terraformplan, но и делает ее вывод настолько огромным, что никому не захочется его читать. А это означает, что никто не заметит ту небольшую красную строчку, которая предупреждает об удалении вашей базы данных.

•*Большие модули сложно тестировать*. Тестировать инфраструктурный код трудно, а если его очень много — то практически невозможно. Мы вернемся к этому в главе 7.

Если подытожить, ваш код должен состоять из небольших модулей, каждый из которых делает что-то одно. Это вовсе не новая или спорная идея. Вы много раз об этом слышали, только немного в другом контексте, как, например, в книге «Чистый код»51:

Первое правило функций: они должны быть компактными. Второе правило функций: они должны быть еще компактнее.

Роберт Мартин

Представьте, что вы используете язык программирования общего назначения, такой как Java, Python или Ruby, и вам попалась одна огромная функция длиной 20 000 строк:

```
def huge_function(data_set)
  x_pca = PCA(n_components=2).fit_transform(X_train)
  clusters = clf.fit_predict(X_train)
  ax = plt.subplots(1, 2, figsize=(4))
  ay = plt.subplots(0, 2, figsize=(2))
  fig = plt.subplots(3, 4, figsize=(5))
  fig.subplots_adjust(top=0.85)
  predicted = svc_model.predict(X_test)
  images_and_predictions = list(zip(images_test, predicted))
  for x in 0..xlimit
    ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
    ax[0].set_title('Predicted Training Labels')
    ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
    ax[1].set_title('Actual Training Labels')
    ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
  end
  for y in 0..ylimit
    ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
    ay[0].set_title('Predicted Training Labels')
    ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
    ay[1].set_title('Actual Training Labels')
    ay[2].scatter(X_pca[y], X_pca[1], c=clusters)
  end
  #
  # ... еще 20 тысяч строчек...
  #
end
```

Вы сразу же понимаете, что этот код странный и его лучше разбить на ряд небольших автономных функций, каждая из которых делает что-то одно:

```
def calculate_images_and_predictions(images_test, predicted)
  x_pca = PCA(n_components=2).fit_transform(X_train)
  clusters = clf.fit_predict(X_train)
  ax = plt.subplots(1, 2, figsize=(4))
  fig = plt.subplots(3, 4, figsize=(5))
  fig.subplots_adjust(top=0.85)
  predicted = svc_model.predict(X_test)
  return list(zip(images_test, predicted))
end
def process_x_coords(ax)
  for x in 0..xlimit
```

```
    ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
    ax[0].set_title('Predicted Training Labels')
    ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
    ax[1].set_title('Actual Training Labels')
    ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
  end
  return ax
end
def process_y_coords(ax)
  for y in 0..ylimit
    ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
    ay[0].set_title('Predicted Training Labels')
    ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
    ay[1].set_title('Actual Training Labels')
    ay[2].scatter(X_pca[y], X_pca[1], c=clusters)
  end
  return ay
end
#
# ... множество других мелких функций...
#
```

Ту же стратегию нужно применять и к Terraform. Представьте, что вы имеете дело с архитектурой, изображенной на рис. 6.1.

Если эта архитектура описана в едином огромном модуле Terraform длиной 20 000 строк, вы должны сразу же почувствовать, что с этим кодом что-то не так. Лучше всего разбить его на ряд небольших автономных модулей, каждый из которых выполняет одну задачу (рис. 6.2).

Модуль webserver-cluster, над которым вы работаете, начинает разрастаться. К тому же он отвечает сразу за три малосвязанные между собой задачи.

• *Группа автомасштабирования (ASG)*. Модуль webserver-cluster развертывает группу ASG, которая умеет выполнять скользящие обновления с нулевым временем простоя.

• *Балансировщик нагрузки (ALB)*. Модуль webserver-cluster развертывает ALB.

• *Демонстрационное приложение*. Модуль webserver-cluster также развертывает простое демонстрационное приложение.

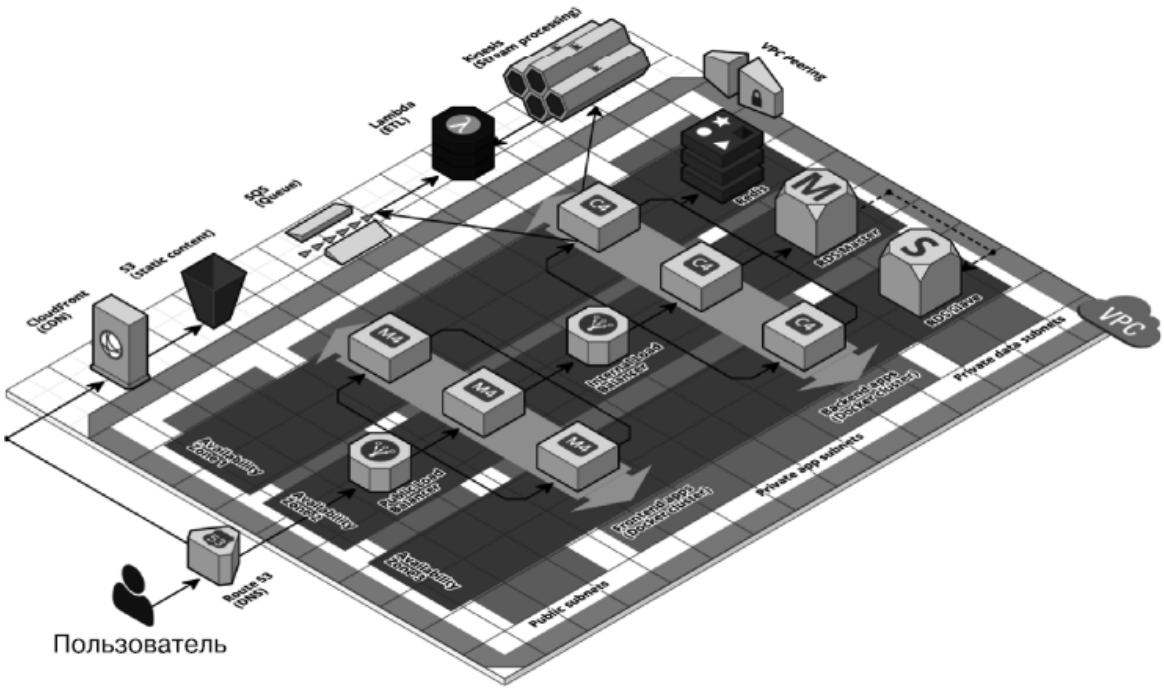


Рис. 6.1. Относительно сложная архитектура AWS

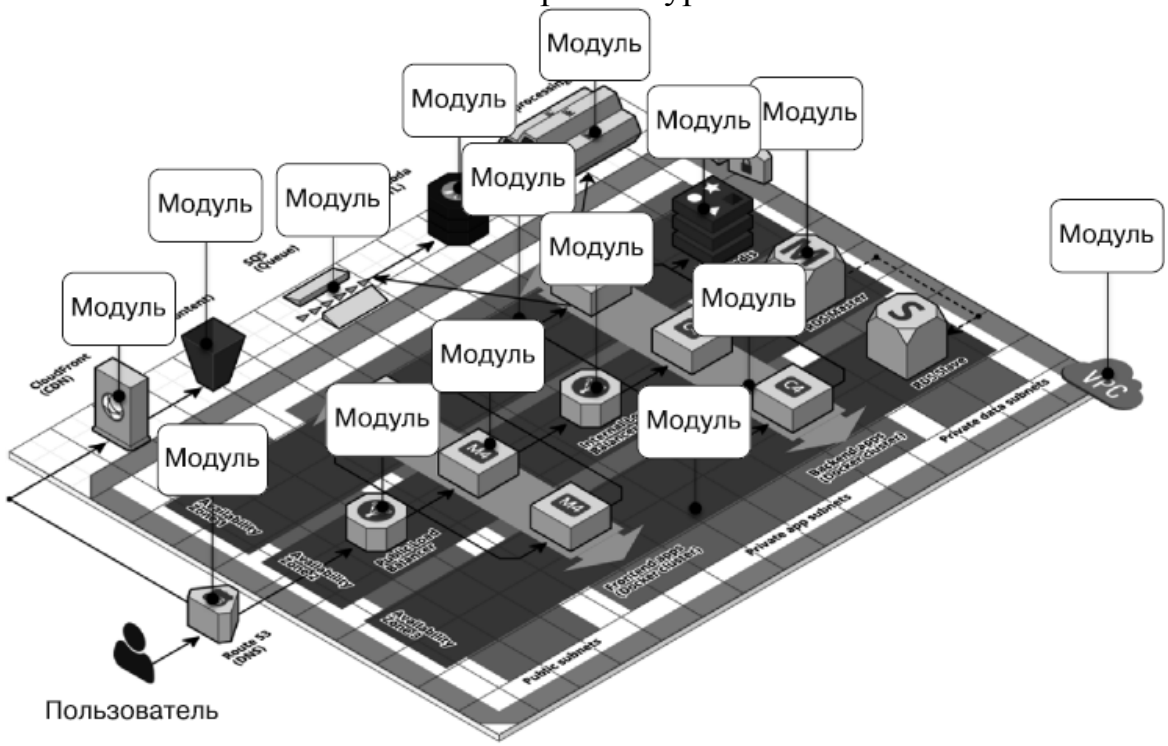


Рис. 6.2. Относительно сложная архитектура AWS, разбитая на множество мелких модулей

Разделим этот код на три небольших модуля.

• modules/cluster/asg-rolling-deploy. Обобщенный универсальный автономный модуль для развертывания группы ASG, которая умеет

выполнять скользящие обновления с нулевым временем простоя.

• modules/networking/alb. Обобщенный универсальный автономный модуль для развертывания ALB.

• modules/services/hello-world-app. Модуль для развертывания демонстрационного приложения.

Прежде чем начинать, не забудьте выполнить команду terraformdestroy для удаления всех копий webserver-cluster, которые могли остаться с предыдущих глав. После этого можете приступать к написанию модулей asg-rolling-deploy и alb. Создайте новую папку modules/cluster/asg-rolling-deploy и переместите (скопируйте и вставьте) следующие ресурсы из файла module/services/webserver-cluster/main.tf в modules/cluster/asg-rolling-deploy/main.tf:

•aws_launch_configuration;

• aws_autoscaling_group;

• aws_autoscaling_schedule (оба экземпляра);

• aws_security_group (для серверов, но не для ALB);

• aws_security_group_rule (оба правила для серверов, но не те, что для ALB);

•aws_cloudwatch_metric_alarm (оба экземпляра).

Далее переместите следующие переменные из файла module/services/webserver-cluster/variables.tf в modules/cluster/asg-rolling-deploy/variables.tf:

•cluster_name;

• ami;

• instance_type;

• min_size;

• max_size;

• enable_autoscaling;

• custom_tags;

•server_port.

Теперь перейдем к модулю ALB. Создайте новую папку modules/networking/alb и переместите следующие ресурсы из файла module/services/webserver-cluster/main.tf в modules/networking/alb/main.tf:

•aws_lb;

• aws_lb_listener;

• aws_security_group (тот, что для ALB, но не те, что для серверов);

•aws_security_group_rule (оба правила для ALB, но не те, что для серверов).

Создайте файл modules/networking/alb/variables.tf и объявите в нем одну переменную:

```
variable "alb_name" {
  description = "The name to use for this ALB"
  type        = string
}
```

Используйте эту переменную в качестве аргумента name для ресурса aws_lb:

```
resource "aws_lb" "example" {
  name               = var.alb_name
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
  security_groups    = [aws_security_group.alb.id]
}
```

и аргумента name для ресурса aws_security_group:

```
resource "aws_security_group" "alb" {
  name = var.alb_name
}
```

Мы перетасовали много кода, поэтому можете воспользоваться примерами для данной главы на странице github.com/brikis98/terraform-up-and-running-code.

Компонуемые модули

Теперь у вас есть два небольших модуля, asg-rolling-deploy и alb, каждый из которых хорошо делает что-то одно. Как их объединить? Как создавать компонуемые модули, пригодные к повторному использованию? Этот вопрос актуален не только для Terraform — программисты размышляют о нем на протяжении десятилетий. Процитирую Дуга Макилроя52, создателя Unix-каналов и ряда других инструментов для Unix, включая diff, sort, join и tr:

Это философия Unix: пишите программы, которые делают что-то одно, и делают это хорошо. Пишите программы, которые могут работать вместе.

Дуг Макилрой

Один из способов этого добиться — использовать *композицию функций.* Это когда вы можете взять вывод одной функции и передать его в качестве ввода другой. Представьте, что у вас есть следующие небольшие функции на Ruby:

```
# Простая функция для сложения
def add(x, y)
  return x + y
end
# Простая функция для вычитания
```

```
def sub(x, y)
  return x - y
end
# Простая функция для умножения
def multiply(x, y)
  return x * y
end
```

С помощью композиции функций вы можете их скомпоновать, передав вывод add и sub на вход multiply:

```
# Сложная функция, объединяющая несколько более простых
def do_calculation(x, y)
  return multiply(add(x, y), sub(x, y))
end
```

Одним из основных способов сделать функции компонуемыми является минимизация *побочных эффектов*. Для этого по возможности следует избегать чтения состояния извне, а вместо этого передавать его через входные параметры. Кроме того, вместо записи состояния вовне лучше возвращать результаты своих вычислений через выходные параметры. Минимизация побочных эффектов — это один из основных принципов функционального программирования, который упрощает понимание, тестирование и повторное использование кода. Последнее особенно важно, так как благодаря композиции сложная функция может представлять собой комбинацию более простых.

И хотя избежать побочных эффектов при работе с инфраструктурным кодом нельзя, вы все равно можете следовать тем же основным правилам в своем коде Terraform: передавайте все в виде входных переменных, возвращайте все через выходные и создавайте сложные модули на основе более простых.

Откройте файл modules/cluster/asg-rolling-deploy/variables.tf и добавьте четыре новые входные переменные:

```
variable "subnet_ids" {
  description = "The subnet IDs to deploy to"
  type        = list(string)
}
variable "target_group_arns" {
    description = "The ARNs of ELB target groups in which to register Instances"
  type        = list(string)
  default     = []
}
```

```
variable "health_check_type" {
  description = "The type of health check to perform. Must be one of: EC2, ELB."
  type        = string
  default     = "EC2"
}
variable "user_data" {
  description = "The User Data script to run in each Instance at boot"
  type        = string
  default     = ""
}
```

Первая переменная, subnet_ids, говорит модулю asg-rolling-deploy о том, в каких подсетях нужно развертываться. Если в модуле webserver-cluster подсети и VPC по умолчанию были прописаны прямо в коде, то этот модуль можно использовать в любых VPC и подсетях благодаря переменной subnet_ids, которая доступна извне. Следующие две переменные, target_group_arns и health_check_type, определяют то, как ASG интегрируется с балансировщиком нагрузки. Если у модуля webserver-cluster был встроенный экземпляр ALB, asg-rolling-deploy задумывался как универсальный модуль, поэтому выбор параметров балансировщика нагрузки с помощью входных переменных позволяет применять ASG в широком спектре сценариев: например, без ALB, с одним балансировщиком, с несколькими NLB и т. д.

Возьмите эти три входные переменные и передайте их ресурсу aws_autoscaling_group в файле modules/cluster/asg-rolling-deploy/main.tf, заменив вручную прописанные параметры, которые ссылались на ресурсы (скажем, ALB) и источники данных (вроде aws_subnet_ids) и не были скопированы в наш модуль asg-rolling-deploy:

```
resource "aws_autoscaling_group" "example" {
  # Напрямую зависит от имени конфигурации запуска, поэтому вместе
  # с ним всегда нужно заменять и эту группу ASG
                                                    name                 =
"${var.cluster_name}-${aws_launch_configuration.example.name}"
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = var.subnet_ids
  # Настраиваем интеграцию с балансировщиком нагрузки
  target_group_arns = var.target_group_arns
  health_check_type = var.health_check_type
  min_size = var.min_size
  max_size = var.max_size
```

```
  # Ждем, пока как минимум столько серверов не пройдут проверку
    # работоспособности, прежде чем считать развертывание ASG завершенным
  min_elb_capacity = var.min_size
  # (...)
}
```

Четвертую переменную, user_data, нужно передать в скрипт пользовательских данных. Если в модуле webserver-cluster этот скрипт был прописан вручную и мог использоваться лишь для развертывания приложения Hello, World, модуль asg-rolling-deploy позволяет развертывать в ASG любое приложение, так как скрипт теперь передается в виде входной переменной. Итак, передайте переменную user_data ресурсу aws_launch_configuration (заменив ею ссылку на источник данных template_file, которую мы не скопировали в модуль asg-rolling-deploy):

```
resource "aws_launch_configuration" "example" {
  image_id        = var.ami
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data       = var.user_data
  # Требуется при использовании группы автомасштабирования
  # в конфигурации запуска.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

Следует также добавить парочку полезных переменных в файл modules/cluster/asgrolling-deploy/outputs.tf:

```
output "asg_name" {
  value       = aws_autoscaling_group.example.name
  description = "The name of the Auto Scaling Group"
}
output "instance_security_group_id" {
  value       = aws_security_group.instance.id
  description = "The ID of the EC2 Instance Security Group"
}
```

Вывод этих данных делает модуль asg-rolling-deploy еще более универсальным, поскольку с помощью этих выходных значений его пользователи смогут изменять его поведение, подключая собственные правила к группе безопасности.

По аналогичным причинам несколько выходных переменных следует добавить и в файл modules/networking/alb/outputs.tf:

```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "The domain name of the load balancer"
}
output "alb_http_listener_arn" {
  value       = aws_lb_listener.http.arn
  description = "The ARN of the HTTP listener"
}
output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "The ALB Security Group ID"
}
```

Вы скоро увидите, как их использовать.

Последним шагом будет преобразование webserver-cluster в модуль hello-world-app, способный развернуть приложение Hello, World с помощью asg-rolling-deploy и alb. Для этого переименуйте module/services/webserver-cluster в module/services/hello-world-app. После всех предыдущих изменений в файле module/services/hello-world-app/main.tf должны остаться только следующие ресурсы:

- template_file (для пользовательских данных);
- aws_lb_target_group;
- aws_lb_listener_rule;
- terraform_remote_state (для БД);
- aws_vpc;
- aws_subnet_ids.

Добавьте следующую переменную в файл modules/services/hello-world-app/variables.tf:

```
variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
}
```

Теперь добавьте созданный вами ранее модуль asg-rolling-deploy в hello-world-app, чтобы развернуть ASG:

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"
  cluster_name  = "hello-world-${var.environment}"
  ami           = var.ami
  user_data     = data.template_file.user_data.rendered
```

```
  instance_type = var.instance_type
  min_size          = var.min_size
  max_size          = var.max_size
  enable_autoscaling = var.enable_autoscaling
  subnet_ids        = data.aws_subnet_ids.default.ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"
  custom_tags = var.custom_tags
}
```

Добавьте в hello-world-app еще и модуль alb, который вы тоже создали ранее, чтобы развернуть ALB:

```
module "alb" {
  source     = "../../networking/alb"
  alb_name   = "hello-world-${var.environment}"
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

Обратите внимание на то, что входная переменная environment используется для соблюдения соглашения об именовании, чтобы все ваши ресурсы были распределены по пространствам имен в зависимости от среды (например, hello-world-stage, hello-world-prod). Этот код также устанавливает соответствующие значения для переменных subnet_ids, target_group_arns, health_check_type и user_data, которые вы добавили ранее.

Теперь вам нужно настроить целевую группу ALB и правило прослушивателя для этого приложения. Сделайте так, чтобы ресурс aws_lb_target_group в файле modules/services/hello-world-app/main.tf использовал в своем поле name переменную environment:

```
resource "aws_lb_target_group" "asg" {
  name     = "hello-world-${var.environment}"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = data.aws_vpc.default.id
  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
```

```
}
```

Теперь сделайте так, чтобы параметр listener_arn ресурса aws_lb_listener_rule указывал на вывод alb_http_listener_arn модуля alb:

```
resource "aws_lb_listener_rule" "asg" {
  listener_arn = module.alb.alb_http_listener_arn
  priority     = 100
  condition {
    field  = "path-pattern"
    values = ["*"]
  }
  action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}
```

И передайте важные выходные параметры из asg-rolling-deploy и alb в качестве вывода модуля hello-world-app:

```
output "alb_dns_name" {
  value       = module.alb.alb_dns_name
  description = "The domain name of the load balancer"
}
output "asg_name" {
  value       = module.asg.asg_name
  description = "The name of the Auto Scaling Group"
}
output "instance_security_group_id" {
  value       = module.asg.instance_security_group_id
  description = "The ID of the EC2 Instance Security Group"
}
```

Это композиция функций в действии: вы выстраиваете сложное поведение (приложение Hello, World) из более простых частей (модули ASG и ALB). В Terraform часто можно встретить как минимум два типа модулей.

•*Обобщенные модули*. Такие модули, как asg-rolling-deploy и alb, являются базовыми составными блоками вашего кода, которые можно использовать во множестве разных сценариев. Вы видели, как с их помощью развертывается приложение Hello, World, но те же модули можно применить, к примеру, чтобы развернуть ASG для запуска кластера Kafka или полностью автономного экземпляра ALB, способного распределять нагрузку между разнообразными приложениями. Использовать один

балансировщик для всех приложений сразу будет дешевле, чем выделять для каждого из них по одному ALB.

•*Узкоспециализированные модули*. Модули вроде hello-world-app сочетают в себе несколько обобщенных модулей с расчетом на один конкретный сценарий использования, скажем, развертывание приложения Hello, World.

В реальной работе вам, возможно, придется дробить свои модули еще сильнее, чтобы улучшить композицию и повторное применение. Например, вы можете применять коллекцию открытых, универсальных модулей из репозитория terraform-aws-consul (http://bit.ly/33hhOwr) для работы с HashiCorp Consul. Consul — это открытое, распределенное хранилище типа «ключ — значение», которому иногда нужно прослушивать сетевые запросы на множестве разных портов (серверные RPC, CLI RPC, Serf WAN, HTTP API, DNS и т. д.), поэтому оно обычно требует около 20 групп безопасности. В репозитории terraform-aws-consul правила этих групп описываются в отдельном модуле consul-security-group-rules (http://bit.ly/31oo5oc).

Чтобы понять, зачем это нужно, можно сначала посмотреть на то, как развертывается система Consul. Ее часто развертывают в качестве хранилища данных для HashiCorp Vault, открытого распределенного хранилища конфиденциальной информации, в котором можно безопасно размещать пароли, API-ключи, сертификаты TLS и т. д. В репозитории terraform-aws-vault (http://bit.ly/2MM6AKn) можно найти коллекцию открытых универсальных модулей для работы с Vault. Там же представлена диаграмма, на которой изображена типичная промышленная архитектура для этой системы (рис. 6.3).

В промышленных условиях Vault обычно запускается в одной группе ASG, состоящей из 3–5 серверов (развернутых с помощью модуля vault-cluster (http://bit.ly/2TngRON), а Consul — в другой, тоже с 3–5 серверами (развернутыми с модулем consul-cluster (http://bit.ly/2YQXIWe), поэтому каждую из этих систем можно масштабировать и защищать по отдельности. Однако в тестовом окружении запуск такого количества серверов будет излишним, поэтому, чтобы сэкономить, Vault и Consul лучше запускать в одной группе ASG, состоящей, скажем, из одного сервера и развернутой с использованием модуля vault-cluster. Если бы все правила группы безопасности Consul были описаны в модуле consul-cluster, вы бы не смогли применять их повторно (разве что вы вручную скопируете 20 с лишним правил) при развертывании Consul с модулем vault-cluster. Но, поскольку правила описаны в отдельном модуле, consul-security-group-rules, вы можете подключить их к vault-cluster или к кластеру почти любого другого типа.

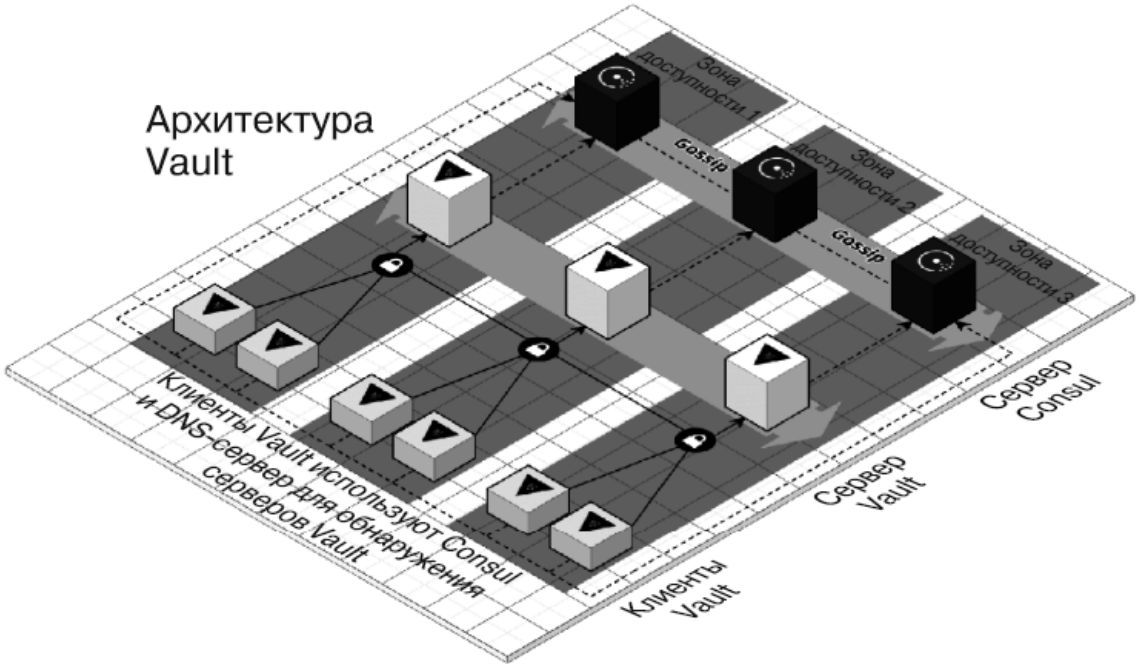


Рис. 6.3. Архитектура HashiCorp Vault и Consul

Для простого приложения вроде Hello, World такого рода разбиение будет чрезмерным, но, если вы имеете дело с настоящей сложной инфраструктурой, выделение правил группы безопасности, политик IAM и других сквозных элементов в отдельные автономные модули — часто необходимый шаг для поддержки разных методик развертывания. Мы их использовали не только для Consul и Vault, но и для стека ELK (в промышленной среде мы запускали Elasticsearch, Logstash и Kibana в отдельных кластерах, а в тестовой — в одном и том же), Confluent Platform (в промышленной среде мы запускали Kafka, ZooKeeper, REST Proxy и Schema Registry в отдельных кластерах, а в тестовой — в одном и том же), стека TICK (в промышленной среде мы запускали Telegraf, InfluxDB, Chronograf и Kapacitor в отдельных кластерах, а в тестовой — в одном и том же) и т. д.

Тестируемые модули

Вы уже написали довольно много кода в виде трех модулей: asg-rolling-deploy, alb и hello-world-app. Теперь нужно убедиться, что этот код и правда работает.

Указанные модули не корневые и не предназначены для развертывания напрямую. Чтобы их развернуть, нужно написать какой-то код Terraform, подключить нужные вам аргументы, сконфигурировать поля provider и backend и т. д. Для этого можно создать папку examples, которая будет содержать примеры использования ваших модулей. Попробуем это сделать.

Создайте файл examples/asg/main.tf следующего содержания:

```
provider "aws" {
  region = "us-east-2"
}
module "asg" {
  source = "../../modules/cluster/asg-rolling-deploy"
  cluster_name  = var.cluster_name
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  min_size          = 1
  max_size          = 1
  enable_autoscaling = false
  subnet_ids = data.aws_subnet_ids.default.ids
}
data "aws_vpc" "default" {
  default = true
}
data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}
```

Этот фрагмент кода развертывает группу ASG с одним сервером, применяя модуль asg-rolling-deploy. Попробуйте выполнить команды terraforminit и terraformapply и убедитесь, что все работает без ошибок и ASG действительно создается. Теперь добавьте файл README.md с данными инструкциями, после чего этот крошечный пример станет куда более мощным. С помощью всего нескольких файлов и строк кода вы получаете следующее.

•*Механизм ручных тестов*. Этот демонстрационный код можно использовать при работе над модулем asg-rolling-deploy, развертывая и удаляя его по много раз путем ручного выполнения команд terraformapply и terraformdestroy. Это позволит убедиться в его предсказуемом поведении.

• *Механизм автоматических тестов*. Как вы увидите в главе 7, с помощью этого демонстрационного кода можно также создавать автоматические тесты для ваших модулей. Советую размещать тесты в папке test.

• *Исполняемая документация*. Если сохранить этот пример (вместе с файлом README.md) в системе управления версиями, каждый член вашей команды сможет его найти и понять с его помощью, как работает ваш модуль. Это также позволит запускать модуль без написания какого-либо кода. Таким образом, вы предоставляете своей команде обучающий

материал и вместе с тем можете подтвердить его корректность автоматическими тестами.

У каждого модуля Terraform, который находится в папке modules, должна быть папка examples с соответствующим примером. А у последнего в этой папке должен быть подходящий тест в папке test. У каждого модуля, скорее всего, будет ряд примеров (и, следовательно, несколько тестов), которые иллюстрируют разные конфигурации и сценарии использования. Скажем, для модуля asg-rolling-deploy можно придумать другие примеры, чтобы показать, как он работает с правилами автомасштабирования, как к нему подключать балансировщики нагрузки, как ему назначить пользовательские теги и т. д.

Если все это объединить, структура каталогов типичного репозитория modules будет выглядеть так:

```
modules
└ examples
  └ alb
  └ asg-rolling-deploy
    └ one-instance
    └ auto-scaling
    └ with-load-balancer
    └ custom-tags
  └ hello-world-app
  └ mysql
└ modules
  └ alb
  └ asg-rolling-deploy
  └ hello-world-app
  └ mysql
└ test
  └ alb
  └ asg-rolling-deploy
  └ hello-world-app
  └ mysql
```

В качестве упражнения предлагаю вам превратить код RDS, который вы написали, в модуль MySQL и добавить к модулям alb, asg-rollingdeploy, mysql и hello-world-app множество разных примеров.

Разработку нового модуля крайне полезно *начинать* с написания демонстрационного кода. Если сразу заняться реализацией, можно легко погрязнуть в деталях, и к моменту, когда вы вернетесь к API, ваш модуль будет малопонятным и сложным в применении. Если же вы начнете с кода

примера, у вас будет возможность подумать об опыте использования, выработать для своего модуля аккуратный API и затем вернуться к реализации. Поскольку код примера и так служит основным способом тестирования модуля, такой подход является разновидностью *разработки через тестирование* (Test-Driven Development, TDD); подробнее о тестировании мы поговорим в главе 7.

Существует еще один подход, который вы сможете оценить по достоинству, как только начнете регулярно тестировать свои модули: *закрепление версий*. Вы должны закрепить все свои модули за определенной версией Terraform, используя аргумент required_version. Как минимум следует требовать конкретную мажорную версию Terraform:

```
terraform {
  # Требуем любую версию Terraform вида 0.12.x
  required_version = ">= 0.12, < 0.13"
}
```

Этот код позволит использовать для вашего модуля только версии Terraform вида 0.12.x, но не 0.11.x или 0.13.x. Это очень важно, потому что при каждом мажорном выпуске Terraform теряет обратную совместимость: например, переход с 0.11.x на 0.12.x требует существенного изменения кода, поэтому вам вряд ли понравится, если такой переход произойдет случайно. Если добавить параметр required_version и попытаться выполнить terraformapply, используя другую версию, сразу же появится ошибка:

```
$ terraform apply
Error: Unsupported Terraform Core version
This configuration does not support Terraform version 0.11.11. To proceed,
either choose another supported Terraform version or update the root module's
version constraint. Version constraints are normally set for good reason, so
updating the constraint may lead to other errors or unexpected behavior.
```

Для кода промышленного уровня рекомендую еще строже закреплять версию:

```
terraform {
  # Требуем исключительно версию Terraform 0.12.0
  required_version = "= 0.12.0"
}
```

Это вызвано тем, что даже переход на новые «заплаточные» версии (скажем, 0.12.0 —> 0.12.1) может создать проблемы. Иногда они содержат ошибки, а иногда у них ломается обратная совместимость (хотя в наши дни это случается реже). Но еще более серьезной проблемой является то, что файл состояния, измененный более новой версией Terraform, уже нельзя использовать в старых версиях. Представьте, к примеру, что весь ваш код

развернут с помощью Terraform 0.12.0, и тут вдруг разработчик, у которого установлена версия 0.12.1, применяет к некоторым из ваших модулей команду terraformapply. В результате файлы состояния этих модулей больше не совместимы с 0.12.0, поэтому вы вынуждены обновить все свои компьютеры для разработки и CI-серверы на 0.12.1!

Ситуация, скорее всего, улучшится, когда Terraform достигнет версии 1.0.0 и начнет соблюдать обратную совместимость, но, пока этого не произошло, я советую привязываться к *определенной* версии этой системы. Это позволит избежать незапланированных обновлений. Вы сможете обновляться тогда, когда будете готовы, и делать это сразу для всех своих компьютеров и CI-серверов.

Рекомендую также закреплять версии всех ваших провайдеров:

```
provider "aws" {
  region = "us-east-2"
  # Разрешаем использовать любую версию провайдера AWS вида 2.x
  version = "~> 2.0"
}
```

Этот пример привязывает код провайдера AWS к версиям вида 2.x (синтаксис ~>2.0 эквивалентен >=2.0,<3.0). И снова, чтобы избежать случайного внесения обратно несовместимых изменений, необходимо закрепить мажорную версию. Закрепление конкретной версии зависит от провайдера. Предположим, провайдер AWS часто обновляется и имеет хорошую поддержку обратной совместимости, поэтому будет разумно закрепить только его мажорную версию и позволить автоматическое обновление заплаток, чтобы получить удобный доступ к новым возможностям. Однако все провайдеры разные, поэтому обращайте внимание на то, насколько хорошо они справляются с поддержкой обратной совместимости, и закрепляйте номера версий соответствующим образом.

Модули, готовые к повторному использованию

Следующим шагом после написания и тестирования модулей будет их выпуск. Как вы уже видели в разделе «Управление версиями» на с. 53, для этого подходят теги Git в сочетании с семантическим версионированием:

```
$ git tag -a "v0.0.5" -m "Create new hello-world-app module"
$ git push --follow-tags
```

Например, чтобы развернуть в тестовом окружении версию v0.0.5 вашего модуля hello-world-app, в файле live/stage/services/hello-world-app/main.tf следует прописать такой код:

```
provider "aws" {
  region = "us-east-2"
```

```
  # Разрешаем использовать любую версию провайдера AWS вида 2.x
  version = "~> 2.0"
}
module "hello_world_app" {
  # TODO: подставить сюда URL и версию вашего собственного модуля!!
  source = "git@github.com:foo/modules.git//services/hello-world-app?ref=v0.0.5"
  server_text = "New server text"
  environment = "stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key = "stage/data-stores/mysql/terraform.tfstate"
  instance_type = "t2.micro"
  min_size = 2
  max_size = 2
  enable_autoscaling = false
}
```

Далее укажем доменное имя ALB в виде выходного параметра в файле live/stage/services/hello-world-app/outputs.tf:

```
output "alb_dns_name" {
  value = module.hello_world_app.alb_dns_name
  description = "The domain name of the load balancer"
}
```

Теперь вы можете развернуть свой модуль с поддержкой версионирования, выполнив команды terraforminit и terraformapply:

```
$ terraform apply
(...)
Apply complete! Resources: 13 added, 0 changed, 0 destroyed.
Outputs:
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

Если все работает хорошо, можете развернуть эту же версию (и, следовательно, этот же код) в других средах, включая промышленную. Если у вас когда-нибудь возникнет проблема, вы сможете откатиться назад, развернув более старую версию.

Еще одним способом выпуска модулей является их публикация в реестре Terraform. Публичный реестр модулей Terraform находится по адресу registry.terraform.io и содержит сотни универсальных, открытых и поддерживаемых сообществом модулей для AWS, Google Cloud, Azure и

многих других провайдеров. Для публикации в нем своих модулей необходимо соблюсти несколько требований53.

• Модуль должен находиться в репозитории GitHub.

• Репозиторий должен иметь название вида terraform-<PROVIDER>-<NAME>, где PROVIDER — это провайдер, на которого рассчитан модуль (например, aws), а NAME — имя модуля (скажем, vault).

• Модулю нужна определенная структура файлов: код Terraform должен находиться в корне репозитория, у вас должен быть файл README.md и необходимо соблюдать соглашение об именовании файлов main.tf, variables.tf и outputs.tf.

• Для выпуска кода в репозитории нужно использовать теги Git и семантическое версионирование (x.y.z).

Если ваш модуль отвечает всем этим требованиям, вы можете сделать его доступным для всех желающих. Для этого войдите в реестр модулей Terraform с помощью своей учетной записи GitHub и опубликуйте свой модуль в веб-интерфейсе (рис. 6.4).

Реестр модулей Terraform умеет анализировать входные и выходные параметры модуля, поэтому вы можете видеть их в пользовательском интерфейсе вместе с полями type и description, как показано на рис. 6.5.

Более того, Terraform поддерживает специальный синтаксис для подключения модулей из реестра. Вместо длинного адреса Git с малозаметными параметрами ref вы можете использовать в аргументе source более короткий URL-адрес реестра, указывая версию с помощью отдельного аргумента version. Вот как это выглядит:

```
module "<NAME>" {
  source  = "<OWNER>/<REPO>/<PROVIDER>"
  version = "<VERSION>"
  # (...)
}
```

NAME — это идентификатор модуля в вашем коде Terraform, OWNER — владелец репозитория в GitHub (например, в github.com/foo/bar таковым является foo), REPO — название GitHub-репозитория (скажем, в github.com/foo/bar это bar), PROVIDER — нужный вам провайдер (предположим, aws), а VERSION — версия модуля, которую вы хотите использовать. Вот пример того, как подключить модуль Vault из реестра Terraform:

```
module "vault" {
  source  = "hashicorp/vault/aws"
  version = "0.12.2"
  # (...)
```

}

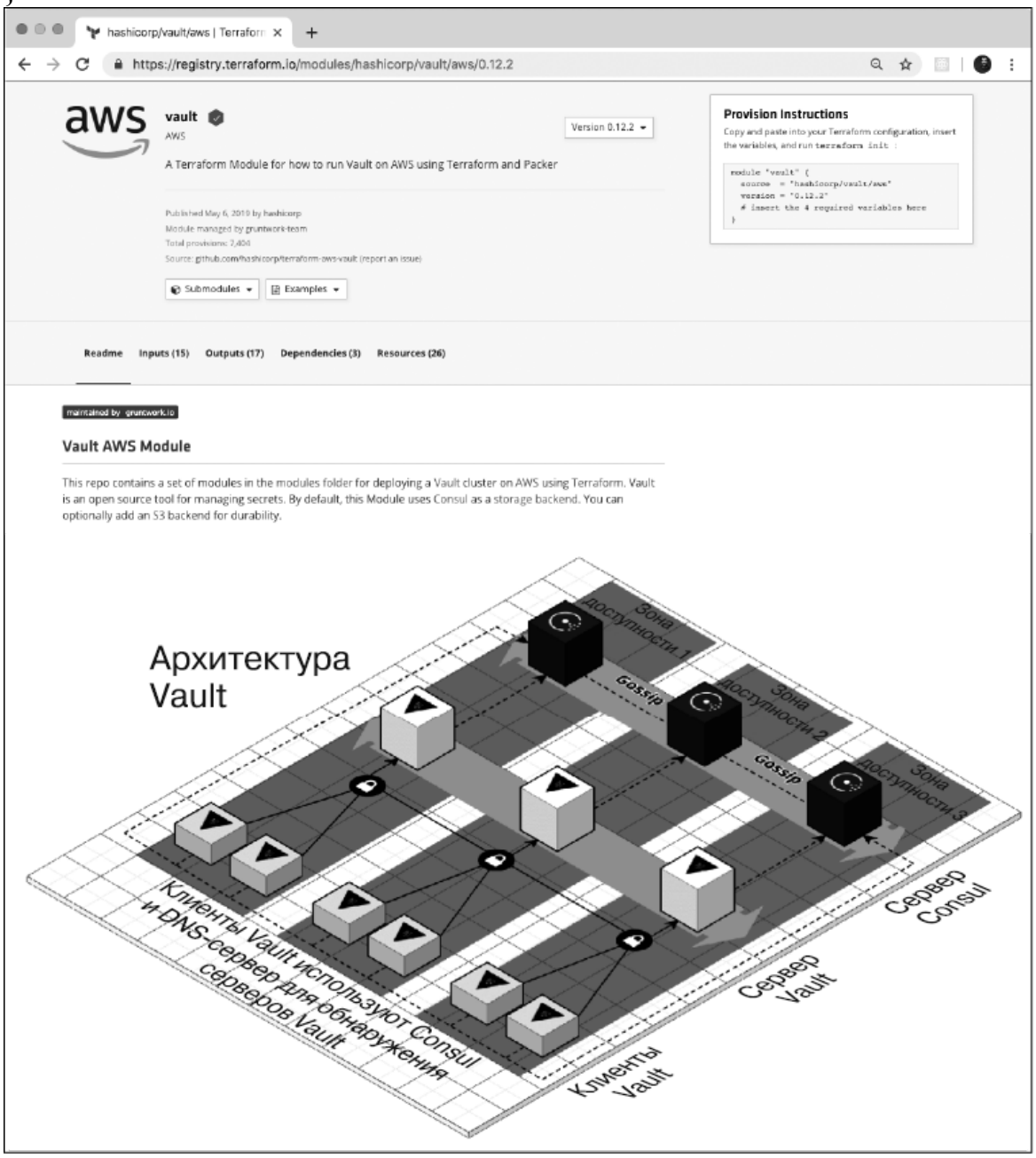


Рис. 6.4. Модуль HashiCorp Vault в реестре Terraform

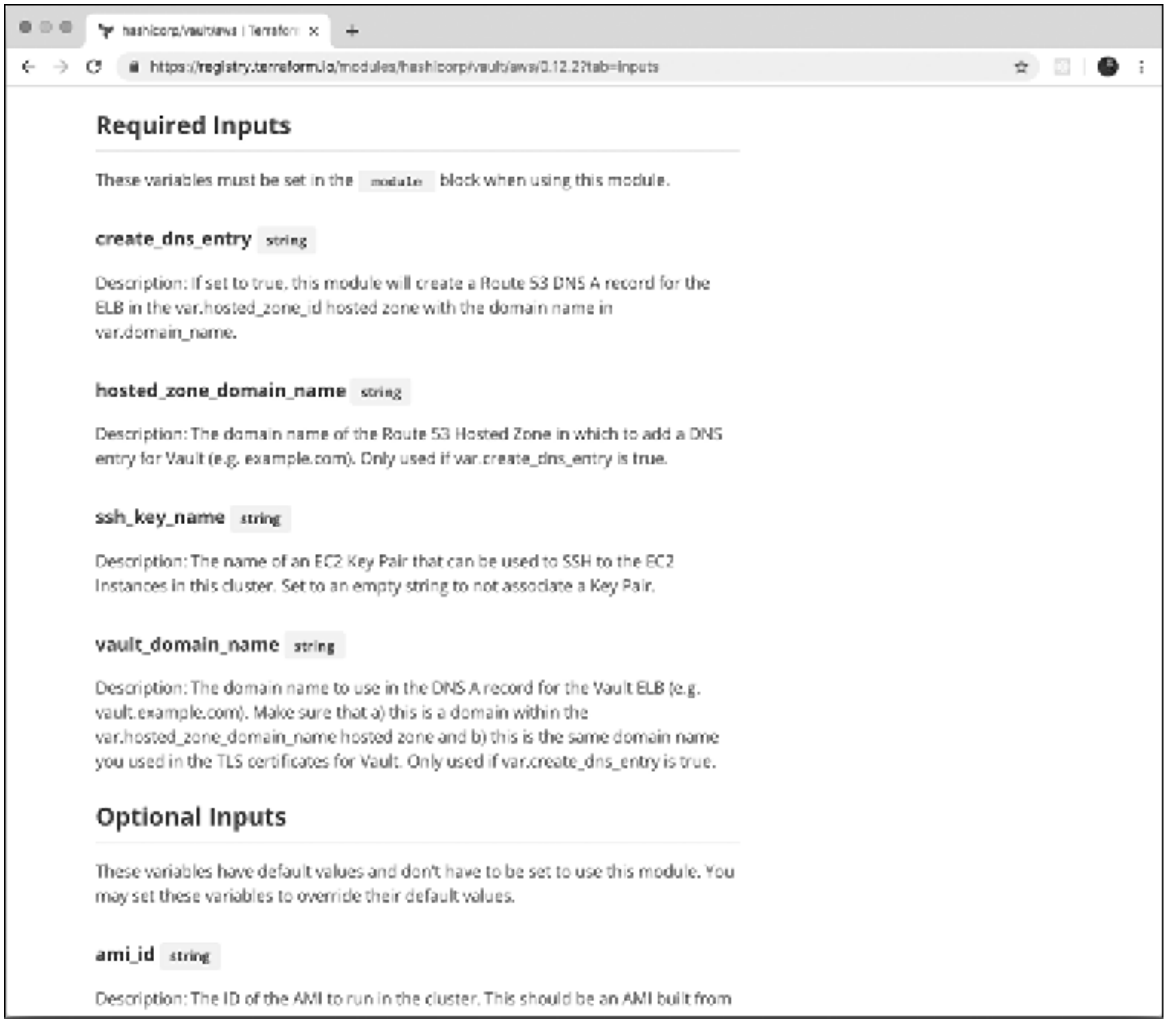


Рис. 6.5. Реестр Terraform автоматически анализирует и отображает ввод и вывод модуля

Если у вас есть подписка на сервис Terraform Enterprise от HashiCorp, вы можете делать то же самое с закрытым реестром модулей Terraform, который находится в ваших закрытых Git-репозиториях и доступен только вашей команде. Это отличный способ распространения модулей внутри вашей компании.

За пределами возможностей Terraform-модулей

Несмотря на то что эта книга посвящена Terraform, для построения собственной инфраструктуры промышленного уровня понадобятся и другие инструменты: Docker, Packer, Chef, Puppet и, конечно же, старый добрый bash-скрипт. Большую часть этого кода можно разместить в папке modules рядом с кодом Terraform. Например, в репозитории HashiCorp Vault, который вы видели ранее, папка modules содержит не только модули Terraform, как уже упомянутый выше vault-cluster, но и bash-скрипты: вроде run-vault

(http://bit.ly/2GTCxg8), который можно выполнять на сервере Linux во время загрузки (скажем, в качестве пользовательских данных) для конфигурации и запуска Vault.

Однако иногда возникает необходимость пойти еще дальше и выполнить внешний код (например, скрипт) прямо из модуля Terraform. Временами это нужно для интеграции Terraform с другой системой (скажем, вы уже применяли Terraform для выполнения скриптов пользовательских данных на серверах EC2), а изредка это попытка обойти ограничения Terraform, вызванные нехваткой какого-то API провайдера или невозможностью реализовать сложную логику в декларативном стиле. Если поискать, в Terraform можно найти несколько «аварийных люков», которые позволяют это сделать:

- средства инициализации ресурсов;
- средства инициализации ресурсов с использованием null_resource;
- внешний источник данных.

Пройдемся по ним по очереди.

Средства инициализации ресурсов

Terraform использует *средства инициализации ресурсов* во время запуска для выполнения скриптов на локальном или удаленном компьютере. Обычно это связано с компоновкой/очисткой ресурсов или управлением конфигурацией. Таких средств существует несколько, включая local-exec (скрипт в локальной системе), remote-exec (скрипт на удаленном компьютере), chef (запускает Chef Client для удаленного ресурса) и file (копирует файл на удаленный ресурс)54.

Чтобы добавить в ресурс средства инициализации, можно воспользоваться блоком provisioner. Например, ниже показано, как с помощью local-exec выполнить скрипт на локальном компьютере:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

Если выполнить для этого кода команду terraformapply, она выведет текст Hello,Worldfrom, за которым следуют подробности о локальной операционной системе (полученные из команды uname):

```
$ terraform apply
(...)
```

aws_instance.example (local-exec): Hello, World from Darwin x86_64 i386
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Для использования средства инициализации remote-exec потребуется чуть больше усилий. Чтобы выполнить код на удаленном ресурсе, таком как сервер EC2, вашему клиенту Terraform придется сделать следующее.

• *Связаться с сервером EC2 по сети*. Вы уже знаете, как разрешить это с помощью группы безопасности.

• *Пройти аутентификацию на сервере EC2*. Средство инициализации remote-exec поддерживает протоколы SSH и WinRM. Поскольку вы будете запускать сервер под управлением Linux (Ubuntu), нужно использовать аутентификацию по SSH. Значит, вам необходимо сконфигурировать SSH-ключи.

Для начала создадим группу безопасности, которая разрешает входящие соединения на стандартном для SSH порте под номером 22:

```
resource "aws_security_group" "instance" {
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"
     # Чтобы этот пример можно было легко попробовать, мы разрешаем все
    # соединения по SSH.
        # В реальных условиях вы должны принимать их только с доверенных IP-адресов.
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Обычно для аутентификации нужно сгенерировать на своем компьютере пару SSH-ключей: открытый и секретный. Первый следует загрузить в AWS, а второй сохранить в надежном месте, откуда к нему сможет обратиться Terraform. Но, чтобы вам проще было попробовать этот пример, можете использовать ресурс tls_private_key, который сгенерирует секретный ключ автоматически:

```
# Чтобы этот пример можно было легко попробовать, мы генерируем секретный ключ
# в Terraform.
# В реальных условиях управление SSH-ключами следует вынести за пределы Terraform.
resource "tls_private_key" "example" {
```

```
  algorithm = "RSA"
  rsa_bits = 4096
}
```

Этот секретный ключ хранится в состоянии Terraform, что не очень хорошо для промышленного использования, но подойдет для этого примера. Теперь загрузим открытый ключ в AWS с помощью ресурса aws_key_pair:

```
resource "aws_key_pair" "generated_key" {
  public_key = tls_private_key.example.public_key_openssh
}
```

Приступим к написанию кода для сервера EC2:

```
resource "aws_instance" "example" {
  ami                  = "ami-0c55b159cbfafe1f0"
  instance_type        = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name             = aws_key_pair.generated_key.key_name
}
```

Первые несколько строк этого кода должны показаться знакомыми: они развертывают Ubuntu Amazon Machine Image (AMI) на t2.micro и привязывают к этому серверу EC2 группу безопасности, которую вы создали ранее. Единственное нововведение — это использование атрибута key_name, который позволяет AWS привязать ваш открытый ключ к серверу EC2. AWS добавит этот ключ в его файл authorized_keys, благодаря чему вы сможете зайти на этот сервер по SSH с соответствующим секретным ключом.

Теперь добавим к серверу EC2 средство инициализации remote-exec:

```
resource "aws_instance" "example" {
  ami                  = "ami-0c55b159cbfafe1f0"
  instance_type        = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name             = aws_key_pair.generated_key.key_name
  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }
}
```

Это почти ничем не отличается от средства инициализации local-exec, только вместо одной исполняемой команды (command) здесь с помощью аргумента inline передается их список. В завершение вам нужно сконфигурировать Terraform для подключения к этому серверу EC2 по SSH при запуске remote-exec. Для этого предусмотрен блок connection:

```
resource "aws_instance" "example" {
```

```
  ami                    = "ami-0c55b159cbfafe1f0"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name
  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }
  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ubuntu"
    private_key = tls_private_key.example.private_key_pem
  }
}
```

Блок connection заставляет Terraform подключиться к публичному IP-адресу сервера EC2 по SSH с помощью имени пользователя "ubuntu" (так по умолчанию называется корневой пользователь в образах AMI с Ubuntu) и автоматически сгенерированного секретного ключа. Обратите внимание на ключевое слово self, которое применяется для задания параметра host. Это выражение со следующим синтаксисом:

self.<ATTRIBUTE>

Этот специальный синтаксис можно использовать исключительно в блоках connection и provisioner для обращения к выходному атрибуту родительского ресурса. Если применить стандартный синтаксис вида aws_instance.example.<ATTRIBUTE>, получится ошибка циклической зависимости, так как ресурс не может ссылаться на самого себя. Таким образом, выражение self является обходным решением, которое было добавлено специально для средств инициализации ресурсов.

Если выполнить для этого кода команду terraformapply, получится следующее:

```
$ terraform apply
(...)
aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Provisioning with 'remote-exec'...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
(... repeats a few more times ...)
```

aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connected!
aws_instance.example (remote-exec): Hello, World from Linux x86_64 x86_64
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Сравнение средств инициализации ресурсов и пользовательских данных

Вы познакомились с двумя разными способами выполнения скриптов на сервере с помощью Terraform: один с использованием средства инициализации remote-exec, а другой — с применением скрипта пользовательских данных. Последний мне кажется более полезным по нескольким причинам.

Средство инициализации remote-exec требует открытия доступа к вашим серверам по SSH или WinRM, что усложняет управление (как вы сами могли убедиться, поработав с группами безопасности и SSH-ключами) и ухудшает безопасность по сравнению с пользовательскими данными, которым нужен лишь доступ к API AWS (он требуется в любом случае для использования Terraform).

Вы можете применять скрипты пользовательских данных в сочетании с группами ASG. Благодаря этому все серверы в заданной группе ASG (включая те, что запускаются в результате автомасштабирования или автовосстановления) выполнят скрипт во время своей загрузки. Средства инициализации ресурсов действуют только во время работы Terraform, но не с ASG.

Скрипт пользовательских данных можно видеть в консоли EC2 (выберите сервер и щелкните Actions□Instance Settings□View/Change User Data (Действия□Настройки сервера□Просмотреть/изменить пользовательские данные)), а его журнал выполнения можно найти на самом сервере EC2 (обычно в файле /var/log/cloud-init*.log). Обе возможности полезны при отладке, и ни одна из них не доступна для средств инициализации ресурсов.

У средств инициализации ресурсов всего два преимущества.

Максимальный размер скриптов пользовательских данных равен 16 Кбайт, тогда как скрипты инициализации ресурсов могут быть произвольной длины.

Chef, Puppet и Salt автоматически устанавливают, конфигурируют и запускают на сервере соответствующие клиенты. Благодаря этому для настройки серверов проще использовать инструменты управления конфигурацией вместо специализированных скриптов.

Средство инициализации remote-exec не знает, когда именно сервер EC2 завершит загрузку и будет готов принимать соединения, поэтому оно периодически пытается подключиться по SSH, пока у него это не получится или пока не истечет время ожидания. Время ожидания по умолчанию равно пяти минутам, но вы можете его сконфигурировать. Рано или поздно соединение будет установлено и вы получите от сервера ответ Hello, World.

Следует отметить, что по умолчанию средство инициализации действует *во время создания ресурсов*. Это означает, что оно выполняется вместе с командой terraformapply и только во время начального создания ресурса. Оно не будет срабатывать при всех последующих запусках terraformapply, поэтому его основное применение — выполнение кода первоначальной инициализации. Если в средстве инициализации ресурсов указать when="destroy", оно будет действовать *во время их удаления*, то есть: а) после выполнения terraformdestroy и б) непосредственно перед удалением ресурса.

Вы можете указать несколько средств инициализации для одного и того же ресурса, и Terraform запустит их по очереди, сверху вниз. Чтобы объяснить Terraform, как обрабатывать ошибки, полученные в результате инициализации, можно использовать аргумент on_failure: если присвоить ему "continue", Terraform проигнорирует ошибку и продолжит создание/удаление ресурса; если присвоить ему "abort", Terraform прервет создание/удаление.

Средства инициализации ресурсов с использованием null_resource

Средства инициализации можно определять только внутри ресурса, но иногда при их выполнении хочется обойтись без привязки к определенному ресурсу. Это можно сделать с помощью сущности под названием null_resource, которая ведет себя как обычный ресурс Terraform, но при этом ничего не создает. Определив средство инициализации для null_resource, вы можете запустить свой скрипт в рамках жизненного цикла Terraform, не привязываясь ни к какому «настоящему» ресурсу:

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

У null_resource есть удобный аргумент под названием triggers, который принимает ассоциативный массив с ключами и значениями. При любом изменении значений ресурс null_resource будет создаваться заново, запуская тем самым все средства инициализации, которые в нем определены.

Например, если вы хотите выполнять средство инициализации внутри null_resource при каждом запуске команды terraformapply, можете воспользоваться встроенной функцией uuid(). При каждом вызове внутри аргумента triggers она возвращает новый, свежесгенерированный идентификатор UUID:

```
resource "null_resource" "example" {
  # Используйте UUID, чтобы ресурс null_resource принудительно
  # создавался заново при каждом вызове 'terraform apply'
  triggers = {
    uuid = uuid()
  }
  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

Теперь при каждом выполнении terraformapply будет запускаться средство инициализации local-exec:

```
$ terraform apply
(...)
null_resource.example (local-exec): Hello, World from Darwin x86_64 i386
$ terraform apply
null_resource.example (local-exec): Hello, World from Darwin x86_64 i386
```

Внешний источник данных

Средства инициализации ресурсов обычно являются основным инструментом выполнения скриптов в Terraform, но они не всегда подходят. Иногда нужен скрипт для извлечения данных и предоставления доступа к ним прямо в коде Terraform. Для этого можно использовать источник данных external, который позволяет выполнить внешнюю команду, реализующую определенный протокол.

Этот протокол работает определенным образом.

• Вы можете передавать данные из Terraform во внешнюю программу, используя аргумент query источника данных external. Внешняя программа может читать эти аргументы из стандартного ввода в виде JSON.

• Внешняя программа может передавать данные обратно в Terraform, записывая JSON в стандартный вывод. Остальной код Terraform может извлекать эти данные из JSON с помощью выходного атрибута result, принадлежащего внешнему источнику данных.

Вот пример:

```
data "external" "echo" {
```

```
  program = ["bash", "-c", "cat /dev/stdin"]
  query = {
    foo = "bar"
  }
}
output "echo" {
  value = data.external.echo.result
}
output "echo_foo" {
  value = data.external.echo.result.foo
}
```

В этом примере источник данных external используется для выполнения bash-скрипта, который возвращает обратно в стандартный вывод любые данные, полученные из стандартного ввода. Таким образом, любое значение, которое мы передадим через аргумент query, должно вернуться без изменений в виде выходного атрибута result. Вот что получится, когда мы выполним terraformapply для этого кода:

```
$ terraform apply
(...)
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
echo = {
  "foo" = "bar"
}
echo_foo = bar
```

Как видите, data.external.<NAME>.result содержит ответ в формате JSON, возвращенный внешней программой, и вы можете перемещаться по нему с помощью синтаксиса вида data.external.<NAME>.result.<PATH> (например, data.external.echo.result.foo).

Источник данных external — прекрасный «аварийный люк» на случай, когда вам нужно обращаться к данным в своем коде Terraform и у вас нет такого источника, который бы умел извлекать эти данные. Но не переусердствуйте, используя его и другие «аварийные люки», так как они делают ваш код менее переносимым и более хрупким. Например, источник данных external, который вы только что видели, использует bash. Значит, вы не сможете развернуть этот модуль Terraform из Windows.

Резюме

Итак, вы познакомились со всеми компонентами, необходимыми для создания кода Terraform промышленного уровня. Подытожим. В следующий

раз, когда вы начнете работать над новым модулем, выполните такие действия.

1. Пройдитесь по списку задач для подготовки инфраструктуры промышленного уровня, представленному в табл. 6.2, и определите, какие пункты вы будете реализовывать, а какие нет. Сопоставьте полученный результат с табл. 6.1, чтобы озвучить начальству примерные сроки выполнения.

2. Создайте папку examples и сначала напишите демонстрационный код. Выработайте на его основе максимально удобный и аккуратный API для своих модулей. Создайте по одному примеру для каждого важного сценария применения вашего модуля. Добавьте документацию и предусмотрите разумные значения по умолчанию, чтобы ваш пример было как можно легче развертывать.

3. Создайте папку modules и реализуйте придуманный вами API в виде набора небольших, универсальных и компонуемых модулей. Используйте для этого Terraform в сочетании с другими инструментами, такими как Docker, Packer и bash. Не забудьте закрепить версии Terraform и провайдера.

4. Создайте папку test и напишите автоматические тесты для каждого примера.

Теперь пришло время обсудить написание автоматических тестов для инфраструктурного кода, чем мы и займемся в главе 7.

48 *Хофштадтер Д.* Гедель, Эшер, Бах: эта бесконечная гирлянда. — Самара: Бахрах-М, 2001.

49 Don’t Shave That Yak! (блог Сета, 5 марта 2005 года), bit.ly/2OK45uL.

50 *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. — СПб.: Питер, 2020.

51 *Мартин Р.* Чистый код. Создание, анализ и рефакторинг. Библиотека программиста. — СПб.: Питер, 2019.

52 *Salus P.H.* A Quarter-Century of Unix. — N.Y.: Addison-Wesley Professional, 1994.

53 Все подробности о публикации модулей можно найти на странице bit.ly/2M53hi0.

54 Полный список средств инициализации ресурсов можно найти на странице bit.ly/2M7s1pK.

7. Как тестировать код Terraform

Мир DevOps полон разных страхов: все боятся допустить простой в работе, потерять данные или быть взломанными. При внесении любого изменения вы всегда спрашиваете себя, какие последствия оно будет иметь. Будет ли оно вести себя одинаково во всех окружениях? Вызовет ли оно очередной перебой в работе? И, если это случится, насколько вам придется

задержаться на работе на этот раз, чтобы все исправить? По мере роста компании все больше ставится на кон, что делает процесс развертывания еще страшнее и повышает риск ошибок. Многие компании пытаются минимизировать этот риск за счет менее частых развертываний, но в итоге каждое отдельное развертывание становится более крупным и склонным к ошибкам.

Если вы управляете своей инфраструктурой в виде кода, у вас появляется лучший способ минимизации рисков: тесты. Их цель — дать вам достаточно уверенности для внесения изменений. Ключевым словом здесь является *уверенность*: никакие тесты не могут гарантировать отсутствие ошибок, поэтому вы скорее имеете дело с вероятностью. Если удастся запечатлеть всю свою инфраструктуру и процессы развертывания в виде кода, вы сможете проверить этот код в тестовом окружении. В случае успеха есть большой шанс того, что этот же код будет работать и в промышленных условиях. В мире страха и неопределенности высокая вероятность и уверенность дорого стоят.

В этой главе мы пройдемся по процессу тестирования инфраструктурного кода, как ручного, так и автоматического, с акцентом на последний.

• Ручные тесты:
• основы ручного тестирования;
• очистка ресурсов после тестов.
• Автоматические тесты:
• модульные тесты;
• интеграционные тесты;
• сквозные тесты;
• другие подходы к тестированию.

Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

Ручные тесты

Размышляя о том, как тестировать код Terraform, полезно провести некоторые параллели с тестированием кода, написанного на языках

программирования общего назначения, таких как Ruby. Представьте, что вы пишете простой веб-сервер на Ruby в файле web-server.rb:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

Этот код вернет ответ 200OK с телом Hello,World для URL-адреса /; для любого другого адреса ответ будет 404. Как бы вы протестировали этот код вручную? Обычно для этого добавляют еще немного кода, чтобы запускать веб-сервер локально:

```
# Этот код выполняется, только если скрипт был вызван непосредственно
# из терминала, но не при подключении из другого файла
if __FILE__ == $0
  # Запускаем сервер локально на порте 8000
  server = WEBrick::HTTPServer.new :Port => 8000
  server.mount '/', WebServer
  # Останавливаем сервер нажатием Ctrl+C
  trap 'INT' do server.shutdown end
  # Запускаем сервер
  server.start
end
```

Если запустить этот файл в терминале, он загрузит веб-сервер на порте 8000:

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO  WEBrick 1.3.1
[2019-05-25 14:11:52] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO  WEBrick::HTTPServer#start: pid=19767 port=8000
```

Чтобы проверить работу этого сервера, можно воспользоваться браузером или curl:

```
$ curl localhost:8000/
Hello, World
$ curl localhost:8000/invalid-path
Not Found
```

Теперь представьте, что мы изменили этот код, добавив в него точку входа /api, которая возвращает 201Created и тело в формате JSON:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    when "/api"
      response.status = 201
      response['Content-Type'] = 'application/json'
      response.body = '{"foo":"bar"}'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

Чтобы вручную протестировать этот обновленный код, нажмите Ctrl+C и перезагрузите веб-сервер, запустив скрипт еще раз:

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO  WEBrick 1.3.1
[2019-05-25 14:11:52] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO  WEBrick::HTTPServer#start: pid=19767 port=8000
^C
[2019-05-25 14:15:54] INFO  going to shutdown ...
[2019-05-25 14:15:54] INFO  WEBrick::HTTPServer#start done.
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO  WEBrick 1.3.1
```

[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]

[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767 port=8000

Для проверки новой версии можно снова воспользоваться командой curl:

```
$ curl localhost:8000/api
{"foo":"bar"}
```

Основы ручного тестирования

Как будет выглядеть подобного рода ручное тестирование в Terraform? Например, из предыдущих глав у вас остался код для развертывания ALB. Вот фрагмент файла modules/networking/alb/main.tf:

```
resource "aws_lb" "example" {
  name               = var.alb_name
  load_balancer_type = "application"
  subnets            = var.subnet_ids
  security_groups    = [aws_security_group.alb.id]
}
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"
  # По умолчанию возвращаем простую страницу 404
  default_action {
    type = "fixed-response"
    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
resource "aws_security_group" "alb" {
  name = var.alb_name
}
# (...)
```

Если сравнить этот листинг с кодом на Ruby, можно заметить одно довольно очевидное отличие: AWS ALB, целевые группы, прослушиватели,

группы безопасности и любые другие ресурсы нельзя развертывать на собственном компьютере.

Из этого следует *ключевой вывод о тестировании № 1*: тестирование кода Terraform не может проходить локально.

Это относится не только к Terraform, но и к большинству средств IaC. Единственный практичный способ выполнять ручное тестирование в Terraform — развернуть код в реальном окружении (то есть в AWS). Иными словами, самостоятельный запуск команд terraformapply и terraformdestroy, которым вы занимались, читая книгу, — это и есть ручное тестирование в Terraform.

Это одна из причин, почему так важно иметь в папке examples каждого модуля простые в развертывании примеры (см. главу 6). Чтобы протестировать модуль alb, проще всего воспользоваться демонстрационным кодом, который вы создали в examples/alb:

```
provider "aws" {
  region = "us-east-2"
  # Разрешаем любую версию провайдера AWS вида 2.x
  version = "~> 2.0"
}
module "alb" {
  source = "../../modules/networking/alb"
  alb_name   = "terraform-up-and-running"
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

Чтобы развернуть этот пример, нужно выполнить команду terraformapply, как вы это неоднократно делали:

```
$ terraform apply
(...)
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
Outputs:
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

В конце развертывания можно использовать такой инструмент, как curl, чтобы, к примеру, убедиться, что по умолчанию ALB возвращает 404:

```
$ curl \
  -s \
  -o /dev/null \
  -w "%{http_code}" \
  hello-world-stage-477699288.us-east-2.elb.amazonaws.com
404
```

Проверка инфраструктуры

Наша инфраструктура включает в себя балансировщик нагрузки, работающий по HTTP, поэтому, чтобы убедиться в ее работоспособности, мы используем curl и HTTP-запросы. Другие типы инфраструктуры могут потребовать иные методы проверки. Например, если ваш инфраструктурный код развертывает базу данных MySQL, для его тестирования придется использовать клиент MySQL. Если ваш инфраструктурный код развертывает VPN-сервер, для его тестирования понадобится клиент VPN. Если ваш инфраструктурный код развертывает сервер, который вообще не прослушивает никаких запросов, вам придется зайти на сервер по SSH и выполнить локально кое-какие команды. Этот список можно продолжить. Базовую структуру тестирования, описанную в этой главе, можно применить к инфраструктуре любого вида. Однако этапы проверки будут зависеть от того, что именно вы проверяете.

Напомню: ALB возвращает 404 ввиду отсутствия в конфигурации других правил прослушивателя, а действие по умолчанию в модуле alb имеет ответ 404:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"
  # По умолчанию возвращаем простую страницу 404
  default_action {
    type = "fixed-response"
    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

Итак, вы уже умеете запускать и тестировать свой код. Теперь можно приступить к внесению изменений. Каждый раз, когда вы что-то меняете (чтобы, например, действие по умолчанию возвращало 401), вам нужно использовать команду terraformapply, чтобы развернуть новый код:

```
$ terraform apply
(...)
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
Outputs:
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

Чтобы проверить новую версию, можно заново запустить curl:

```
$ curl \
  -s \
  -o /dev/null \
  -w "%{http_code}" \
  hello-world-stage-477699288.us-east-2.elb.amazonaws.com
401
```

Когда закончите, выполните команду terraformdestroy, чтобы удалить ресурсы:

```
$ terraform destroy
(...)
Apply complete! Resources: 0 added, 0 changed, 5 destroyed.
```

Иными словами, при работе с Terraform каждому разработчику нужны хорошие примеры кода для тестирования и настоящее окружение для разработки (вроде учетной записи AWS), которое служит эквивалентом локального компьютера и используется для выполнения тестов. В процессе ручного тестирования вам, скорее всего, придется создавать и удалять большое количество компонентов инфраструктуры, и это может привести к множеству ошибок. В связи с этим окружение должно быть полностью изолировано от более стабильных сред, предназначенных для финального тестирования и в особенности для промышленного применения.

Учитывая вышесказанное, настоятельно рекомендую каждой команде разработчиков подготовить *изолированную среду*, в которой вы сможете создавать и удалять любую инфраструктуру без последствий. Чтобы минимизировать вероятность конфликтов между разными разработчиками (представьте, что два разработчика пытаются создать балансировщик нагрузки с одним и тем же именем), идеальным решением будет выделить каждому члену команды отдельную, полностью изолированную среду. Например, если вы используете Terraform в сочетании с AWS, каждый разработчик в идеале должен иметь собственную учетную запись, в которой он сможет тестировать все, что ему захочется55.

Очистка ресурсов после тестов

Наличие множества изолированных окружений необходимо для высокой продуктивности разработчиков, но, если не проявлять осторожность, у вас может накопиться много лишних ресурсов, которые захламят все ваши среды и будут стоить вам круглую сумму.

Чтобы держать расходы под контролем, регулярно чистите свои изолированные среды. Это *ключевой вывод о тестировании № 2*.

Вам как минимум следует создать в команде такую культуру, когда после окончания тестирования разработчики удаляют все, что они развернули, с помощью команды terraformdestroy. Возможно, удастся найти инструменты для очистки лишних или старых ресурсов, которые можно запускать на регулярной основе (скажем, с помощью cron). Вот некоторые примеры для разных сред развертывания.

•*cloud-nuke* (http://bit.ly/2OIgM9r). Это инструмент с открытым исходным кодом, который может удалить все ресурсы в вашей облачной среде. Сейчас он поддерживает целый ряд сервисов в AWS (вроде Amazon EC2 Instances, ASG, ELB и т. д.). Поддержка других ресурсов и облаков (Google Cloud, Azure) ожидается в будущем. Его ключевая особенность — возможность удаления всех ресурсов, начиная с определенного возраста. Например, cloud-nuke часто запускают ежедневно с помощью cron, чтобы удалить в каждой изолированной среде все ресурсы старше двух дней. При этом предполагается, что любая инфраструктура, которую разработчик подготовил для ручного тестирования, становится ненужной по прошествии пары суток:

$ cloud-nuke aws --older-than 48h

• *Janitor Monkey* (http://bit.ly/2M4GoLB). Это инструмент с открытым исходным кодом, чистящий ресурсы AWS по графику, который можно настроить (по умолчанию — раз в неделю). Он поддерживает гибкие правила, которые определяют, подлежит ли ресурс удалению, и даже может отправить владельцу ресурса уведомление за несколько дней до очистки. Это часть проекта Netflix Simian Army, в который также входит инструмент Chaos Monkey для проверки устойчивости приложений. Имейте в виду, что у проекта Simian Army больше нет активной поддержки, но разные его части подхватываются другими командами: например, на смену Janitor Monkey пришел инструмент Swabbie (http://bit.ly/2OLrOLb).

• *aws-nuke* (http://bit.ly/2ZB8lOe). Это инструмент с открытым исходным кодом для удаления всего содержимого учетной записи AWS. Учетные записи и ресурсы, подлежащие удалению, указываются в конфигурационном файле в формате YAML:

```
# Регионы для удаления
regions:
```

```
- us-east-2
# Учетные записи для удаления
accounts:
  "111111111111": {}
# Удалять только эти ресурсы
resource-types:
  targets:
  - S3Object
  - S3Bucket
  - IAMRole
```

Запускается aws-nuke следующим образом:

```
$ aws-nuke -5c config.yml
```

Автоматические тесты

Осторожно: впереди много кода!

Написание автоматических тестов для инфраструктурного кода — занятие не для слабонервных. Этот раздел — самая сложная часть книги и требует от читателя повышенного внимания. Если вы просто пролистываете, можете его пропустить. Но если вы действительно хотите научиться тестировать свой инфраструктурный код, закатывайте рукава и готовьтесь к настоящему программированию! Код на Ruby можно не запускать (он нужен лишь для того, чтобы у вас сформировалась общая картина происходящего), а вот код на Go следует записывать и выполнять как можно активней.

Концепция автоматического тестирования состоит в том, что для проверки поведения настоящего кода пишутся тесты. В главе 8 вы узнаете, что с помощью CI-сервера эти тесты можно запускать после каждой отдельной фиксации. Если они не будут пройдены, фиксацию сразу же можно отменить или исправить. Таким образом, ваш код всегда будет в рабочем состоянии.

Существует три типа автоматических тестов.

• *Модульные тесты*. Проверяют функциональность одного небольшого фрагмента кода — *модуля*. Его определение может варьироваться, но в языках программирования общего назначения под ним понимают отдельную функцию или класс. Внешние зависимости (например, базы данных, веб-

сервисы и даже файловая система) заменяются *тестовыми mock-объектами*. Это позволяет полностью контролировать их поведение (например, mock-объект базы данных может возвращать ответ, прописанный вручную) и убедиться в том, что ваш код справляется с множеством разных сценариев.

•*Интеграционные тесты*. Проверяют корректность совместной работы нескольких модулей. В языках программирования общего назначения интеграционный тест состоит из кода, который позволяет убедиться в корректном взаимодействии нескольких функций или классов. Реальные зависимости используются вперемешку с mock-объектами: например, если вы тестируете участок приложения, который обращается к базе данных, стоит тестировать его с настоящей БД, а другие зависимости, скажем, систему аутентификации, заменить mock-объектами.

•*Сквозные тесты*. Подразумевают запуск всей вашей архитектуры (например, приложений, хранилищ данных, балансировщиков нагрузки) и проверку работы системы как единого целого. Обычно эти тесты выполняются как бы от имени конечного пользователя: допустим, Selenium может автоматизировать взаимодействие с вашим продуктом через браузер. В сквозном тестировании везде используются реальные компоненты без каких-либо mock-объектов, а архитектура при этом является отражением настоящей промышленной системы (но с меньшим количеством или с менее дорогими серверами, чтобы сэкономить).

У каждого типа тестов свое назначение, и с их помощью можно выявить разного рода ошибки, поэтому их стоит применять совместно. Модульные тесты выполняются быстро и позволяют сразу получить представление о внесенных изменениях и проверить разнообразные комбинации. Это дает уверенность в том, что элементарные компоненты вашего кода (отдельные модули) ведут себя так, как вы ожидали. Однако тот факт, что модули работают корректно по отдельности, вовсе не означает, что они смогут работать совместно, поэтому, чтобы убедиться в совместимости ваших элементарных компонентов, нужны интеграционные тесты. С другой стороны, корректное поведение разных частей системы не гарантирует, что эта система будет работать как следует после развертывания в промышленной среде, поэтому, чтобы проверить ваш код в условиях, приближенных к реальным, необходимы сквозные тесты.

Теперь посмотрим, как писать тесты каждого из этих типов для кода Terraform.

Модульные тесты

Чтобы понять, как писать модульные тесты для кода Terraform, можно сначала посмотреть на то, как это делается в языках программирования общего назначения, таких как Ruby. Взгляните на код веб-сервера, написанный на Ruby:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    when "/api"
      response.status = 201
      response['Content-Type'] = 'application/json'
      response.body = '{"foo":"bar"}'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

Создание модульного теста для этого кода усложняется тем, что вам нужно выполнить несколько действий.

1. Создать экземпляр класса WebServer. Это сложнее, чем кажется, так как конструктору WebServer, который наследует AbstractServlet, необходимо передать целый класс HTTPServer из WEBrick. Вы могли бы создать его mock-объект, но это потребовало бы много усилий.

2. Создать объект request типа HTTPRequest. Получение экземпляра этого класса, равно как и создание его mock-объекта, требует довольно большого объема работы.

3. Создать объект response типа HTTPResponse. Опять же экземпляр этого класса нельзя получить легким путем, а для создания его mock-объекта нужно много усилий.

Если вам сложно писать модульные тесты, это часто говорит о плохом качестве кода и является поводом для рефакторинга. Чтобы облегчить модульное тестирование, обработчики (то есть код, который обрабатывает пути /, /api и «Страница не найдена») можно вынести в отдельный класс Handlers:

```
class Handlers
```

```
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

У этого нового класса есть два ключевых свойства.

• *Простые значения в качестве ввода*. Класс Handlers не зависит от HTTPServer, HTTPRequest или HTTPResponse. Весь его ввод состоит из простых параметров, таких как строка с URL-адресом.

• *Простые значения в качестве вывода*. Вместо того чтобы менять поля объекта HTTPResponse (создавая тем самым побочный эффект), методы класса Handlers возвращают HTTP-ответ в виде простого значения (массива с кодом состояния, типом содержимого и телом).

Код, который принимает на вход и возвращает простые значения, обычно легче понять, обновить и протестировать. Сначала изменим класс WebServer так, чтобы он отвечал на запросы с помощью Handlers.

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    handlers = Handlers.new
    status_code, content_type, body = handlers.handle(request.path)
    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end
```

Этот код вызывает метод handle из класса Handlers и возвращает полученные из него код статуса, тип содержимого и тело в качестве HTTP-ответа. Как видите, код, использующий класс Handlers, выглядит аккуратным и простым. Кроме того, это облегчает тестирование. Вот как выглядит модульный тест для точки входа /:

```
class TestWebServer < Test::Unit::TestCase
  def initialize(test_method_name)
    super(test_method_name)
    @handlers = Handlers.new
```

```
  end
  def test_unit_hello
    status_code, content_type, body = @handlers.handle("/")
    assert_equal(200, status_code)
    assert_equal('text/plain', content_type)
    assert_equal('Hello, World', body)
  end
end
```

Код теста вызывает тот же метод handle из класса Handlers и использует несколько вызовов assert для проверки ответа, который возвращается из точки входа /. Вот как запустить этот тест:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000287 seconds.
----------------------------------------
1 tests, 3 assertions, 0 failures, 0 errors
100% passed
----------------------------------------
```

Похоже, тест пройден. Теперь добавим модульные тесты для точек входа /api и 404:

```
def test_unit_api
  status_code, content_type, body = @handlers.handle("/api")
  assert_equal(201, status_code)
  assert_equal('application/json', content_type)
  assert_equal('{"foo":"bar"}', body)
end
def test_unit_404
  status_code, content_type, body = @handlers.handle("/invalid-path")
  assert_equal(404, status_code)
  assert_equal('text/plain', content_type)
  assert_equal('Not Found', body)
end
```

Запустите тесты еще раз:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000572 seconds.
----------------------------------------
3 tests, 9 assertions, 0 failures, 0 errors
100% passed
----------------------------------------
```

Всего за 0,000 527 2 секунды вы можете проверить, работает ли код вашего веб-сервера так, как вы того ожидали. В этом сила модульных тестов: быстрые результаты, которые помогают укрепить уверенность в своем коде. Если допущена какая-либо ошибка (например, случайно поменялся ответ точки входа /api), вы узнаете об этом почти мгновенно:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
=====================================================================
Failure: test_unit_api(TestWebServer)
web-server-test.rb:25:in `test_unit_api'
    22:     status_code, content_type, body = Handlers.new.handle("/api")
    23:     assert_equal(201, status_code)
    24:     assert_equal('application/json', content_type)
 => 25:     assert_equal('{"foo":"bar"}', body)
    26:   end
    27:
    28:   def test_unit_404
<"{\"foo\":\"bar\"}"> expected but was
<"{\"foo\":\"whoops\"}">
diff:
? {"foo":"bar   "}
?        whoops
=====================================================================
Finished in 0.007904 seconds.
---------------------------------------
3 tests, 9 assertions, 1 failures, 0 errors
66.6667% passed
---------------------------------------
```

Основы модульных тестов

Какой эквивалент такого рода модульных тестов для кода Terraform? Для начала нужно определиться с тем, что в мире Terraform считается модулем. Ближайший аналог отдельной функции или класса — обобщенный модуль (определение термина дано в подразделе «Компонуемые модули» на с. 226), такой как alb из главы 6. Как бы вы его протестировали?

В Ruby для написания модульных тестов необходимо провести такой рефакторинг, чтобы код можно было выполнять без сложных зависимостей вроде HTTPServer, HTTPRequest или HTTPResponse. Если подумать о том,

чем занимается код Terraform (обращение к API AWS для создания балансировщиков нагрузки, прослушивателей, целевых групп и т. д.), то в 99 % случаев он взаимодействует со сложными зависимостями! Не существует практичного способа свести число внешних зависимостей к нулю, но, даже если бы вы могли это сделать, у вас бы практически не осталось кода для тестирования56.

Это подводит нас к *ключевому выводу о тестировании № 3*: вы не можете проводить *чистое* тестирование кода Terraform.

Но не отчаивайтесь. Вы все еще можете укрепить свою уверенность в том, что ваш код Terraform ведет себя предсказуемо. Для этого автоматические тесты должны использовать ваш код для развертывания реальной инфраструктуры в реальном окружении (например, в настоящей учетной записи AWS). Иными словами, модульные тесты для Terraform на самом деле являются интеграционными. Но я все равно предпочитаю называть их модульными, чтобы подчеркнуть нашу цель: протестировать отдельный (обобщенный) модуль и как можно быстрее получить результат.

Это означает, что базовая стратегия написания модульных тестов для Terraform подразумевает следующее.

1. Создание обобщенного автономного модуля.
2. Создание простого в развертывании примера для этого модуля.
3. Выполнение terraformapply для развертывания примера в реальной среде.
4. Проверка того, что развернутый вами код работает так, как вы ожидали. Этот этап зависит от типа инфраструктуры, которую вы тестируете: например, чтобы проверить балансировщик ALB, ему нужно послать HTTP-запрос и убедиться в том, что он возвращает тот ответ, который вы ожидаете.
5. Выполнение terraformdestroy в конце для очистки ресурсов.

Иными словами, вы выполняете все те же шаги, что и при ручном тестировании, но оформляете их в виде кода. Такой образ мышления хорошо подходит для создания автоматических тестов для кода Terraform: спросите себя, как бы вы проверили данный модуль, чтобы убедиться в его работе, и затем запрограммируйте этот тест.

Для написания тестов подходит любой язык программирования. В этой книге все тесты написаны на языке Go. Это позволяет использовать открытую библиотеку тестирования Terratest (http://bit.ly/2Tbzvch), которая поддерживает широкий спектр инструментов IaC (скажем, Terraform, Packer, Docker, Helm) в разнообразных окружениях (таких как AWS, Google Cloud, Kubernetes). Библиотека Terratest напоминает швейцарский армейский нож: в ней сотни инструментов, которые существенно упрощают тестирование

инфраструктурного кода, включая полноценную поддержку только что описанной стратегии, когда вы развертываете код с помощью terraformapply, убеждаетесь, что он работает, и затем выполняете в конце terraformdestroy, чтобы очистить ресурсы.

Чтобы использовать Terratest, вам нужно сделать следующее.

1. Установить Go: golang.org/doc/install.
2. Настроить переменную среды GOPATH: golang.org/doc/code.html#GOPATH.
3. Добавить $GOPATH/bin в переменную среды PATH.
4. Установить Dep, диспетчер зависимостей для Go: golang.github.io/dep/docs/installation.html57.
5. Создать внутри GOPATH папку для тестов. Учитывая, что переменная GOPATH по умолчанию равна $HOME/go, вы могли бы создать $HOME/go/src/terraform-up-and-running.
6. Выполнить команду depinit в только что созданной вами папке. В результате у вас должны появиться файлы Gopkg.toml и Gopkg.lock, а также пустая папка vendors.

Чтобы быстро проверить, верно ли сконфигурировано ваше окружение, создайте в своей новой папке файл go_sanity_test.go следующего содержания:

```
package test
import (
    "fmt"
    "testing"
)
func TestGoIsWorking(t *testing.T)  {
    fmt.Println()
    fmt.Println("If you see this text, it's working!")
    fmt.Println()
}
```

Выполните этот тест с помощью команды gotest и убедитесь, что она возвращает следующий вывод:

```
$ go test -v
If you see this text, it's working!
PASS
ok  terraform-up-and-running  0.004s
```

Флаг -v означает verbose («подробно»). Он делает так, чтобы тест показывал весь журнальный вывод.

Если все работает, можете удалить go_sanity_test.go и приступить к написанию модульного теста для модуля alb. Создайте в папке test файл

alb_example_test.go со следующим каркасом своего теста:

```
package test
import (
    "testing"
)
func TestAlbExample(t *testing.T) {
}
```

Для начала вы должны указать Terratest, где находится ваш код Terraform. Используйте для этого тип terraform.Options:

```
package test
import (
    "github.com/gruntwork-io/terratest/modules/terraform"
    "testing"
)
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь
        // вел к папке с примерами для alb!
        TerraformDir: "../examples/alb",
    }
}
```

Следует отметить, что для проверки модуля alb вам действительно нужно протестировать код примера в папке examples (обновите относительный путь в TerraformDir, чтобы он вел к нужной папке). Это означает, что демонстрационный код теперь имеет тройное назначение: он служит исполняемой документацией, средством ручного тестирования и инструментом для выполнения автоматических тестов ваших модулей.

Обратите также внимание на новую инструкцию импорта для библиотеки Terratest вверху файла. Чтобы загрузить эту зависимость на ваш компьютер, выполните depensure:

```
$ dep ensure
```

Команда depensure просканирует ваш код на Go, найдет все новые инструкции импорта, автоматически загрузит их вместе со всеми зависимостями в папку vendor и пропишет их в файл Gopkg.lock. Если для вас в этом слишком много магии, можете использовать команду depensure-add, чтобы перечислить все нужные вам зависимости вручную:

```
$ dep ensure -add github.com/gruntwork-io/terratest/modules/terraform
```

Следующий этап автоматического тестирования — выполнение команд terraforminit и terraformapply, которые развернут ваш код. У Terratest для этого есть вспомогательные средства:

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь
        // вел к папке с примерами для alb!
        TerraformDir: "../examples/alb",
    }
    terraform.Init(t, opts)
    terraform.Apply(t, opts)
}
```

Выполнение команд init и apply в Terratest — настолько рутинная операция, что для этого предусмотрен удобный вспомогательный метод, который делает все одной командой:

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь
        // вел к папке с примерами для alb!
        TerraformDir: "../examples/alb",
    }
    // Развертываем пример
    terraform.InitAndApply(t, opts)
}
```

Код выше уже представляет собой довольно функциональный модульный тест: он выполняет terraforminit и terraformapply и проваливает тест, если эти команды не завершаются успешно (например, из-за проблем в вашем коде Terraform). Но вы можете пойти еще дальше и выполнить HTTP-запросы к развернутому балансировщику нагрузки, чтобы убедиться, что он возвращает нужные вам данные. Для этого вам надо как-то получить доменное имя развернутого балансировщика. К счастью, пример alb возвращает его в виде выходной переменной:

```
output "alb_dns_name" {
  value       = module.alb.alb_dns_name
  description = "The domain name of the load balancer"
}
```

У Terratest есть встроенные вспомогательные средства для чтения вывода из кода Terraform:

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь
        // вел к папке с примерами для alb!
        TerraformDir: "../examples/alb",
```

```
        }
        // Развертываем пример
        terraform.InitAndApply(t, opts)
        // Получаем URL-адрес ALB
        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)
    }
```

Функция OutputRequired возвращает вывод с заданным именем или проваливает тест, если этот вывод пустой или не существует. Предыдущий листинг формирует на основе этого вывода URL-адрес, используя встроенную в Go функцию fmt.Sprintf (не забудьте импортировать пакет fmt). Следующим шагом будет выполнение HTTP-запросов по этому URL-адресу:

```
    package test
    import (
        "fmt"
        "crypto/tls"
        "github.com/gruntwork-io/terratest/modules/http-helper"
        "github.com/gruntwork-io/terratest/modules/terraform"
        "testing"
    )
    func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
            // Сделайте так, чтобы этот относительный путь
            // вел к папке с примерами для alb!
            TerraformDir: "../examples/alb",
        }
        // Развертываем пример
        terraform.InitAndApply(t, opts)
        // Получаем URL-адрес ALB
        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)
           // Проверяем в ALB действие по умолчанию, которое должно вернуть 404
        expectedStatus := 404
        expectedBody := "404: page not found"
                    http_helper.HttpGetWithValidation(t, url, expectedStatus, expectedBody)
    }
```

Этот код импортирует из Terratest новый пакет, http_helper. Чтобы его загрузить, нужно еще раз выполнить rundep. Метод http_helper.HttpGetWithValidation сделает HTTP-запрос типа GET по переданному вами URL-адресу и провалит тест, если код состояния и тело ответа не совпадают с теми, которые вы указали.

У этого кода есть одна проблема: между завершением команды terraformapply и тем, когда доменное имя балансировщика нагрузки становится доступным (то есть распространилось по системе), проходит время. Если вызвать http_helper.HttpGetWithValidation незамедлительно, он, вполне вероятно, окажется неудачным, хотя через 30 секунд или минуту ALB заработает в нормальном режиме. Как уже обсуждалось в подразделе «Отложенная согласованность согласуется… с отлагательством» на с. 211, такого рода асинхронное поведение с отложенной согласованностью является для AWS (а точнее, для большинства распределенных систем) нормальным и решением будет повторное выполнение вызовов. У Terratest есть вспомогательный метод и для этого:

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь
        // вел к папке с примерами для alb!
        TerraformDir: "../examples/alb",
    }
    // Развертываем пример
    terraform.InitAndApply(t, opts)
    // Получаем URL-адрес ALB
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)
       // Проверяем в ALB действие по умолчанию, которое должно
вернуть 404
    expectedStatus := 404
    expectedBody := "404: page not found"
    maxRetries := 10
    timeBetweenRetries := 10 * time.Second
    http_helper.HttpGetWithRetry(
        t,
        url,
        &tls.Config{},
        expectedStatus,
        expectedBody,
        maxRetries,
```

```
            timeBetweenRetries,
        )
    }
```

Метод http_helper.HttpGetWithRetry почти не отличается от метода http_helper.HttpGetWithValidation, только при отсутствии ответа с нужными кодом состояния или телом он повторит попытку. Максимальное количество повторений (десять) и длину интервалов между ними (десять секунд) можно настроить. Если в какой-то момент он получит ожидаемый ответ, тест будет пройден; если после истечения максимального количества попыток ожидаемый ответ так и не пришел, тест считается проваленным.

В конце теста нужно выполнить команду terraformdestroy, чтобы очистить ресурсы. Для этого у Terratest есть вспомогательный метод: terraform.Destroy. Но, если вызывать его в самом конце, из-за любой программной ошибки выше по коду (например, HttpGetWithRetry даст сбой из-за неправильной конфигурации ALB) тест может завершиться, не доходя до него, в результате чего развернутая инфраструктура не будет удалена.

Таким образом, вам нужно убедиться в том, что terraform.Destroy выполняется всегда, даже если тест проваливается. Во многих языках программирования для этого предусмотрены конструкции try/finally или try/ensure. Но в Go это делается с помощью выражения defer, которое гарантирует, что переданный в него код будет выполнен при завершении окружающей его функции (каким бы оно ни было):

```
    func TestAlbExample(t *testing.T) {
        opts := &terraform.Options{
            // Сделайте так, чтобы этот относительный путь
            // вел к папке с примерами для alb!
            TerraformDir: "../examples/alb",
        }
        // Удаляем все в конце теста
        defer terraform.Destroy(t, opts)
        // Развертываем пример
        terraform.InitAndApply(t, opts)
        // Получаем URL-адрес ALB
        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)
           // Проверяем в ALB действие по умолчанию, которое должно
вернуть 404
        expectedStatus := 404
        expectedBody := "404: page not found"
        maxRetries := 10
```

```
        timeBetweenRetries := 10 * time.Second
        http_helper.HttpGetWithRetry(
            t,
            url,
            &tls.Config{},
            expectedStatus,
            expectedBody,
            maxRetries,
            timeBetweenRetries,
        )
    }
```

Обратите внимание, что defer размещается в начальной части кода, еще до вызова terraform.InitAndApply. Это нужно для того, чтобы тест не был провален еще до выражения defer, иначе вызов terraform.Destroy может не попасть в очередь на выполнение.

Этот модульный тест готов к запуску. Поскольку он развертывает инфраструктуру в AWS, перед его выполнением следует привычным способом аутентифицировать свою учетную запись (см. врезку «Способы аутентификации» на с. 64). Как упоминалось в этой главе, ручное тестирование следует проводить в изолированной учетной записи. Это вдвойне справедливо для автоматических тестов, поэтому советую сделать аутентификацию от имени совершенно отдельного пользователя. По мере создания все новых автоматических тестов у вас могут создаваться сотни и тысячи ресурсов, поэтому крайне важно изолировать их от всего остального.

Я обычно рекомендую командам разработчиков выделить совершенно отдельную среду (например, в отдельной учетной записи AWS) специально для автоматических тестов — отдельно даже от изолированных окружений, которые используются для ручного тестирования. Так, можно безопасно удалять в этой среде все ресурсы старше нескольких часов (предполагается, что ни один тест не выполняется настолько долго).

После входа в учетную запись AWS, которую можно безопасно применять для тестирования, запустите тест:

```
$ go test -v -timeout 30m
TestAlbExample 2019-05-26T13:29:32+01:00 command.go:53:
Running command terraform with args [init -upgrade=false]
(...)
TestAlbExample 2019-05-26T13:29:33+01:00 command.go:53:
Running command terraform with args [apply -input=false -lock=false]
(...)
TestAlbExample 2019-05-26T13:32:06+01:00 command.go:121:
```

```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
(...)
TestAlbExample 2019-05-26T13:32:06+01:00 command.go:53:
Running command terraform with args [output -no-color alb_dns_name]
(...)
TestAlbExample 2019-05-26T13:38:32+01:00 http_helper.go:27:
Making an HTTP GET call to URL
http://terraform-up-and-running-1892693519.us-east-2.elb.amazonaws.com
(...)
TestAlbExample 2019-05-26T13:38:32+01:00 command.go:53:
Running command terraform with args
[destroy -auto-approve -input=false -lock=false]
(...)
TestAlbExample 2019-05-26T13:39:16+01:00 command.go:121:
Destroy complete! Resources: 5 destroyed.
(...)
PASS
ok   terraform-up-and-running   229.492s
```

Обратите внимание на аргумент -timeout30m, который используется вместе с командой gotest. Go по умолчанию ограничивает время работы тестов десятью минутами; когда это время заканчивается, тестовое выполнение принудительно завершается, в результате чего тест не просто проваливается, но даже не доходит до кода очистки (terraformdestroy). Эта проверка ALB должна занять около пяти минут, но при реализации тестов, которые развертывают реальную инфраструктуру, время ожидания лучше увеличить, чтобы тест не прервался на полпути и не оставил после себя разного рода инфраструктуру.

Этот тест сгенерирует длинный журнальный вывод, но, если внимательно почитать, можно заметить все его ключевые моменты.

1. Выполнение terraforminit.
2. Выполнение terraformapply.
3. Чтение выходной переменной с помощью terraformapply.
4. Неоднократная отправка HTTP-запросов к ALB.
5. Выполнение terraformdestroy.

Этот код работает несравнимо медленнее модульных тестов на Ruby, но теперь менее чем за пять минут вы автоматически узнаете, ведет ли себя ли ваш модуль alb так, как вы того ожидали. Для инфраструктуры AWS это максимально быстро, и в результате вы должны увериться в том, что ваш код работает как следует. Если в вашем примере есть какая-либо ошибка

(например, код состояния для действия по умолчанию был случайно изменен на 401), вы довольно скоро об этом узнаете:

```
$ go test -v -timeout 30m
(...)
Validation failed for URL
http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com.
Response status: 401. Response body: 404: page not found.
(...)
Sleeping for 10s and will try again.
(...)
Validation failed for URL
http://terraform-up-and-running-h2ezYz-931760451.us-east-
2.elb.amazonaws.com.
Response status: 401. Response body: 404: page not found.
(...)
Sleeping for 10s and will try again.
(...)
--- FAIL: TestAlbExample (310.19s)
    http_helper.go:94:
    HTTP GET to URL
    http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com
    unsuccessful after 10 retries
FAIL    terraform-up-and-running    310.204s
```

Внедрение зависимостей

Теперь посмотрим, сколько усилий потребуется, чтобы создать модульный тест для чуть более сложного кода. Еще раз вернемся к примеру с веб-сервером на Ruby и представим, что нам нужно добавить новую точку входа /web-service, которая шлет HTTP-вызовы к внешней зависимости:

```
class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
      # Новая точка входа, которая вызывает веб-сервис
      uri = URI("http://www.example.org")
      response = Net::HTTP.get_response(uri)
```

```
      [response.code.to_i, response['Content-Type'], response.body]
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

Обновленный класс Handlers обрабатывает URL-адрес /web-service, отправляя HTTP-запрос типа GET на example.org и передавая ответ. Если обратиться к этой точке входа с помощью curl, получится следующий результат:

```
$ curl localhost:8000/web-service
<!doctype html>
<html>
<head>
  <title>Example Domain</title>
  <-- (...) -->
</head>
<body>
<div>
  <h1>Example Domain</h1>
  <p>
    This domain is established to be used for illustrative
    examples in documents. You may use this domain in
    examples without prior coordination or asking for permission.
  </p>
  <!-- (...) -->
</div>
</body>
</html>
```

Как написать модульный тест для этого нового метода? Если проверять код как есть, тестирование будет зависеть от поведения внешнего компонента (в данном случае сервиса example.org). Есть ряд отрицательных последствий.

• Если у этой зависимости возникнут перебои в работе, ваш тест провалится, несмотря на то что с вашим кодом все в порядке.

• Если эта зависимость со временем поменяет свое поведение (например, начнет возвращать другое тело ответа), ваши тесты будут периодически проваливаться. Придется постоянно обновлять их код, несмотря на то что у вашей реализации нет никаких проблем.

• Если эта зависимость медленная, она притормозит ваши тесты, что нивелирует одно из главных преимуществ модульного тестирования — быструю обратную связь.

• Если вы захотите убедиться в том, что ваш код справляется с различными крайними случаями, связанными с поведением этой внешней зависимости (например, как ваш код реагирует на перенаправление?), придется делать это без контроля над ней.

Работа с настоящими зависимостями важна в интеграционном и сквозном тестировании. Однако в модульных тестах количество внешних зависимостей следует минимизировать. Типичной стратегией для этого является *внедрение зависимостей* — когда вы передаете (или внедряете) внешние зависимости за пределами своего кода, а не прописываете их вручную.

Например, класс Handlers не должен знать все подробности о том, как вызывать веб-сервис. Эту логику можно вынести в отдельный класс WebService:

```
class WebService
  def initialize(url)
    @uri = URI(url)
  end
  def proxy
    response = Net::HTTP.get_response(@uri)
     [response.code.to_i, response['Content-Type'], response.body]
  end
end
```

Этот класс принимает на вход URL-адрес и предоставляет метод proxy для проксирования с этого адреса HTTP-ответа типа GET. Мы можем сделать так, чтобы класс Handlers принимал в качестве ввода экземпляр WebService и использовал его в своем методе web_service:

```
class Handlers
  def initialize(web_service)
    @web_service = web_service
  end
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
```

```
      # Новая точка входа, которая вызывает веб-сервис
      @web_service.proxy
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

Теперь в коде реализации можно внедрить реальный экземпляр WebService, который шлет HTTP-вызовы к example.org:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    web_service = WebService.new("http://www.example.org")
    handlers = Handlers.new(web_service)
    status_code, content_type, body = handlers.handle(request.path)
    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end
```

Вы можете создать в коде своего теста mock-объект класса WebService, который позволяет указать фиктивный ответ:

```
class MockWebService
  def initialize(response)
    @response = response
  end
  def proxy
    @response
  end
end
```

Теперь вы можете создать экземпляр этого класса MockWebService и внедрить его в класс Handlers в своих модульных тестах:

```
def test_unit_web_service
  expected_status = 200
  expected_content_type = 'text/html'
  expected_body = 'mock example.org'
        mock_response = [expected_status, expected_content_type, expected_body]
  mock_web_service = MockWebService.new(mock_response)
  handlers = Handlers.new(mock_web_service)
  status_code, content_type, body = handlers.handle("/web-service")
```

```
  assert_equal(expected_status, status_code)
  assert_equal(expected_content_type, content_type)
  assert_equal(expected_body, body)
end
```

Выполните тесты еще раз, чтобы убедиться в том, что они по-прежнему работают:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Started
...
Finished in 0.000645 seconds.
-------------------------------------------
4 tests, 12 assertions, 0 failures, 0 errors
100% passed
-------------------------------------------
```

Превосходно! Этот подход минимизирует внешние зависимости, позволяя писать быстрые и надежные тесты, а также проверять всевозможные крайние случаи. И, поскольку три тестовых случая, которые вы добавили ранее, по-прежнему успешно выполняются, вы можете быть уверены, что ваш рефакторинг ничего не сломал.

Теперь вернемся к Terraform и посмотрим, как внедрение зависимостей будет выглядеть для наших модулей. Начнем с hello-world-app. Сперва следует создать простой в развертывании пример в папке examples (если вы этого еще не сделали):

```
provider "aws" {
  region = "us-east-2"
  # Разрешаем любую версию провайдера AWS вида 2.x
  version = "~> 2.0"
}
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"
  server_text = "Hello, World"
  environment = "example"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "examples/terraform.tfstate"
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

Проблема с зависимостями сразу бросается в глаза: hello-world-app предполагает, что вы уже развернули модуль mysql, и требует, чтобы вы предоставили информацию о бакете S3, в котором mysql хранит свое состояние. Для этого предусмотрены аргументы db_remote_state_bucket и db_remote_state_key. Наша цель — создать для hello-world-app модульный тест и по возможности минимизировать число внешних зависимостей, хотя свести его к нулю в Terraform не получится.

Один из первых шагов по минимизации зависимостей состоит в прояснении того, от чего зависит ваш модуль. Для этого все источники данных и ресурсы, представляющие внешние зависимости, можно вынести в отдельный файл dependencies.tf. Так будет выглядеть файл modules/services/hello-world-app/dependencies.tf:

```
data "terraform_remote_state" "db" {
  backend = "s3"
  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}
data "aws_vpc" "default" {
  default = true
}
data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}
```

Это помогает пользователям вашего кода понять с первого взгляда, на какие внешние компоненты он полагается. В случае с модулем hello-world-app можно сразу же увидеть, что он зависит от базы данных, VPC и подсетей. Но как внедрить зависимости снаружи модуля, чтобы их можно было заменить во время тестирования? Вы уже знаете ответ: входные переменные.

Для каждой из этих зависимостей в файле modules/services/hello-world-app/variables.tf нужно добавить входную переменную:

```
variable "vpc_id" {
  description = "The ID of the VPC to deploy into"
  type        = string
  default     = null
}
variable "subnet_ids" {
```

```
  description = "The IDs of the subnets to deploy into"
  type        = list(string)
  default     = null
}
variable "mysql_config" {
  description = "The config for the MySQL DB"
  type        = object({
    address = string
    port    = number
  })
  default     = null
}
```

Теперь у нас есть по одной входной переменной для VPC ID, идентификаторов подсетей и конфигурации MySQL. У каждой есть поле default, поэтому все они *опциональные*: пользователь может указать для них собственные значения или оставить default. Значение по умолчанию, которое используется для этих переменных, вам еще не встречалось: null. Если бы вместо этого в качестве default было указано *пустое значение* (например, пустая строка для vpc_id или пустой список для subnet_ids), вы бы не смогли отличить его от такого же значения, сознательно установленного пользователем. В таких случаях может пригодиться значение null, так как оно сигнализирует о том, что переменная не задана и что пользователь полагается на поведение по умолчанию.

Обратите внимание: переменная mysql_config имеет конструктор типа object для создания вложенного типа с ключами address и port. Этот тип специально предназначен для того, чтобы соответствовать выходным типам модуля mysql:

```
output "address" {
  value       = aws_db_instance.example.address
  description = "Connect to the database at this endpoint"
}
output "port" {
  value       = aws_db_instance.example.port
  description = "The port the database is listening on"
}
```

Одно из преимуществ этого состоит в том, что по завершении рефакторинга у вас появится возможность совместного использования модулей hello-world-app и mysql следующим образом:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"
```

```
  server_text            = "Hello, World"
  environment            = "example"
  # Все выходные переменные из модуля mysql передаются напрямую!
  mysql_config = module.mysql
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
module "mysql" {
  source = "../../../modules/data-stores/mysql"
  db_name     = var.db_name
  db_username = var.db_username
  db_password = var.db_password
}
```

Поскольку типы mysql_config и выходных переменных модуля mysql совпадают, вы можете передавать их напрямую одной строчкой. И если типы когда-нибудь поменяются и перестанут совпадать, Terraform сразу же выдаст вам ошибку, чтобы вы знали, что их нужно обновить. Это пример не простой, а типобезопасной композиции функций.

Но, чтобы это заработало, нужно закончить рефакторинг кода. Поскольку конфигурацию MySQL можно передавать в качестве ввода, значит, переменные db_remote_state_bucket и db_remote_state_key должны теперь быть опциональными, поэтому присвойте им значение null по умолчанию:

```
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the DB's Terraform state"
  type        = string
  default     = null
}
variable "db_remote_state_key" {
  description = "The path in the S3 bucket for the DB's Terraform state"
  type        = string
  default     = null
}
```

Дальше используйте параметр count, чтобы опционально создать в файле modules/services/hello-world-app/dependencies.tf три источника данных в зависимости от того, установлено ли значение null соответствующей входной переменной:

```
data "terraform_remote_state" "db" {
```

```
  count = var.mysql_config == null ? 1 : 0
  backend = "s3"
  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}
data "aws_vpc" "default" {
  count   = var.vpc_id == null ? 1 : 0
  default = true
}
data "aws_subnet_ids" "default" {
  count  = var.subnet_ids == null ? 1 : 0
  vpc_id = data.aws_vpc.default.id
}
```

Теперь нужно обновить все ссылки на эти источники данных так, чтобы пользователь мог выбирать между источником и входной переменной. Оформим это в виде локальных значений:

```
locals {
  mysql_config = (
    var.mysql_config == null
      ? data.terraform_remote_state.db[0].outputs
      : var.mysql_config
  )
  vpc_id = (
    var.vpc_id == null
      ? data.aws_vpc.default[0].id
      : var.vpc_id
  )
  subnet_ids = (
    var.subnet_ids == null
      ? data.aws_subnet_ids.default[0].ids
      : var.subnet_ids
  )
}
```

Заметьте, что источники данных теперь являются массивами, так как они используют параметр count, поэтому каждый раз при ссылке на них нужно применять синтаксис поиска по массиву (например, [0]). Пройдитесь по коду и вместо ссылок на эти источники подставьте ссылки на

соответствующие локальные значения. Сначала сделайте так, чтобы источник aws_subnet_ids применял local.vpc_id:

```
data "aws_subnet_ids" "default" {
  count = var.subnet_ids == null ? 1 : 0
  vpc_id = local.vpc_id
}
```

Затем присвойте local.subnet_ids параметрам subnet_ids в модулях asg и alb:

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"
  cluster_name  = "hello-world-${var.environment}"
  ami           = var.ami
  user_data     = data.template_file.user_data.rendered
  instance_type = var.instance_type
  min_size          = var.min_size
  max_size          = var.max_size
  enable_autoscaling = var.enable_autoscaling
  subnet_ids        = local.subnet_ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"
  custom_tags = var.custom_tags
}
module "alb" {
  source = "../../networking/alb"
  alb_name   = "hello-world-${var.environment}"
  subnet_ids = local.subnet_ids
}
```

Обновите переменные db_address и db_port внутри user_data так, чтобы они использовали local.mysql_config:

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")
  vars = {
    server_port = var.server_port
    db_address  = local.mysql_config.address
    db_port     = local.mysql_config.port
    server_text = var.server_text
  }
}
```

И присвойте параметру vpc_id ресурса aws_lb_target_group значение local.vpc_id:

```
resource "aws_lb_target_group" "asg" {
  name     = "hello-world-${var.environment}"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = local.vpc_id
  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

Благодаря этим изменениям вы теперь можете при желании внедрить VPC ID, идентификаторы подсетей и/или конфигурационные параметры MySQL в модуль hello-world-app. Но, если их опустить, модуль возьмет подходящий источник данных и сам извлечет эти значения. Обновим пример Hello World, позволив внедрять конфигурацию MySQL, но при этом опустим VPC ID и идентификаторы подсетей, так как VPC по умолчанию вполне подходит для тестирования. Добавьте в файл examples/hello-world-app/variables.tf новую входную переменную:

```
variable "mysql_config" {
  description = "The config for the MySQL DB"
  type = object({
    address = string
    port    = number
  })
  default = {
    address = "mock-mysql-address"
    port    = 12345
  }
}
```

Передайте эту переменную модулю hello-world-app в файле examples/hello-world-app/main.tf:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"
  server_text = "Hello, World"
  environment = "example"
```

```
  mysql_config = var.mysql_config
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

Теперь в модульном тесте переменной mysql_config можно присвоить любое значение на ваш выбор. Создайте модульный тест test/hello_world_app_example_test.go следующего содержания:

```
func TestHelloWorldAppExample(t *testing.T) {
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь вел к папке
        // с примерами для hello-world-app!
        TerraformDir: "../examples/hello-world-app/standalone",
    }
    // Очищаем все ресурсы в конце теста
    defer terraform.Destroy(t, opts)
    terraform.InitAndApply(t, opts)
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)
    expectedStatus := 200
    expectedBody := "Hello, World"
    maxRetries := 10
    timeBetweenRetries := 10 * time.Second
    http_helper.HttpGetWithRetry(
        t,
        url,
        &tls.Config{},
        expectedStatus,
        expectedBody,
        maxRetries,
        timeBetweenRetries,
    )
}
```

Этот модульный тест почти идентичен тому, который мы создали для примера с alb. Единственное отличие — параметр TerraformDir указывает на пример hello-world-app (не забудьте обновить путь в соответствии со своей файловой системой), а ожидаемый ответ от ALB имеет код состояния 200OK и тело HelloWorld. В этот тест нужно добавить всего один новый элемент — переменную mysql_config:

```
opts := &terraform.Options{
    // Сделайте так, чтобы этот относительный путь вел к папке
    // с примерами для hello-world-app!
    TerraformDir: "../examples/hello-world-app/standalone",
    Vars: map[string]interface{}{
        "mysql_config": map[string]interface{}{
            "address": "mock-value-for-test",
            "port": 3306,
        },
    },
}
```

Параметр Vars в terraform.Options позволяет устанавливать переменные в коде Terraform. Этот код передает некоторые фиктивные данные для переменной mysql_config. Вы можете использовать для этого любое значение, например IP-адрес небольшой базы данных, полностью размещенной в оперативной памяти, которая запускается на время тестирования.

Выполните команду gotest с аргументом -run, чтобы запустить *только* этот новый тест (по умолчанию Go запускает все тесты в текущей папке, включая тот, который вы создали для примера с ALB):

```
$ go test -v -timeout 30m -run TestHelloWorldAppExample
(...)
PASS
ok  terraform-up-and-running  204.113s
```

Если все пройдет хорошо, тест выполнит terraformapply и начнет слать HTTP-запросы к балансировщику нагрузки. Как только придет ответ, он выполнит команду terraformdestroy, чтобы очистить все ресурсы. В общей сложности все это должно занять всего пару минут. Теперь у вас есть неплохой модульный тест для приложения Hello, World.

Параллельное выполнение тестов

В предыдущем пункте вы запускали тесты по одному, используя команду gotest с аргументом -run. Если опустить последний, Go выполнит все ваши тесты — по очереди. Четыре-пять минут на один тест — не так уж плохо для проверки инфраструктурного кода, но, если таких тестов десятки и все они запускаются последовательно, это может занять несколько часов. Чтобы ускорить обратную связь, тесты по возможности лучше распараллеливать.

Для параллельного выполнения в начало каждого теста нужно добавить t.Parallel(). Вот как это выглядит на примере

test/hello_world_app_example_test.go:

```
func TestHelloWorldAppExample(t *testing.T) {
    t.Parallel()
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь вел
        // к папке с примерами для hello-world-app!
        TerraformDir: "../examples/hello-world-app/standalone",
        Vars: map[string]interface{}{
            "mysql_config": map[string]interface{}{
                "address": "mock-value-for-test",
                "port": 3306,
            },
        },
    }
    // (...)
}
```

И аналогично в файле test/alb_example_test.go:

```
func TestAlbExample(t *testing.T) {
    t.Parallel()
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь вел
        // к папке с примерами для alb!
        TerraformDir: "../examples/alb",
    }
    // (...)
}
```

Теперь, если выполнить gotest, эти тесты запустятся параллельно. Но есть одна загвоздка: некоторые ресурсы, создаваемые этими тестами (например, ASG, группа безопасности и ALB), имеют одинаковые имена. В результате из-за конфликта имен тесты будут провалены. Но даже без t.Parallel(), если одни и те же тесты выполняются разными членами команды или запускаются внутри среды CI, подобные конфликты неизбежны.

Из этого следует *ключевой вывод о тестировании № 4*: все ваши ресурсы должны быть разделены по пространствам имен.

Иными словами, пишите свои модули и примеры таким образом, чтобы имя каждого ресурса можно было при желании сконфигурировать. В случае с примером для alb это означает, что конфигурируемым должно быть имя балансировщика. Добавьте в файл examples/alb/variables.tf новую входную переменную с разумным значением по умолчанию:

```
variable "alb_name" {
```

```
  description = "The name of the ALB and all its resources"
  type        = string
  default     = "terraform-up-and-running"
}
```

Затем передайте это значение модулю alb в файле examples/alb/main.tf:

```
module "alb" {
  source     = "../../modules/networking/alb"
  alb_name   = var.alb_name
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

Теперь откройте файл test/alb_example_test.go и присвойте этой переменной уникальное значение:

```
package test
import (
	"fmt"
	"crypto/tls"
	"github.com/gruntwork-io/terratest/modules/http-helper"
	"github.com/gruntwork-io/terratest/modules/random"
	"github.com/gruntwork-io/terratest/modules/terraform"
	"testing"
	"time"
)
func TestAlbExample(t *testing.T) {
	t.Parallel()
	opts := &terraform.Options{
		// Сделайте так, чтобы этот относительный путь вел
		// к папке с примерами для alb!
		TerraformDir: "../examples/alb",
		Vars: map[string]interface{}{
			"alb_name": fmt.Sprintf("test-%s", random.UniqueId()),
		},
	}
	// (...)
}
```

Обратите внимание на новый вспомогательный метод из пакета random, входящего в состав Terratest. Вам нужно еще раз выполнить depensure. Этот код присваивает переменной alb_name значение test-<RANDOM_ID>, где RANDOM_ID — случайный идентификатор, возвращаемый вспомогательным методом random.UniqueId() из Terratest. Данный метод возвращает рандомизированную строку из шести символов в кодировке

base-62. Это короткий идентификатор, который можно добавлять к именам большинства ресурсов, не превышая максимальную длину, и который достаточно случайный для того, чтобы сделать конфликты крайне маловероятными (626 — это более 56 миллиардов комбинаций). Благодаря этому вы сможете запускать параллельно огромное количество тестов ALB, не заботясь о конфликтах имен.

Аналогично модифицируйте пример Hello, World. Для начала добавьте новую входную переменную в файл examples/hello-world-app/variables.tf:

```
variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
  default     = "example"
}
```

Затем передайте эту переменную модулю hello-world-app:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"
  server_text = "Hello, World"
  environment = var.environment
  mysql_config = var.mysql_config
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

И присвойте переменной environment в файле test/hello_world_app_example_test.go значение, которое включает в себя random.UniqueId():

```
func TestHelloWorldAppExample(t *testing.T) {
    t.Parallel()
    opts := &terraform.Options{
        // Сделайте так, чтобы этот относительный путь вел
        // к папке с примерами для hello-world-app!
        TerraformDir: "../examples/hello-world-app/standalone",
        Vars: map[string]interface{}{
            "mysql_config": map[string]interface{}{
                "address": "mock-value-for-test",
                "port": 3306,
            },
            "environment": fmt.Sprintf("test-%s", random.UniqueId()),
        },
```

```
        }
        // (...)
    }
```

После внесения этих изменений параллельный запуск ваших тестов должен быть безопасным:

```
$ go test -v -timeout 30m
TestAlbExample 2019-05-26T17:57:21+01:00 (...)
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)
TestAlbExample 2019-05-26T17:57:21+01:00 (...)
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)
(...)
PASS
ok  terraform-up-and-running  216.090s
```

Вы должны увидеть, что оба теста выполняются одновременно. Таким образом, общее время тестирования должно занимать примерно столько времени, сколько нужно для завершения самого медленного теста в вашем наборе (а не совокупное время работы всех тестов, как в случае с последовательным запуском).

Параллельное выполнение тестов в одной и той же папке

Еще один аспект параллелизма, который следует учитывать, — ситуация, когда несколько автоматических тестов запускаются для одной и той же папки Terraform. Например, вы могли бы запустить несколько разных тестов для examples/hello-world-app, каждый из которых перед выполнением terraform apply присваивает свои собственные значения входным переменным. Если вы попробуете это сделать, возникнет проблема: ваши тесты начнут конфликтовать, поскольку все они выполняют terraform init и тем самым переопределяют файлы состояния Terraform в папке .terraform.

Если вы хотите параллельно запускать несколько тестов для одной и той же папки, самое простое решение — сделать так, чтобы каждый тест копировал состояние в уникальную папку и запускал Terraform из нее, чтобы избежать конфликтов. У Terratest для этого есть встроенный вспомогательный метод, который даже следит за корректным размещением относительных путей для модулей Terraform: подробности ищите в документации к методу test_structure.CopyTerraformFolderToTemp.

Интеграционные тесты

Подготовив несколько модульных тестов, мы можем переходить к модульному тестированию. Чтобы сформировать общее понимание, которое позже можно будет применить к коду Terraform, лучше начать с примера веб-сервера, написанного на Ruby. Для интеграционного тестирования такого сервера нужно выполнить следующее.

1. Запустить веб-сервер на локальном компьютере так, чтобы он прослушивал порт.

2. Отправить ему HTTP-запросы.

Создадим в файле web-server-test.rb метод, который выполняет эти шаги:

```
def do_integration_test(path, check_response)
  port = 8000
  server = WEBrick::HTTPServer.new :Port => port
  server.mount '/', WebServer
  begin
    # Запускаем веб-сервер в отдельном потоке, чтобы он не блокировал тест
    thread = Thread.new do
      server.start
    end
    # Отправляем HTTP-запрос по определенному пути веб-сервера
    uri = URI("http://localhost:#{port}#{path}")
    response = Net::HTTP.get_response(uri)
      # Используем для проверки ответа заданную лямбда-функцию check_response
    check_response.call(response)
  ensure
    # В конце теста останавливаем сервер и поток
    server.shutdown
    thread.join
  end
end
```

Метод do_integration_test конфигурирует веб-сервер на порте 8000, запускает его в фоновом потоке (чтобы он не блокировал выполнение теста), шлет HTTP-запрос типа GET по заданному пути, передает HTTP-ответ на проверку функции check_response и в конце останавливает веб-сервер. Вот как с помощью этого метода можно написать интеграционный тест для точки входа /:

```
def test_integration_hello
  do_integration_test('/', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Hello, World', response.body)
  })
end
```

Этот код вызывает метод do_integration_test с путем / и передает ему лямбду (встраиваемую функцию), которая проверяет, равны ли код и тело ответа 200OK и соответственно Hello,World. Похожи интеграционные тесты и для других точек входа, хотя для /web-service выполняется более общая проверка (то есть assert_include вместо assert_equal), чтобы минимизировать потенциальные проблемы при изменении точки входа example.org:

```
def test_integration_api
  do_integration_test('/api', lambda { |response|
    assert_equal(201, response.code.to_i)
    assert_equal('application/json', response['Content-Type'])
    assert_equal('{"foo":"bar"}', response.body)
  })
end
def test_integration_404
  do_integration_test('/invalid-path', lambda { |response|
    assert_equal(404, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Not Found', response.body)
  })
end
def test_integration_web_service
  do_integration_test('/web-service', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_include(response['Content-Type'], 'text/html')
    assert_include(response.body, 'Example Domain')
  })
end
```

Запустим все наши тесты:

```
$ ruby web-server-test.rb
(...)
Finished in 0.221561 seconds.
---------------------------------------------
8 tests, 24 assertions, 0 failures, 0 errors
```

100% passed

Обратите внимание, что при модульном тестировании на выполнение всех тестов уходило 0,000572 секунды. Теперь же это время выросло примерно в 387 раз, до 0,221561 секунды. Конечно, выполнение по-прежнему происходит молниеносно, но это связано лишь с тем, что код веб-сервера на Ruby мало что умеет. Этот пример специально сделан как можно более компактным. Здесь важны не абсолютные показатели, а относительная тенденция: интеграционные тесты обычно работают медленней, чем модульные. Позже мы еще к этому вернемся.

Теперь сосредоточимся на интеграционном тестировании кода Terraform. Если раньше мы тестировали отдельные модули, то теперь нужно проверить, как они работают вместе. Для этого мы должны развернуть их одновременно и убедиться в том, что они ведут себя корректно. Ранее мы развернули демонстрационное приложение Hello, World с фиктивными данными вместо настоящей БД MySQL. Теперь развернем реальный модуль MySQL и посмотрим, как с ним интегрируется приложение Hello, World. У вас уже должен быть подходящий код в папках live/stage/data-stores/mysql и live/stage/services/hello-world-app. Таким образом, вы можете создать интеграционный тест для своей среды финального тестирования (точнее, для ее части).

Конечно, как уже упоминалось в этой главе, все автоматические тесты должны выполняться в изолированной учетной записи AWS. Поэтому при проверке кода для среды финального тестирования все тесты следует запускать от имени изолированного тестового пользователя. Если в ваших модулях есть код, который специально прописан для среды финального тестирования, самое время сделать его конфигурируемым, чтобы вы могли внедрять тестовые значения. Позвольте менять в файле live/stage/data-stores/mysql/variables.tf название базы данных, используя новую входную переменную db_name:

```
variable "db_name" {
  description = "The name to use for the database"
  type        = string
  default     = "example_database_stage"
}
```

Передайте это значение модулю mysql в файле live/stage/data-stores/mysql/main.tf:

```
module "mysql" {
  source = "../../../../modules/data-stores/mysql"
  db_name     = var.db_name
```

```
  db_username = var.db_username
  db_password = var.db_password
}
```

Теперь создадим в файле test/hello_world_integration_test.go каркас интеграционного теста, а детали реализации оставим на потом:

```
// Подставьте сюда подходящие пути к вашим модулям
const dbDirStage = "../live/stage/data-stores/mysql"
const appDirStage = "../live/stage/services/hello-world-app"
func TestHelloWorldAppStage(t *testing.T) {
	t.Parallel()
	// Развертываем БД MySQL
	dbOpts := createDbOpts(t, dbDirStage)
	defer terraform.Destroy(t, dbOpts)
	terraform.InitAndApply(t, dbOpts)
	// Развертываем hello-world-app
	helloOpts := createHelloOpts(dbOpts, appDirStage)
	defer terraform.Destroy(t, helloOpts)
	terraform.InitAndApply(t, helloOpts)
	// Проверяем, работает ли hello-world-app
	validateHelloApp(t, helloOpts)
}
```

У теста следующая структура: развернуть mysql и hello-world-app, проверить приложение, удалить hello-world-app (выполняется в конце благодаря defer) и в завершение удалить mysql (выполняется в конце благодаря defer). Методы createDbOpts, createHelloOpts и validateHelloApp пока не существуют, поэтому реализуем их по очереди. Начнем с метода createDbOpts:

```
func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
	uniqueId := random.UniqueId()
	return &terraform.Options{
		TerraformDir: terraformDir,
		Vars: map[string]interface{}{
			"db_name": fmt.Sprintf("test%s", uniqueId),
			"db_password": "password",
		},
	}
}
```

Пока ничего нового: код передает методу terraform.Options заданную папку и устанавливает переменные db_name и db_password.

Дальше нужно разобраться с тем, где модуль mysql будет хранить свое состояние. До сих пор конфигурация backend содержала значения, прописанные вручную:

```
terraform {
  backend "s3" {
    # Поменяйте на имя своего бакета!
    bucket         = "terraform-up-and-running-state"
    key            = "stage/data-stores/mysql/terraform.tfstate"
    region         = "us-east-2"
    # Замените именем своей таблицы DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Эти значения создают большую проблему, потому что, если оставить их как есть, будет перезаписан реальный файл состояния среды финального тестирования! Одним из вариантов — использование рабочих областей (как обсуждалось в подразделе «Изоляция через рабочие области» на с. 114), но для этого все равно нужен доступ к бакету S3, принадлежащему учетной записи среду финального тестирования, тогда как все ваши тесты должны выполняться от имени совершенно отдельного пользователя AWS. Вместо этого лучше использовать частичную конфигурацию, как было описано в разделе «Ограничения хранилищ Terraform» на с. 110. Вынесите всю конфигурацию backend во внешний файл, такой как backend.hcl:

```
bucket         = "terraform-up-and-running-state"
key            = "stage/data-stores/mysql/terraform.tfstate"
region         = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt        = true
```

Таким образом, конфигурация backend в файле live/stage/data-stores/mysql/main.tf остается пустой:

```
terraform {
  backend "s3" {
  }
}
```

При развертывании модуля mysql в настоящей среде финального тестирования нужно указать аргумент -backend-config, чтобы Terraform использовал конфигурацию backend из файла backend.hcl:

```
$ terraform init -backend-config=backend.hcl
```

При выполнении тестов для модуля mysql вы можете заставить Terratest передать тестовые значения, используя параметр BackendConfig для terraform.Options:

```
func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
    uniqueId := random.UniqueId()
    bucketForTesting := "YOUR_S3_BUCKET_FOR_TESTING"
                                                bucketRegionForTesting := "YOUR_S3_BUCKET_REGION_FOR_TESTING"
                dbStateKey := fmt.Sprintf("%s/%s/terraform.tfstate", t.Name(), uniqueId)
    return &terraform.Options{
        TerraformDir: terraformDir,
        Vars: map[string]interface{}{
            "db_name": fmt.Sprintf("test%s", uniqueId),
            "db_password": "password",
        },
        BackendConfig: map[string]interface{}{
            "bucket":      bucketForTesting,
            "region":      bucketRegionForTesting,
            "key":         dbStateKey,
            "encrypt":     true,
        },
    }
}
```

Вы должны указать собственные значения для переменных bucketForTesting и bucketRegionForTesting. В качестве хранилища в тестовой учетной записи AWS можно создать один бакет S3, так как параметр key (путь внутри бакета) содержит идентификатор uniqueId, который должен быть достаточно уникальным, чтобы все тесты имели разные значения.

Далее следует внести некоторые изменения в модуль hello-world-app в среде финального тестирования. Откройте файл live/stage/services/hello-world-app/variables.tf и сделайте доступными переменные db_remote_state_bucket, db_remote_state_key и environment:

```
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the database's remote state"
  type        = string
}
variable "db_remote_state_key" {
  description = "The path for the database's remote state in S3"
  type        = string
```

```
}
variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
  default     = "stage"
}
```

Передайте эти значения модулю hello-world-app в файле live/stage/services/hello-world-app/main.tf:

```
module "hello_world_app" {
  source = "../../../../modules/services/hello-world-app"
  server_text            = "Hello, World"
  environment            = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key    = var.db_remote_state_key
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

Теперь вы можете реализовать метод createHelloOpts:

```
func createHelloOpts(
        dbOpts *terraform.Options,
        terraformDir string) *terraform.Options {
        return &terraform.Options{
                TerraformDir: terraformDir,
                Vars: map[string]interface{}{
                                                        "db_remote_state_bucket": dbOpts.BackendConfig["bucket"],
                        "db_remote_state_key": dbOpts.BackendConfig["key"],
                        "environment": dbOpts.Vars["db_name"],
                },
        }
}
```

Обратите внимание, что переменным db_remote_state_bucket и db_remote_state_key присвоены значения из BackendConfig для модуля mysql. Благодаря этому модуль hello-world-app читает именно то состояние, которое только что было записано модулем mysql. Переменная environment равна db_name, чтобы все ресурсы распределялись по пространствам имен одним и тем же образом.

Наконец-то вы можете реализовать метод validateHelloApp:

```
func validateHelloApp(t *testing.T, helloOpts *terraform.Options) {
                albDnsName := terraform.OutputRequired(t, helloOpts,
"alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)
        maxRetries := 10
        timeBetweenRetries := 10 * time.Second
        http_helper.HttpGetWithRetryWithCustomValidation(
            t,
            url,
            &tls.Config{},
            maxRetries,
            timeBetweenRetries,
            func(status int, body string) bool {
                return status == 200 &&
                    strings.Contains(body, "Hello, World")
            },
        )
    }
```

Как и наши модульные тесты, этот метод использует пакет http_helper, только на этот раз мы вызываем из него http_helper.HttpGetWithRetryWithCustomValidation, что позволяет нам указать наши собственные правила проверки кода состояния и тела HTTP-ответа. Это необходимо для проверки наличия в HTTP-ответе строки Hello, World, а не для точного сопоставления строк, так как ответ, который возвращает скрипт пользовательских данных внутри модуля hello-world-app, содержит и другой текст.

Теперь запустите интеграционный тест и проверьте, работает ли он:

```
$ go test -v -timeout 30m -run "TestHelloWorldAppStage"
(...)
PASS
ok  terraform-up-and-running  795.63s
```

Отлично! Теперь у вас есть интеграционный тест, с помощью которого можно убедиться в корректной совместной работе нескольких ваших модулей. Он получился более сложным, чем модульный тест, и его выполнение длится вдвое дольше (10–15 минут вместо 4–5). Сделать его быстрее не так уж просто — узкое место здесь зависит от того, как долго AWS будет развертывать RDS, ASG, ALB и т. д. Но в определенных обстоятельствах работу теста можно сократить с помощью стадий тестирования.

Стадии тестирования

Если взглянуть на код вашего интеграционного теста, можно заметить, что он состоит из нескольких отдельных стадий.

1. Запустить terraformapply для модуля mysql.
2. Запустить terraformapply для модуля hello-world-app.
3. Выполнить проверку и убедиться в том, что все работает.
4. Выполнить terraformdestroy для модуля hello-world-app.
5. Выполнить terraformdestroy для модуля mysql.

Если вы запускаете эти тесты в среде CI, нужно выполнить каждую стадию, от начала до конца. Но если вы используете их в локальной среде для разработки и вместе с этим итеративно вносите изменения в свой код, все стадии необязательны. Например, если вы редактируете только модуль hello-world-app, повторный запуск всего теста после каждой фиксации влечет за собой развертывание и удаление модуля mysql, хотя ваши изменения его никак не касаются. Это добавляет к времени работы теста 5–10 минут без какой-либо необходимости.

В идеале рабочий процесс должен выглядеть определенным образом.

1. Запустить terraformapply для модуля mysql.
2. Запустить terraformapply для модуля hello-world-app.
3. Переход к итеративной разработке:

а) внести изменение в модуль hello-world-app;

б) повторно выполнить terraformapply для модуля hello-world-app, чтобы развернуть ваши обновления;

в) проверить и убедиться в том, что все работает;

г) если все работает, перейти к следующему шагу, если нет — вернуться к шагу 3«а».

4. Выполнить terraformdestroy для модуля hello-world-app.
5. Выполнить terraformdestroy для модуля mysql.

Возможность быстро выполнить внутренний цикл в пункте 3 — ключ к быстрой итеративной разработке с использованием Terraform. Для этого нужно разбить код своего теста на стадии, после чего вы сможете выбирать, какие из них выполнять, а какие можно пропустить.

Terratest имеет встроенную поддержку этой стратегии в виде пакета test_structure (чтобы его добавить, не забудьте выполнить depensure). Суть в том, что каждая стадия вашего теста заворачивается в функцию с именем; затем вы сможете заставить Terratest пропустить некоторые из этих имен, используя переменные среды. Каждая стадия проверки сохраняет тестовые данные на диск, чтобы их можно было прочитать во время последующих выполнений. Попробуем это на примере test/hello_world_integration_test.go. Сначала набросаем каркас теста, а затем наполним его внутренние методы:

```
func TestHelloWorldAppStageWithStages(t *testing.T) {
    t.Parallel()
    // Сохраняем функцию в переменную с коротким именем,
    // просто чтобы примеры с кодом было легче уместить
    // на страницах этой книги.
    stage := test_structure.RunTestStage
    // Развертываем БД MySQL
    defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
    stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })
    // Развертываем приложение hello-world-app
    defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
     stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage)
})
    // Проверяем, работает ли hello-world-app
    stage(t, "validate_app", func() { validateApp(t, appDirStage) })
}
```

Структура та же, что и прежде: развернуть mysql и hello-world-app, проверить приложение, удалить hello-world-app (выполняется в конце благодаря defer) и mysql (выполняется в конце с помощью defer). Разница лишь в том, что теперь каждая стадия обернута в test_structure.RunTestStage. Метод RunTestStage принимает три аргумента.

•t. Первым аргументом выступает значение t, которое Go передает всем автоматическим тестам. С его помощью можно управлять состоянием теста. Например, вы можете его провалить, вызвав t.Fail().

• *Имя стадии*. Второй аргумент позволяет задать имя этой стадии тестирования. Вскоре вы увидите пример того, как с помощью этого имени можно пропускать отдельные стадии.

•*Код для выполнения*. Третий аргумент — это код, который нужно выполнить на данной стадии тестирования. Это может быть любая функция.

Теперь реализуем функции для каждой стадии тестирования. Начнем с deployDb:

```
func deployDb(t *testing.T, dbDir string) {
    dbOpts := createDbOpts(t, dbDir)
       // Сохраняем данные на диск, чтобы в процессе других стадий
теста,
    // запущенных позже, можно было их прочитать
    test_structure.SaveTerraformOptions(t, dbDir, dbOpts)
    terraform.InitAndApply(t, dbOpts)
}
```

Как и прежде, чтобы развернуть mysql, код вызывает createDbOpts и terraform.InitAndApply. Единственное изменение лишь в том, что теперь между этими двумя шагами находится вызов test_structure.SaveTerraformOptions, который записывает содержимое dbOpts на диск, чтобы позже его могли прочитать другие стадии тестирования. Например, вот реализация функции teardownDb:

```
func teardownDb(t *testing.T, dbDir string) {
    dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
    defer terraform.Destroy(t, dbOpts)
}
```

Эта функция вызывает test_structure.LoadTerraformOptions, чтобы загрузить с диска содержимое dbOpts, которое было записано ранее функцией deployDb. Причина, по которой эти данные передаются через диск, а не в оперативной памяти, связана с тем, что каждая стадия может запускаться самостоятельно — то есть в отдельном процессе. Как вы увидите позже в этой главе, при первых нескольких запусках gotest имеет смысл выполнить deployDb, но пропустить teardownDb, а затем, при последующих запусках, сделать наоборот. Чтобы во время всех этих запусков использовалась одна и та же база данных, информацию о ней следует хранить на диске.

Теперь реализуем функцию deployHelloApp:

```
func deployApp(t *testing.T, dbDir string, helloAppDir string) {
    dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
    helloOpts := createHelloOpts(dbOpts, helloAppDir)
      // Сохраняем данные на диск, чтобы в процессе других стадий теста,
    // запущенных позже, можно было их прочитать
    test_structure.SaveTerraformOptions(t, helloAppDir, helloOpts)
    terraform.InitAndApply(t, helloOpts)
}
```

Этот код повторно использует ранее определенную функцию createHelloOpts и вызывает для нее terraform.InitAndApply. И снова все поведение заключается в запуске методов test_structure.SaveTerraformOptions и test_structure.SaveTerraformOptions для загрузки с диска dbOpts и сохранения на диск helloOpts соответственно. Вы уже догадываетесь, как будет выглядеть реализация метода teardownApp:

```
func teardownApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
    defer terraform.Destroy(t, helloOpts)
}
```

А вот реализация метода validateApp:

```
func validateApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
    validateHelloApp(t, helloOpts)
}
```

Таким образом, данный код идентичен оригинальному интеграционному тесту, только теперь каждая стадия завернута в вызов test_structure.RunTestStage, и еще вам нужно приложить немного усилий для сохранения и чтения данных с диска. Эти простые изменения открывают перед вами важную возможность: заставить Terratest пропустить любую стадию с именем foo, установив переменную среды SKIP_foo=true. Разберем типичный процесс написания кода, чтобы увидеть, как это работает.

Для начала нужно запустить тест, пропустив при этом обе стадии очистки, чтобы по окончании тестирования модули mysql и hello-world-app оставались развернутыми. Поскольку эти стадии называются teardown_db и teardown_app, нужно установить переменные среды SKIP_teardown_db и SKIP_teardown_app соответственно. Так Terratest будет знать, что их нужно пропустить:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)
The 'SKIP_deploy_db' environment variable is not set, so executing stage 'deploy_db'.
(...)
The 'deploy_app' environment variable is not set, so executing stage 'deploy_db'.
(...)
The 'validate_app' environment variable is not set,
so executing stage 'deploy_db'.
(...)
The 'teardown_app' environment variable is set, so skipping stage 'deploy_db'.
(...)
The 'teardown_db' environment variable is set, so skipping stage 'deploy_db'.
(...)
PASS
ok  terraform-up-and-running  423.650s
```

Теперь вы можете приступить к последовательному редактированию модуля hello-world-app, повторно запуская тесты при каждом изменении. Но на этот раз сделайте так, чтобы, помимо очистки, пропускалась также стадия развертывания модуля mysql (так как mysql по-прежнему выполняется). Таким образом, будет выполнена только команда deployapp и проведена проверка модуля hello-world-app:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  SKIP_deploy_db=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)
The 'SKIP_deploy_db' environment variable is set, so skipping stage 'deploy_db'.
(...)
The 'deploy_app' environment variable is not set, so executing stage 'deploy_db'.
(...)
The 'validate_app' environment variable is not set, so executing stage 'deploy_db'.
(...)
The 'teardown_app' environment variable is set, so skipping stage 'deploy_db'.
(...)
The 'teardown_db' environment variable is set, so skipping stage 'deploy_db'.
(...)
PASS
ok  terraform-up-and-running  13.824s
```

Обратите внимание на то, как быстро теперь работает каждый из этих тестов: вместо 10–15 минут каждое новое изменение занимает 10–60 секунд (в зависимости от того, что поменялось). Учитывая, что в процессе разработки эти стадии, скорее всего, будут выполняться десятки или даже сотни раз, вы можете сэкономить уйму времени.

Когда после всех изменений модуль hello-world-app начнет работать так, как вы того ожидали, самое время очистить все ресурсы. Запустите тесты еще раз, но теперь пропустите стадии развертывания и проверки, чтобы выполнялась только очистка:

```
$ SKIP_deploy_db=true \
  SKIP_deploy_app=true \
  SKIP_validate_app=true \
```

```
go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)
The 'SKIP_deploy_db' environment variable is set, so skipping stage 'deploy_db'.
(...)
The 'SKIP_deploy_app' environment variable is set, so skipping stage 'deploy_app'.
(...)
The 'SKIP_validate_app' environment variable is set, so skipping stage 'validate_app'.
(...)
The 'SKIP_teardown_app' environment variable is not set, so executing stage 'teardown_app'.
(...)
The 'SKIP_teardown_db' environment variable is not set, so executing stage 'teardown_db'.
(...)
PASS
ok  terraform-up-and-running  340.02s
```

Использование стадий тестирования позволяет быстро получать обратную связь от автоматических тестов, что существенно ускоряет и повышает качество итеративной разработки. Это не повлияет на скорость выполнения тестов в среде CI, однако воздействие на среду разработки будет огромным.

Повторение попыток

Начав регулярно выполнять автоматические тесты для своего инфраструктурного кода, вы столкнетесь с их непредсказуемостью. Иногда они будут проваливаться по причинам временного характера: например, сервер EC2 может не запуститься, в Terraform может проявиться ошибка отложенной согласованности или вам не удастся установить TLS-соединение с S3. Мир инфраструктуры полон беспорядка, поэтому вы должны быть готовы к периодическим сбоям в своих тестах, обрабатывая их соответствующим образом.

Чтобы сделать тесты чуть более устойчивыми, для известных ошибок можно предусмотреть повторение попыток. Например, в процессе написания этой книги я время от времени получаю такого рода ошибку (особенно при параллельном запуске множества тестов):

```
* error loading the remote state: RequestError: send request failed
Post https://xxx.amazonaws.com/: dial tcp xx.xx.xx.xx:443:
```

connect: connection refused

Чтобы ваши тесты лучше справлялись с подобным, вы можете включить в Terratest повторные попытки, используя аргументы MaxRetries, TimeBetweenRetries и RetryableTerraformErrors метода terraform.Options:

```
func createHelloOpts(
    dbOpts *terraform.Options,
    terraformDir string) *terraform.Options {
    return &terraform.Options{
        TerraformDir: terraformDir,
        Vars: map[string]interface{}{
                                                "db_remote_state_bucket":
dbOpts.BackendConfig["bucket"],
            "db_remote_state_key": dbOpts.BackendConfig["key"],
            "environment": dbOpts.Vars["db_name"],
        },
        // Повторяем не более трех раз с интервалом пять секунд
        // между попытками, для известных ошибок
        MaxRetries: 3,
        TimeBetweenRetries: 5 * time.Second,
        RetryableTerraformErrors: map[string]string{
            "RequestError: send request failed": "Throttling issue?",
        },
    }
}
```

Аргументу RetryableTerraformErrors можно передать ассоциативный массив с известными ошибками, которые требуют повторения попытки. В качестве ключей выступают сообщения об ошибках, которые нужно искать в журнальных записях (здесь можно использовать регулярные выражения), а значениями служат дополнительные сведения, которые записываются в журнал, когда Terratest находит одну из ошибок и инициирует повторную попытку. Теперь, когда код теста сталкивается с указанной вами ошибкой, в журнале должно появляться сообщение и по прошествии TimeBetweenRetries ваша команда выполнится еще раз:

```
$ go test -v -timeout 30m
(...)
Running command terraform with args [apply -input=false -lock=false -auto-
approve]
(...)
* error loading the remote state: RequestError: send request failed
Post https://s3.amazonaws.com/: dial tcp 11.22.33.44:443:
```

connect: connection refused
(...)
'terraform [apply]' failed with the error 'exit status code 1'
but this error was expected and warrants a retry. Further details:
Intermittent error, possibly due to throttling?
(...)
Running command terraform with args [apply -input=false -lock=false -auto-approve]

Сквозные тесты

Разобравшись с модульными и интеграционными тестами, можно приступить к тестированию последнего типа, которым вы можете воспользоваться, — *сквозному*. Если вернуться к нашему примеру с Ruby, сквозные тесты могут предполагать развертывание веб-сервера вместе с любыми хранилищами данных, которые ему нужны, и проверку его работы из веб-браузера с помощью такого инструмента, как Selenium. Похоже выглядят и сквозные тесты для инфраструктуры Terraform: сначала мы развертываем весь код в среду, которая симулирует промышленные условия, а затем проверяем его с точки зрения конечного пользователя.

Для написания сквозных тестов можно использовать ту же стратегию, что и для интеграционных, — сначала создается несколько десятков стадий для выполнения terraformapply, проводятся некоторые проверки, а затем все очищается с помощью terraformdestroy. Но на практике такой подход применяют редко. Это связано с *пирамидой тестирования*, которую вы можете видеть на рис. 7.1.

Рис. 7.1. Пирамида тестирования

Суть пирамиды тестирования в том, что в общем случае у вас должно быть много модульных тестов (основание пирамиды), меньше

интеграционных (середина пирамиды) и еще меньше сквозных (вершина пирамиды). Это вызвано тем, что при движении вверх по пирамиде возрастают сложность, хрупкость и время выполнения тестов.

Из этого следует *ключевой вывод о тестировании № 5*: чем меньше модули, тем легче и быстрее их тестировать.

В предыдущих разделах вы уже видели, что даже для тестирования относительно простого модуля hello-world-app требуется довольно много усилий, связанных с пространствами имен, внедрением зависимостей, повторными попытками, обработкой ошибок и разделением на стадии. С развитием инфраструктуры этот процесс только усложняется. Поэтому как можно большую часть тестирования следует выполнять максимально близко к основанию пирамиды, поскольку на этом уровне можно получить самую быструю и надежную обратную связь.

К моменту, когда вы добираетесь до вершины пирамиды, выполнение тестов для развертывания сложной инфраструктуры с нуля теряет всякий смысл. Этому есть две причины.

•*Слишком медленно*. Развертывание целой инфраструктуры с нуля с последующим ее удалением занимает очень много времени: порядка нескольких часов. Тестовые наборы, которые выполняются так долго, приносят относительно мало пользы просто потому, что обратная связь слишком замедляется. Такие тесты обычно запускаются только на ночь. Это означает, что утром вы получите отчет о проваленном тесте, потратите какое-то время на исследование проблемы, внесете исправление и узнаете о результате только на следующий день. Таким образом, вы можете сделать примерно одно исправление в сутки. В подобного рода ситуациях разработчики часто начинают винить в проваленных тестах кого-то другого, убеждать руководство в том, что код нужно развернуть, несмотря на выявленные проблемы, и, в конце концов, вовсе игнорировать непройденные тесты.

• *Высокая хрупкость*. Как уже упоминалось в предыдущих разделах, мир инфраструктуры беспорядочен. Чем больше ресурсов вы развертываете, тем выше вероятность возникновения нерегулярных и непредсказуемых проблем. Представьте, к примеру, что у какого-то ресурса (скажем, сервера EC2) есть один шанс из тысячи (0,1 %) выйти из строя из-за нерегулярной ошибки (в мире DevOps этот показатель будет выше). Это означает, что вероятность развертывания тестом этого ресурса без каких-либо нерегулярных ошибок равна 99,9 %. Но что насчет теста, который развертывает два ресурса? Для его успешного прохождения оба ресурса должны быть развернуты без нерегулярных ошибок. Чтобы просчитать эти шансы, вероятности нужно умножить: 99,9 × 99,9 = 99,8 %. В случае с тремя

ресурсами шансы равны 99,9 × 99,9 × 99,9 = 99,7 %. С *N* ресурсами формула выглядит как 99,9 %N.

Теперь рассмотрим разные виды автоматического тестирования. Если ваш модульный тест, предназначенный для одного модуля, развертывает 20 ресурсов, шансы на успех равны 99,9 %20 = 98,0 %. Значит, два теста из 100 провалятся; обычно их можно сделать более надежными, если добавить несколько повторных попыток. Но представьте, что ваш интеграционный тест с тремя модулями развертывает 60 ресурсов. Теперь шансы на успех равны 99,9 %60 = 94,1 %. Опять же, добавив достаточное количество повторных попыток, вы можете сделать такой тест вполне стабильным для того, чтобы он приносил какую-то пользу. Но если у вас есть сквозной тест из 30 модулей или примерно 600 ресурсов? Шансы на успех падают до 99,9 %600 = 54,9 %. Это означает, что почти каждая вторая проверка будет проваливаться по временным причинам!

С некоторыми из этих ошибок можно справиться с помощью повторных попыток, но это быстро превратится в вечную игру в догонялки. Вы добавляете повторную попытку на случай истечения времени ожидания TLS-соединения и тут же сталкиваетесь с простоем в работе APT-репозитория в своем шаблоне для Packer. Вы добавляете повторные попытки для сборки Packer, и тут ваша сборка прерывается из-за ошибки отложенной согласованности в Terraform. Кое-как справившись с этой проблемой, вы видите, что сборка «падает» в результате перебоев в работе GitHub. И, поскольку сквозные тесты выполняются очень долго, на исправления у вас есть лишь 1–2 попытки в день.

В реальности очень немногие компании со сложной инфраструктурой выполняют сквозные тесты, которые развертывают все *с нуля*. Более распространенная стратегия сквозного тестирования выглядит так.

1. Вы развертываете окружение под названием test, максимально приближенное к промышленным условиям, и делаете это единожды. Оно остается на постоянной основе.

2. Каждый раз, когда вы вносите изменение в свою инфраструктуру, сквозной тест делает следующее:

а) применяет изменение инфраструктуры к тестовой среде;

б) выполняет проверку тестовой среды (например, с помощью Selenium для проверки вашего кода с точки зрения конечного пользователя), чтобы убедиться в том, что все работает.

Сместив свою стратегию сквозного тестирования в сторону инкрементальных изменений, вы уменьшите количество ресурсов, которые развертываются на время проверки, с нескольких сотен до лишь горстки, благодаря чему ваши тесты станут более быстрыми и менее хрупкими.

Кроме того, такой подход к сквозному тестированию больше похож на то, как эти изменения будут развертываться в промышленных условиях. Ваше промышленное окружение не создается заново при каждом обновлении. Изменения применяются последовательно, поэтому такой стиль сквозных тестов имеет огромное преимущество: вы можете убедиться не только в корректности работы своей инфраструктуры, но и в том, что *процесс ее развертывания* работает как следует.

Например, очень важно проверить, можно ли выкатывать обновления инфраструктуры без простоя. Созданный вами модуль asg-rolling-deploy должен поддерживать скользящие развертывания с нулевым временем простоя, но как это проверить? Специально для этого добавим автоматический тест.

Вам будет достаточно внести лишь несколько исправлений в тест test/hello_world_integration_test.go, который вы только что написали, так как внутри hello-world-app используется модуль asg-rolling-deploy. Для начала нужно сделать доступной переменную server_text в файле live/stage/services/hello-world-app/variables.tf:

```
variable "server_text" {
  description = "The text the web server should return"
  default     = "Hello, World"
  type        = string
}
```

Передайте эту переменную модулю hello-world-app в файле live/stage/services/hello-world-app/main.tf:

```
module "hello_world_app" {
  source = "../../../../modules/services/hello-world-app"
  server_text = var.server_text
  environment            = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key    = var.db_remote_state_key
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

В скрипте пользовательских данных модуля hello-world-app используется аргумент server_text, поэтому при каждом изменении этой переменной происходит развертывание с (как мы надеемся) нулевым временем простоя. Проверим это, добавив в конец файла

test/hello_world_integration_test.go, сразу после validate_app, еще одну стадию тестирования под названием redeploy_app:

```
func TestHelloWorldAppStageWithStages(t *testing.T) {
    t.Parallel()
    // Сохраняем функцию в переменную с коротким именем,
    // просто чтобы примеры с кодом было легче уместить
    // на страницах этой книги.
    stage := test_structure.RunTestStage
    // Развертываем БД MySQL
    defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
    stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })
    // Развертываем приложение hello-world-app
    defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
     stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage)
})
    // Проверяем, работает ли hello-world-app
    stage(t, "validate_app", func() { validateApp(t, appDirStage) })
    // Развертываем hello-world-app заново
    stage(t, "redeploy_app", func() { redeployApp(t, appDirStage) })
}
```

Теперь реализуйте новый метод redeployApp:

```
func redeployApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
                albDnsName := terraform.OutputRequired(t, helloOpts,
"alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)
      // Начинаем проверять раз в секунду, возвращает ли приложение
200 OK
    stopChecking := make(chan bool, 1)
    waitGroup, _ := http_helper.ContinuouslyCheckUrl(
        t,
        url,
        stopChecking,
        1*time.Second,
    )
    // Обновляем текст сервера и развертываем заново
    newServerText := "Hello, World, v2!"
    helloOpts.Vars["server_text"] = newServerText
    terraform.Apply(t, helloOpts)
    // Проверяем, развернута ли новая версия
```

```
        maxRetries := 10
        timeBetweenRetries := 10 * time.Second
        http_helper.HttpGetWithRetryWithCustomValidation(
            t,
            url,
            &tls.Config{},
            maxRetries,
            timeBetweenRetries,
            func(status int, body string) bool {
                return status == 200 &&
                    strings.Contains(body, newServerText)
            },
        )
        // Завершаем проверку
        stopChecking <- true
        waitGroup.Wait()
    }
```

Этот метод делает следующее.

1. Использует метод http_helper.ContinuouslyCheckUrl из состава Terratest, чтобы запустить в фоне горутину (легковесный поток, управляемый средой выполнения Go), которая раз в секунду делает HTTP-запрос типа GET по URL-адресу ALB и проваливает тест при получении любого ответа, кроме 200OK.
2. Обновляет переменную server_text, которая выполняет terraformapply, чтобы инициировать скользящее развертывание.
3. По завершении развертывания убеждается в том, что сервер возвращает в виде ответа новое значение server_text.
4. Останавливает горутину, которая непрерывно проверяет URL-адрес ALB.

Если после запуска этот тест дойдет до заключительной стадии, вы увидите, как непрерывные HTTP-вызовы типа GET к URL-адресу ALB начнут чередоваться с выводом команды terraformapply нашего скользящего развертывания:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)
Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
```

http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
(...)
Running command terraform with args
[apply -input=false -lock=false -auto-approve]
(...)
Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
(...)
PASS
Ok terraform-up-and-running 551.04s

Успешное прохождение этого теста означает, что модули hello-world-app и asg-rolling-deploy, как и обещалось, способны выполнять скользящие развертывания с нулевым временем простоя! Каждый раз, когда ваш сквозной тест вносит инкрементальное изменение в вашу тестовую среду, вы можете использовать эту стратегию, чтобы убедиться в том, что развертывание не создает перебоев в работе.

Другие подходы к тестированию

Большая часть этой главы посвящена автоматическим тестам на основе Terratest, но существуют два других подхода к тестированию, которые не помешает иметь в своем арсенале:

• статический анализ;
• тестирование свойств.

По аналогии с тем, как разные виды автоматических тестов (модульные, интеграционные, сквозные) нацелены на разные виды ошибок, каждый из этих подходов помогает выявить разные проблемы, поэтому для получения максимальных результатов их лучше использовать совместно. По очереди рассмотрим эти новые категории.

Статический анализ

Существует несколько инструментов, которые умеют анализировать код Terraform без его запуска. Расскажу про основные.

•terraformvalidate. Эта команда встроена в Terraform, с ее помощью в коде можно искать синтаксические ошибки и опечатки (немного напоминает компилятор).

• *tflint* (https://github.com/wata727/tflint). «Линтер» для Terraform, который может просканировать ваш код в поисках распространенных ошибок и потенциальных проблем на основе встроенных правил.

• *HashiCorp Sentinel* (https://www.hashicorp.com/sentinel). Фреймворк типа «политика как код», который позволяет следить за соблюдением правил в разных продуктах HashiCorp. Например, вы можете создать политику, согласно которой вам нельзя будет создавать в своем коде правила группы безопасности, которые разрешают исходящие соединения с 0.0.0.0/0. На момент написания этой книги Sentinel можно использовать только в связке с продуктами HashiCorp Enterprise, включая Terraform Enterprise.

Тестирование свойств

Существует целый ряд средств тестирования для проверки определенных «свойств» вашей инфраструктуры:

• kitchen-terraform (https://github.com/newcontext-oss/kitchen-terraform);
• rspec-terraform (https://github.com/bsnape/rspec-terraform);
• serverspec (https://serverspec.org/);
• inspec (https://www.inspec.io/);
• goss (https://github.com/aelsabbahy/goss).

Большинство этих инструментов предоставляют *предметно-ориентированный язык* (domain-specific language, DSL) для проверки развернутой вами инфраструктуры на соответствие какого-то рода спецификации. Например, если вы тестируете модуль Terraform, который развертывает сервер EC2, можно использовать следующий код для inspec, чтобы проверить, имеет ли этот сервер подходящие права для доступа к определенным файлам, установлены ли у него конкретные зависимости и прослушивает ли он заданный порт:

```
describe file('/etc/myapp.conf') do
  it { should exist }
  its('mode') { should cmp 0644 }
end
describe apache_conf do
  its('Listen') { should cmp 8080 }
end
describe port(8080) do
```

```
  it { should be_listening }
end
```

Преимущество этих инструментов в том, что языки DSL, помимо своей лаконичности и простоты в использовании, предлагают эффективный декларативный способ проверки большого количества свойств инфраструктуры. Это отлично подходит для соблюдения списка требований, в особенности тех, которые касаются стандартов и правовых норм (вроде стандарта PCI или закона HIPAA). Недостатком этих инструментов является то, что даже после успешной проверки всех свойств ваша инфраструктура может не работать! Для сравнения, чтобы протестировать эти же свойства с помощью Terratest, вам пришлось бы выполнять HTTP-запросы к серверу и проверять, возвращает ли тот ожидаемые ответы.

Резюме

В мире инфраструктуры все постоянно меняется. Не стоят на месте и такие инструменты, как Terraform, Packer, Docker, Kubernetes, AWS, Google Cloud, Azure и др. Это означает, что инфраструктурный код очень быстро теряет свою актуальность. Или, говоря другими словами: *инфраструктурный код без автоматических тестов можно считать неисправным.*

Это одновременно и афоризм, и буквальное утверждение. Каждый раз, когда я берусь за написание инфраструктурного кода, при переходе к автоматическим тестам неминуемо всплывает множество неочевидных проблем. И неважно, сколько усилий я перед этим потратил на рефакторинг, ручное тестирование и разбор кода. Когда вы находите время для автоматизации процесса тестирования, почти всегда происходит какая-то магия и вам удается выявить проблемы, которые вы бы сами никогда не нашли (а вот ваши клиенты — легко). Причем они находятся не только при первом добавлении автоматических тестов, но и с каждой последующей фиксацией кода, особенно по мере изменения мира DevOps вокруг вас.

Автоматические тесты, которыми я оснастил свой инфраструктурный код, нашли ошибки не только в моей инфраструктуре, но и в используемых мной инструментах, включая нетривиальные проблемы в Terraform, Packer, Elasticsearch, Kafka, AWS и т. д. Как показывает эта глава, написание автоматических тестов — занятие непростое, требующее значительных усилий. Еще больше усилий уходит на их поддержку и обеспечение надежности за счет достаточного количества повторных попыток. А самое сложное — поддерживать в порядке тестовую среду, чтобы контролировать свои расходы. Но оно того стоит.

Например, когда я работаю над модулем для развертывания хранилища данных, после каждой фиксации кода в репозитории мои тесты создают десяток копий этого хранилища с разными конфигурациями, записывают и считывают данные, после чего удаляют все это. Каждое успешное прохождение этих тестов дает мне железную уверенность в том, что мой код по-прежнему работает. По меньшей мере они позволяют мне лучше спать. Те часы, которые я потратил на логику повторных попыток и отложенную согласованность, компенсируются тем, что мне не приходится подниматься посреди ночи из-за перебоев в работе.

У этой книги тоже есть тесты!

Для всех примеров кода в этой книге тоже предусмотрены тесты. Весь представленный здесь код и тесты к нему можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

В этой главе вы познакомились с основными принципами тестирования кода Terraform, включая ключевые моменты.

•*Тестирование кода Terraform не может проходить локально*. В связи с этим для ручного тестирования приходится развертывать настоящие ресурсы в одной или нескольких изолированных средах.

• *Чтобы держать расходы под контролем, регулярно чистите свои изолированные среды*. В противном случае ваши ресурсы станут неуправляемыми и расходы выйдут из-под контроля.

• *Вы не можете проводить чистое тестирование кода Terraform*. Поэтому все автоматическое тестирование должно происходить с помощью кода, который развертывает реальные ресурсы в одной или нескольких изолированных средах.

• *Все ваши ресурсы должны быть разделены по пространствам имен*. Это гарантирует отсутствие конфликтов между несколькими тестами, запущенными параллельно.

•*Чем меньше модули, тем легче и быстрее их тестировать*. Это один из ключевых выводов главы 6, но он стоит того, чтобы повторить его еще раз: чем меньше модуль, тем легче его создавать, поддерживать, использовать и тестировать.

В главе 8 вы узнаете, как внедрить код Terraform и автоматические тесты в рабочий процесс вашей команды. Среди прочего мы рассмотрим

управление окружениями, конфигурацию процессов непрерывной интеграции и непрерывного развертывания и т. д.

55 AWS не взимает плату за дополнительные учетные записи, и, если вы используете AWS Organizations, у вас есть возможность создавать «дочерние» учетные записи, все расходы которых записываются на единый «главный» счет.

56 В отдельных случаях точки входа, которые использует Terraform, можно переопределить. Например, вы можете переопределить точку входа, с помощью которой Terraform взаимодействует с Amazon S3, и заменить ее mock-объектом, реализующим API S3. Это нормальное решение для небольшого количества точек входа, но код Terraform обычно выполняет сотни разных API-вызовов к исходному провайдеру и создавать mock-объекты для каждого из них было бы непрактично. Более того, если вы пошли на создание этих mock-объектов, это вовсе не означает, что итоговый модульный тест даст вам достаточно уверенности в корректной работе вашего кода. Если вы создадите mock-объекты для точек входа ASG и ALB, команда terraform apply может завершиться успешно. Но говорит ли это о том, что наш код способен развернуть рабочее приложение поверх этой инфраструктуры?

57 В будущих версиях Terratest, скорее всего, начнет использовать go mod для управления зависимостями. На момент написания этих строк go mod имеет лишь предварительную поддержку, но с выходом Go 1.13 этот компонент должен быть включен по умолчанию, поэтому он, вероятно, станет стандартным инструментом для работы с зависимостями в Go и заодно устранит необходимость в GOPATH. Подробнее об этом читайте по адресу blog.golang.org/using-go-modules.

8. Как использовать Terraform в команде

Чтение книги и разбор примеров кода — это то, чем обычно занимаются в одиночку. Но в реальности вы будете работать в команде, что создает целый ряд новых проблем. Возможно, придется убеждать свою команду в переходе на Terraform и другие инструменты IaC. Вы, вероятно, будете иметь дело с множеством людей, которые одновременно пытаются понять, использовать и модифицировать написанный вами код Terraform. Вам также нужно придумать, как интегрировать Terraform в ваш стек технологий и в рабочий процесс вашей компании.

В этой главе мы подробно рассмотрим ключевые процессы, которые вы должны наладить, чтобы ваша команда смогла использовать Terraform и IaC.

- Внедрение концепции «инфраструктура как код» внутри команды.
- Процесс развертывания кода приложений.
- Процесс развертывания инфраструктурного кода.

• Объединение отдельных элементов.

Последовательно пройдемся по каждой из этих тем.

Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу github.com/brikis98/terraform-up-and-running-code.

Внедрение концепции IaC внутри команды

Если ваша команда привыкла управлять всей инфраструктурой вручную, переход на концепцию IaC не ограничится простым знакомством с новым инструментом или технологией. Вам придется изменить культуру и рабочие процессы в вашей команде. Это задача не из простых, особенно в больших компаниях. Поскольку культура и процессы внутри каждой команды немного различаются, универсальных инструкций попросту не существует. Однако следующие советы будут полезны в большинстве ситуаций:

• убедите свое начальство;

• сделайте переход постепенным;

• дайте своей команде время на обучение.

Убедите свое начальство

Я часто наблюдал эту ситуацию во многих компаниях: разработчик открывает для себя Terraform, вдохновляется возможностями этой технологии, приходит на работу взволнованный, полный энтузиазма, рассказывает о Terraform всем вокруг… а начальник говорит «нет». Конечно, это разочаровывает и обескураживает разработчика. Почему эти преимущества видны не всем? Мы могли бы все автоматизировать! Мы могли бы избежать стольких ошибок! Как еще выплатить весь этот технический долг? Как все вокруг могут быть настолько слепыми?

Проблема в том, что за всеми преимуществами от внедрения средств IaC, таких как Terraform, этот разработчик не видит цену, которую придется заплатить. Приведем несколько примеров.

•*Разный уровень навыков*. Переход на IaC будет означать, что вашим системным администраторам придется тратить большую часть своего времени на написание больших объемов кода: модулей Terraform, тестов на Go, рецептов для Chef и т. д. Некоторые из них вполне счастливы

программировать днями напролет и с радостью воспримут изменения, но для других этот переход будет тяжелым. Многие операционные инженеры и сисадмины привыкли вносить изменения вручную, возможно, с периодическим использованием небольших скриптов; вместо этого им придется переквалифицироваться в полноценных разработчиков, что потребует изучения целого ряда новых навыков (или поиска новых работников).

• *Новые инструменты*. Разработчики программного обеспечения могут прикипеть к инструментам, которые они используют. Иногда такая приверженность становится почти религиозной. Каждый раз, когда вы вводите в обиход новый инструмент, одни разработчики будут в восторге от возможности изучить что-то новенькое, а другие предпочтут уже знакомые им технологии и с неохотой отнесутся к необходимости тратить большое количество времени и энергии на изучение новых языков и методик.

• *Изменение образа мышления*. Если члены вашей команды долгое время управляют инфраструктурой вручную, они уже привыкли вносить изменения *напрямую*: например, путем выполнения команд на сервере по SSH. Для перехода на IaC нужен сдвиг в образе мышления: все изменения вносятся *опосредованно* — сначала вы редактируете и сохраняете свой код, а затем ваши правки применяются каким-то автоматическим процессом. Такой уровень абстракции понравится не всем. При выполнении простых задач он будет казаться более медленным, чем прямой подход, особенно когда вы все еще изучаете новое средство IaC и не умеете использовать его эффективно.

• *Издержки упущенной выгоды*. Инвестируя время и ресурсы в один проект, вы косвенно обделяете другие проекты. Какие из них придется приостановить, чтобы мигрировать на IaC, и насколько они важны?

Некоторых членов вашей команды этот список только подзадорит. Но многие другие, включая ваше начальство, тяжело вздохнут. Изучение новых навыков, освоение новых инструментов и принятие нового образа мышления может пойти на пользу или провалиться, но одно известно наверняка: за все это придется заплатить. Переход на IaC — существенное вложение, которое, как и любое другое, имеет потенциальные плюсы и минусы.

Ваше начальство будет особенно обеспокоено издержками упущенной выгоды. Одна из ключевых обязанностей любого руководителя — следить за тем, чтобы команда работала над самыми приоритетными проектами. Когда вы приходите и начинаете восторженно рассказывать о Terraform, ваш начальник может думать про себя: «О нет, это похоже на огромное начинание, сколько времени это займет?» Это не означает, что ему

непонятны возможности Terraform. Просто время, потраченное на эту технологию, могло бы уйти на развертывание нового приложения, о котором уже несколько месяцев просит команда, занимающаяся поиском, или на подготовку к аудиту PCI (Payment Card Industry), или на расследование перебоев в работе, случившихся на прошлой неделе. Поэтому, если вы хотите убедить свое начальство в том, что ваша команда должна внедрить у себя IaC, продемонстрировать ценность этой технологии недостаточно. Вы должны показать, что польза, которую она принесет вашей команде, перевешивает выгоду от любых других проектов, которыми вы могли бы заниматься в это время.

Один из наименее эффективных способов, как этого можно добиться, заключается в перечислении возможностей вашего любимого средства IaC. Например, инструмент Terraform декларативный, поддерживает разные облака и имеет открытый исходный код. Это одна из многих ситуаций, когда разработчикам есть чему поучиться у коллег из отдела продаж. Большинство продавцов знают, что для продажи продуктов не следует сосредотачиваться на их технических возможностях. Основное внимание лучше уделять преимуществам: то есть вы должны говорить не о том, что может делать продукт («продукт X может делать Y»), а о том, что может делать ваш клиент с помощью этого продукта («вы можете делать Y, используя продукт X!»). Иными словами, покажите клиенту, какие удивительные возможности предоставляет ваш продукт.

Например, вместо того чтобы рассказывать своему начальнику о декларативности Terraform, убедите его, что ваша команда сможет быстрее справляться с проектами. Вместо разговоров о поддержке разных облаков расскажите о душевном спокойствии, которое обретет ваш начальник, зная о том, что потенциальный переход с одного облака на другое не потребует замены всего инструментария. И вместо объяснения преимуществ открытого кода помогите своему начальнику осознать, насколько проще будет искать новых разработчиков среди большого и активного сообщества Open Source.

Иллюстрация преимуществ послужит отличным началом. Однако существует еще более эффективная стратегия, известная самым лучшим продавцам: фокусирование на проблемах. Если понаблюдать за тем, как умелый продавец общается с клиентом, можно заметить, что большую часть разговора он выступает в роли слушателя, пытаясь понять одну конкретную вещь: какую ключевую проблему пытается решить клиент? Какая у него основная «болевая точка»? Лучшие продавцы пытаются решить проблемы своих клиентов, а не продать им какие-то возможности или преимущества. Если так получается, что предложенное решение включает в себя один из

продуктов продавца, это, конечно, плюс, но основное внимание уделяется решению проблем, а не продаже как таковой.

Поговорите со своим начальником и попытайтесь понять наиболее важные проблемы, над которыми он работает в этом квартале или году. Может оказаться, что их нельзя решить с помощью IaC. И это нормально! Возможно, из уст автора книги о Terraform это прозвучит как ересь, но технологии IaC нужны не всем командам. Их внедрение требует относительно больших затрат, которые в долгосрочной перспективе могут окупиться, а могут и нет. Например, если вы работаете в крошечном стартапе с одним системным администратором, или пишете прототип, который может быть выброшен через несколько месяцев, или просто занимаетесь сторонним проектом в свое удовольствие, управление инфраструктурой вручную часто является правильным выбором. Иногда, даже если технологии IaC отлично подходят для вашей команды, переход на них может иметь не самый высокий приоритет, поэтому, учитывая ограниченные ресурсы, лучше сосредоточиться на других проектах.

Если вы все же обнаружите, что одну из ключевых проблем, с которыми борется ваш начальник, можно решить с помощью IaC, покажите ему, как это может выглядеть. Представьте, к примеру, что одна из таких проблем — увеличение времени доступности. В последние месяцы у вас произошло множество перебоев в работе с многочасовыми простоями; ваши клиенты недовольны, а генеральный директор не дает спуску вашему начальнику, ежедневно наведываясь, чтобы проверить состояние дел. Вы начали исследовать проблему и обнаружили, что половина перебоев вызвана человеческим фактором во время развертывания: предположим, кто-то случайно пропустил важный этап процесса выкатывания, кто-то неправильно сконфигурировал сервер или инфраструктура финального тестирования не совпадала с тем, что у вас было в промышленной среде.

Теперь вместо рассказов о возможностях и преимуществах Terraform начните свой разговор с начальником со следующего: «У меня есть идея относительно того, как уменьшить число перебоев вдвое». Это гарантированно привлечет его внимание. Используйте данную возможность, чтобы обрисовать будущее, в котором ваш процесс развертывания полностью автоматизирован, надежен и воспроизводим, благодаря чему удастся исключить человеческий фактор, который был причиной половины предыдущих перебоев. Более того, с автоматизацией развертывания можно внедрить автоматические тесты, что сократит время простоя еще сильнее и позволит всей компании выкатывать обновления в два раза чаще. Пусть ваш начальник осознает, что именно он будет рассказывать об этих новостях генеральному директору. И в конце упомяните о том, что согласно вашим

исследованиям эту красочную картину можно воплотить в жизнь с помощью Terraform.

Конечно, нет никакой гарантии, что начальник согласится, но этот подход увеличит ваши шансы. А чтобы повысить их еще сильнее, переход нужно осуществлять поэтапно.

Сделайте переход постепенным

Один из важнейших уроков, которые я усвоил за время своей профессиональной деятельности: большинство крупных программных проектов проваливаются. Если взять мелкие ИТ-проекты (с бюджетом меньше миллиона долларов), то три четверти из них завершаются успешно, тогда как только один из десяти крупных проектов (дороже 10 миллионов долларов) удается завершить вовремя и без выхода за рамки бюджета, а каждый третий и вовсе закрывается на полпути58.

Поэтому я всегда начинаю волноваться, когда вижу, как компания пытается внедрить IaC одним махом в огромную инфраструктуру и с участием каждой команды. Причем это часто происходит в рамках какой-то более масштабной инициативы. Не могу не покачать головой, когда генеральный и технический директора большой компании приказывают в шестимесячный срок перевести все в облако, закрыть старые вычислительные центры и сделать так, чтобы все «занимались DevOps» (что бы это ни означало). Я не преувеличиваю, когда говорю, что подобные ситуации встречались мне не один десяток раз, и всегда эти инициативы проваливались. Два-три года спустя каждая из этих компаний по-прежнему в процессе миграции, старые вычислительные центры все еще работают, и никто точно не может сказать, занимается ли он DevOps.

Если вы хотите достичь успеха с внедрением IaC или любым другим процессом миграции, у вас есть только один вариант — инкрементальные изменения. Для этого недостаточно разбить задачу на небольшие последовательные шаги. Важно, чтобы каждый шаг приносил какую-то пользу, даже если он внезапно окажется последним.

Чтобы понять, почему это так важно, рассмотрим противоположный подход — *ложный инкрементализм*59. Представьте, что вы разбили масштабный процесс миграции на несколько мелких этапов и только по окончании последнего из них получите какую-то реальную пользу. Например, на первом этапе вы переписали клиентскую часть, но ее нельзя запускать, так как она зависит от новой серверной части. Затем вы переписываете серверный код, но его тоже нельзя использовать, пока данные не будут перенесены в новое хранилище. И только после окончания последнего этапа вы сможете все запустить и убедиться, что вся эта работа

была проделана не зря. Ждать завершения проекта для получения какой-либо выгоды очень рискованно. Если проект отменят, приостановят или существенно изменят на полпути, вложенные вами время и усилия могут не окупиться.

Именно это и происходит со многими масштабными процессами миграции. Проект изначально большой и, как это часто случается в мире программного обеспечения, требует куда больше времени, чем ожидалось. Пока он реализуется, могут поменяться рыночные условия или закончиться терпение у заинтересованных сторон (например, генеральный директор был не против потратить три месяца на ликвидацию технического долга, но после 12 месяцев уже пора выпускать новые продукты), и в итоге проект закрывается, так и не завершившись. Ложный инкрементализм дает наихудший из возможных результатов: вы заплатили огромную цену, а взамен не получили абсолютно ничего.

Таким образом, вам необходим инкрементализм. Нужно, чтобы каждая часть проекта приносила какую-то пользу, чтобы даже в случае его отмены и независимо от того, на каком этапе вы находитесь, он стоил потраченных усилий. Чтобы этого добиться, лучше всего сосредотачиваться на решении одной небольшой конкретной проблемы, а затем переходить к следующей. Например, вместо попыток перехода в облако одним махом попробуйте определить небольшое приложение или команду, испытывающую трудности, и займитесь исключительно их миграцией. Или вместо масштабного внедрения DevOps попытайтесь найти одну небольшую и конкретную проблему (такую как перебои в работе во время развертывания) и выработайте для нее решение (скажем, автоматизируйте самую проблемную часть развертывания с помощью Terraform).

Если вам удастся быстро справиться с одной реальной конкретной проблемой и сделать успешной одну команду, вы начнете набирать обороты. Эта команда может стать на вашу сторону и убедить другие команды в целесообразности миграции. Решение отдельной проблемы с развертыванием может порадовать генерального директора и обеспечить вам поддержку с применением IaC в других проектах. Это позволит вам одержать еще одну молниеносную победу и затем еще одну. Если повторять этот процесс снова и снова, принося пользу как можно раньше и чаще, у затеи с глобальной миграцией будет куда больше шансов на успех. Но даже если это начинание провалится, вы все равно улучшили один процесс развертывания и сделали успешной еще одну команду, поэтому вложения себя окупили.

Дайте своей команде время на обучение

Надеюсь, вам уже ясно, что внедрение IaC может потребовать значительных вложений. Это длительный процесс, и он не происходит магическим образом, по отмашке руководителя. Чтобы привести его в движение, необходимо заручиться поддержкой всех заинтересованных сторон, подготовить обучающие материалы (документацию, видеоруководства и, конечно же, эту книгу!) и дать членам команды время на то, чтобы овладеть новыми технологиями.

Если ваша команда не получит достаточно времени и ресурсов, ваш переход на IaC вряд ли будет успешным. Каким бы качественным ни был ваш код, без полной поддержки со стороны команды все пойдет по такому сценарию.

1. Один разработчик, вдохновленный концепцией IaC, тратит несколько месяцев на написание прекрасного кода Terraform и развертывание с его помощью большого объема инфраструктуры.

2. Этот разработчик счастлив и продуктивен, но, к сожалению, у остальных членов команды не было времени на изучение и внедрение Terraform.

3. Случается неизбежное: перебой в работе системы.

4. Теперь с этой проблемой проходится иметь дело другому члену команды. У него есть два варианта: либо исправить проблему так, как он это делал всегда, путем внесения изменений вручную, что займет несколько минут, либо использовать Terraform, на что могут уйти часы и дни, так как он незнаком с этим инструментом. Ваши коллеги, скорее всего, разумные рациональные люди, поэтому они почти наверняка выберут первый вариант.

5. В результате ручного изменения код Terraform больше не соответствует тому, что на самом деле развернуто. Поэтому, даже если кто-то в вашей команде выберет второй вариант и попробует Terraform, он вполне может получить странную ошибку. И если это случится, он потеряет доверие к коду Terraform и опять вернется к первому варианту, добавив новые ручные изменения. Это дальше усугубит рассинхронизацию кода с реальностью, и следующий, кто попробует Terraform, будет иметь еще более высокие шансы на получение странной ошибки. Это создаст порочный круг, в котором все больше и больше членов команды будут вносить изменения вручную.

6. На удивление быстро все вернутся к ручному управлению инфраструктурой, код Terraform станет абсолютно бесполезным, а месяцы работы, которые ушли на его написание, окажутся пустой тратой времени.

Это не какой-то гипотетический сценарий, а то, что я сам наблюдал во множестве разных компаний. Их огромные и дорогие проекты, полные прекрасного кода Terraform, пылятся в сторонке. Чтобы этого избежать,

будет мало убедить свое начальство в целесообразности использования Terraform. Вы должны дать всем членам своей команды время на изучение этого инструмента и усвоение навыков работы с ним, чтобы, когда снова возникнут неполадки, их было легче исправить в коде, а не вручную.

В ускорении внедрения технологии IaC может помочь четко определенный процесс ее использования. Если вы изучаете или учитесь применять IaC внутри небольшой команды, это вполне можно делать на ходу прямо на компьютере для разработки. Но с ростом вашей компании и расширением сценариев использования IaC следует наладить более систематический, воспроизводимый, автоматизированный рабочий процесс развертывания.

Процесс развертывания кода приложений

В этом разделе вы познакомитесь с типичным рабочим процессом доставки прикладного кода (такого как приложение на Ruby on Rails или Java/Spring), начиная с этапа разработки и заканчивая промышленной средой. В области DevOps этот рабочий процесс известен достаточно хорошо, поэтому вы уже должны быть знакомы с некоторыми его аспектами. Позже в этой главе мы поговорим о доставке инфраструктурного кода (вроде модулей Terraform). Этот процесс куда менее распространен в нашей индустрии, поэтому будет полезно сравнить его с доставкой прикладного кода и провести параллели между их аналогичными этапами.

Рассмотрим рабочий процесс для кода приложений.

1. Используем систему управления версиями.
2. Выполняем код локально.
3. Вносим изменения в код.
4. Подаем изменения на рассмотрение.
5. Выполняем автоматические тесты.
6. Проводим слияние и выпускаем новую версию.
7. Развертываем.

По очереди пройдемся по каждому из этих этапов.

Использование системы управления версиями

Весь ваш код должен находиться в системе управления версиями. Без исключений. Это первый пункт классического теста Джоэла (http://bit.ly/2meqAb7), который Джоэл Спольски создал примерно 20 лет назад. С тех пор изменилось лишь то, что: а) благодаря таким инструментам, как GitHub, использовать системы управления версиями теперь проще, чем когда-либо: б) в виде кода можно описывать все больше и больше вещей. Это относится к документации (как файл README в формате Markdown),

конфигурации приложений (вроде файла настроек на YAML), спецификациям (скажем, тестовый код, написанный на RSpec), тестам (например, автоматические тесты с применением JUnit), базам данных (типа схемы миграции, написанной на Active Record) и, конечно, к инфраструктуре.

В оставшейся части этой книги я буду исходить из того, что для управления версиями вы используете Git. Например, так можно загрузить примеры с кодом из нашего репозитория:

```
$ git clone https://github.com/brikis98/terraform-up-and-running-code.git
```

Эта команда по умолчанию загружает из репозитория ветку master, но вы, скорее всего, занимаетесь разработкой в отдельной ветке. Вот как с помощью команды gitcheckout можно создать ветку под названием example-feature и переключиться на нее:

```
$ cd terraform-up-and-running-code
$ git checkout -b example-feature
Switched to a new branch 'example-feature'
```

Локальное выполнение кода

Теперь, когда код находится на вашем компьютере, можно выполнить его локально. Помните пример с веб-сервером на Ruby из главы 7? Так он запускается:

```
$ cd code/ruby/08-terraform/team
$ ruby web-server.rb
[2019-06-15 15:43:17] INFO  WEBrick 1.3.1
[2019-06-15 15:43:17] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-06-15 15:43:17] INFO  WEBrick::HTTPServer#start: pid=28618 port=8000
```

Теперь его можно протестировать вручную с помощью curl:

```
$ curl http://localhost:8000
Hello, World
```

При желании можно также выполнить автоматические тесты:

```
$ ruby web-server-test.rb
(...)
Finished in 0.633175 seconds.
----------------------------------------
8 tests, 24 assertions, 0 failures, 0 errors
100% passed
----------------------------------------
```

Ключевой момент здесь в том, что ручные и локальные тесты для кода приложения можно выполнять локально на своем компьютере. Позже вы увидите, что это неприменимо к той же части рабочего процесса для инфраструктурных изменений.

Внесение изменений в код

Вы уже можете запустить код приложения. Теперь начнем вносить изменения. Это итеративный процесс: вы редактируете код, повторно запускаете ручные или автоматические тесты, чтобы убедиться в корректности внесенных изменений, снова редактируете код, опять выполняете тесты и т. д.

Например, вы можете заменить вывод web-server.rb на Hello,Worldv2, перезапустить сервер и проверить результат:

```
$ curl http://localhost:8000
Hello, World v2
```

Еще раз выполните тесты и посмотрите, будут ли они пройдены:

```
$ ruby web-server-test.rb
(...)
Failure: test_integration_hello(TestWebServer)
web-server-test.rb:53:in `block in test_integration_hello'
    50:     do_integration_test('/', lambda { |response|
    51:       assert_equal(200, response.code.to_i)
    52:       assert_equal('text/plain', response['Content-Type'])
 => 53:       assert_equal('Hello, World', response.body)
    54:     })
    55:   end
    56:
web-server-test.rb:100:in `do_integration_test'
web-server-test.rb:50:in `test_integration_hello'
<"Hello, World"> expected but was
<"Hello, World v2">
(...)
Finished in 0.236401 seconds.
---------------------------------------------
8 tests, 24 assertions, 2 failures, 0 errors
75% passed
---------------------------------------------
```

Вы сразу же увидите, что автоматические тесты по-прежнему ожидают старое значение, поэтому вы можете быстро их исправить. Суть этой части рабочего процесса в том, чтобы оптимизировать обратную связь, сделав

минимальной задержку между внесением изменения и подтверждением его корректности.

В процессе работы вы должны регулярно фиксировать свой код, указывая четкие сообщения с описанием ваших изменений:

```
$ git commit -m "Updated Hello, World text"
```

Подача изменений на рассмотрение

В конечном счете код и тесты заработают так, как вам того хочется. Это будет означать, что их пора подать на рассмотрение. Это можно сделать либо с помощью отдельного инструмента для разбора кода (такого как Phabricator или ReviewBoard), либо посредством *запроса на включение внесенных изменений* (pull request), если вы используете GitHub. Такие запросы можно создавать несколькими способами. Один из самых простых — публикация ветки example-feature обратно в origin (то есть обратно в GitHub). В этом случае GitHub автоматически выведет в терминал URL-адрес запроса на включение изменений:

```
$ git push origin example-feature
(...)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'example-feature' on GitHub by visiting:
remote: https://github.com/<OWNER>/<REPO>/pull/new/example-feature
remote:
```

Откройте этот URL-адрес в своем браузере, введите название и описание запроса и нажмите кнопку Create (Создать). После этого члены вашей команды смогут сделать разбор изменений, как показано на рис. 8.1.

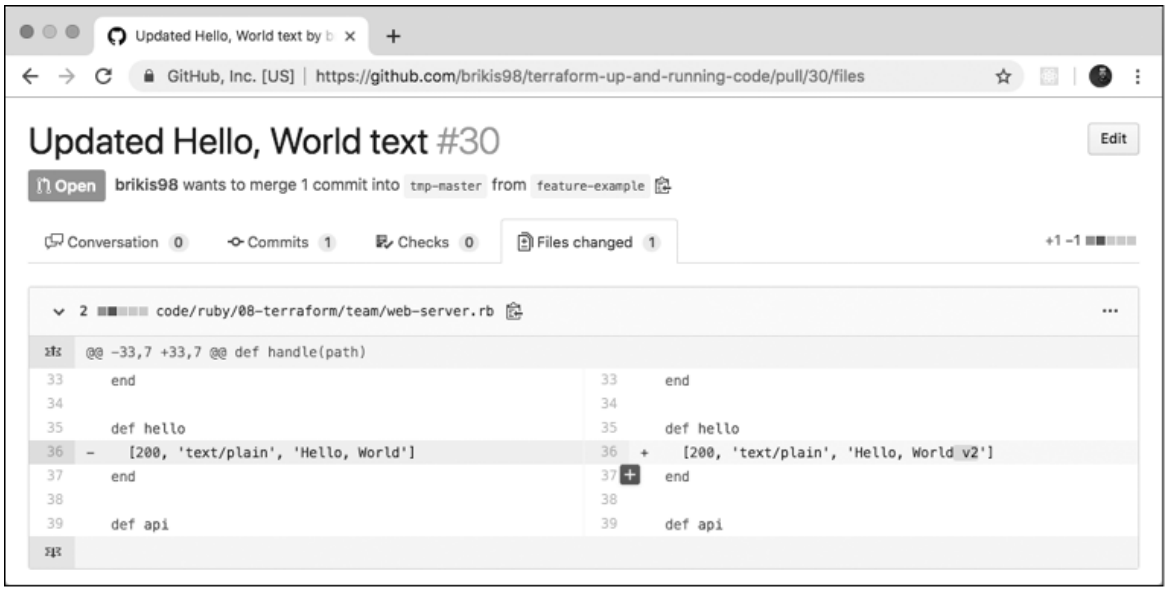


Рис. 8.1. Запрос на включение внесенных изменений в GitHub

Выполнение автоматических тестов

Вы должны настроить хуки фиксации, чтобы автоматические тесты запускались каждый раз, когда вы сохраняете или загружаете свои изменения в систему управления версиями. Чаще всего для этого используют сервер *непрерывной интеграции* (continuous integration, или CI), такой как Jenkins, CircleCI или TravisCI. Самые популярные CI-серверы имеют встроенную интеграцию с GitHub, которая, помимо автоматического запуска тестов при фиксации кода, умеет показывать эти тесты в самом запросе на включение внесенных изменений (рис. 8.2).

На рис. 8.2 можно видеть, что сервер CircleCi выполнил для кода в заданной ветке модульные, интеграционные и сквозные тесты, а также некоторые проверки со статическим анализом (в виде сканирования на предмет уязвимостей с помощью инструмента под названием snyk) и все они прошли успешно.

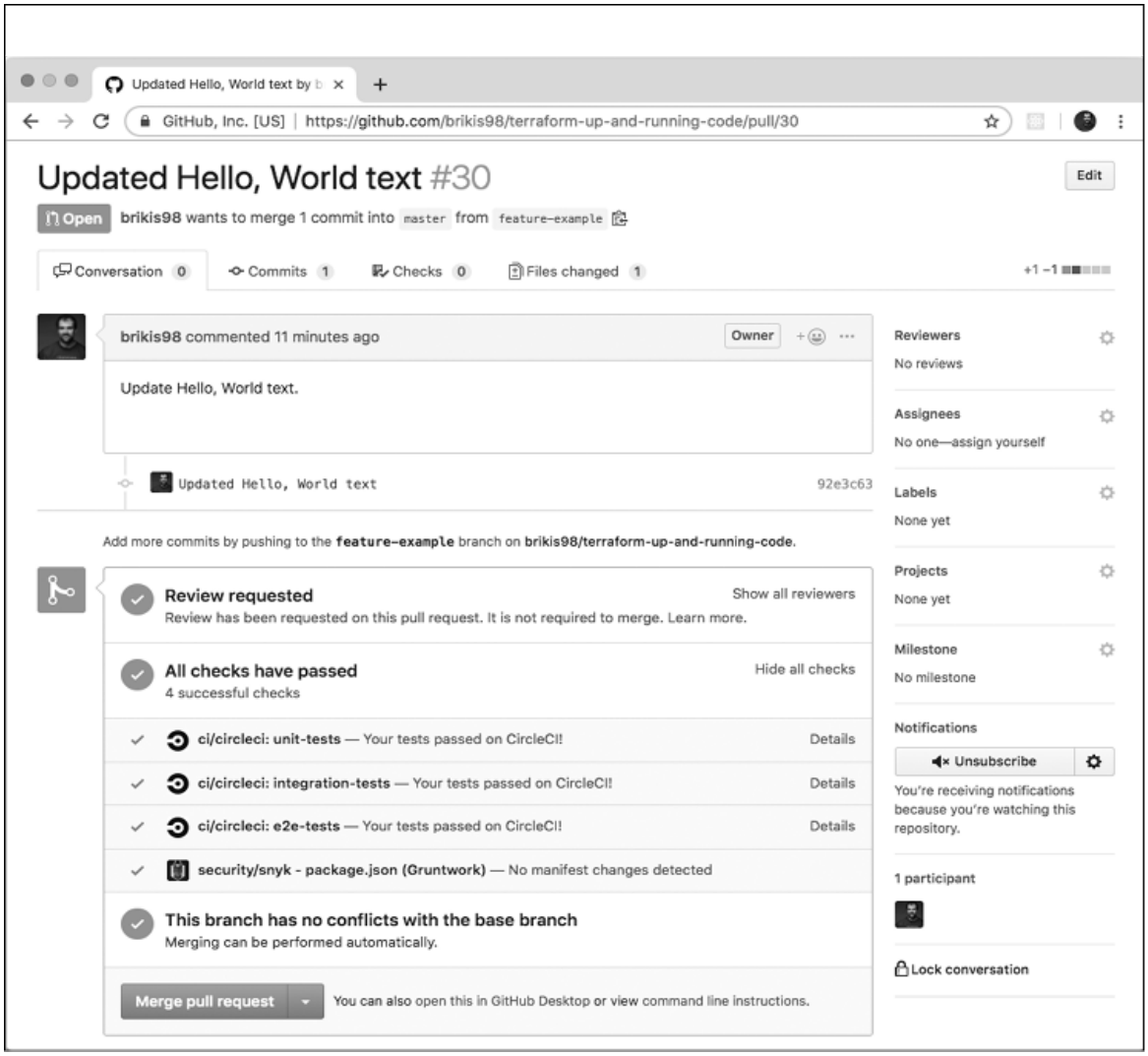


Рис. 8.2. В запросе на включение внесенных изменений в GitHub показываются результаты автоматических тестов из CircleCi

Слияние и выпуск новой версии

Члены вашей команды должны просмотреть ваши изменения в поиске возможных ошибок, заодно исправляя их в соответствии с рекомендациями по оформлению кода (подробнее об этом — чуть позже) и проверяя, пройдены ли имеющиеся тесты. Они также должны следить за тем, предусмотрены ли новые тесты для любой добавленной вами логики. Если все выглядит хорошо, ваш код можно объединить с веткой master.

Следующим шагом будет выпуск кода. Если вы используете подход с неизменяемой инфраструктурой (как обсуждалось в подразделе «Средства шаблонизации серверов» на с. 31), для выпуска кода приложения его нужно упаковать в новый, неизменяемый артефакт с поддержкой версионирования. В зависимости от того, как именно вы хотите упаковывать и развертывать

свое приложение, это может быть новый образ Docker, новый образ виртуальной машины (например, новый экземпляр AMI), новый JAR-файл, новый TAR-файл и т. д. Какой бы формат вы ни выбрали, убедитесь, что ваш артефакт неизменяемый (то есть вы его никогда не модифицируете) и ему присвоен уникальный номер версии (чтобы вы могли отличить его от других).

Например, если вы упаковываете свое приложение с помощью Docker, номер версии можно хранить в теге Docker. В качестве тега подойдет идентификатор фиксации кода (хеш SHA1). Это позволит привязать развертываемый вами образ к коду, который он содержит:

```
$ commit_id=$(git rev-parse HEAD)
$ docker build -t brikis98/ruby-web-server:$commit_id .
```

Эти команды соберут новый образ Docker под названием brikis98/ruby-webserver и назначат ему тег с идентификатором самой последней фиксации кода, который будет иметь примерно такой вид: 92e3c6380ba6d1e8c9134452ab6e26154e6ad849. Если позже придется заниматься отладкой этого образа, вы сможете узнать, какой именно код он содержит, проверив его тег с ID-фиксации:

```
$ git checkout 92e3c6380ba6d1e8c9134452ab6e26154e6ad849
HEAD is now at 92e3c63 Updated Hello, World text
```

У идентификаторов фиксаций есть один недостаток: их сложно читать и запоминать. В качестве альтернативы можно создать тег Git:

```
$ git tag -a "v0.0.4" -m "Update Hello, World text"
$ git push --follow-tags
```

Этот тег тоже ссылается на определенную фиксацию в Git, но с помощью более понятного имени. Вы можете использовать его для своих образов Docker:

```
$ git_tag=$(git describe --tags)
$ docker build -t brikis98/ruby-web-server:$git_tag .
```

Таким образом, при отладке можно загрузить код с определенным тегом:

```
$ git checkout v0.0.4
Note: checking out 'v0.0.4'.
(...)
HEAD is now at 92e3c63 Updated Hello, World text
```

Развертывание

Теперь, когда у вас есть артефакт с поддержкой версионирования, вы можете его развернуть. Развертывание прикладного кода можно выполнить множеством разных способов: в зависимости от типа вашего приложения, от

того, как вы его упаковали, как вы хотите его запускать, какая у вас архитектура, какие инструменты вы используете и т. д. Вот несколько ключевых моментов, которые стоит учесть:

- инструментарий для развертывания;
- стратегии развертывания;
- сервер для развертывания;
- продвижение артефакта по разным окружениям.

Инструментарий для развертывания

Существует множество разных инструментов для развертывания приложений. Выбор зависит от того, как вы упаковали свой код и как хотите его запускать. Вот несколько примеров.

•*Terraform*. Как вы уже видели в этой книге, Terraform можно использовать для развертывания разных типов приложений. Например, в начальных главах вы создали модуль под названием asg-rolling-deploy, который умел выполнять скользящее развертывание в ASG. Если бы вы упаковали свое приложение в виде AMI (например, с помощью Packer), его новые версии можно было бы развертывать с применением того же модуля asg-rolling-deploy; для этого вам нужно было бы обновить параметр ami в своем коде Terraform и выполнить terraformapply.

• *Средства оркестрации Docker*. Существует ряд средств оркестрации для развертывания и администрирования приложений в виде контейнеров Docker, включая Kubernetes (наверное, самое популярное из них), Apache Mesos, HashiCorp Nomad и Amazon ECS. Если ваше приложение упаковано с помощью Docker, вы можете развертывать его новые версии, используя команду kubectlapply из состава Kubernetes (kubectl — это утилита командной строки для взаимодействия с Kubernetes) и передавая ей файл YAML, который определяет имя и тег образа для развертывания.

• *Скрипты*. Terraform и большинство средств оркестрации поддерживают лишь ограниченный набор стратегий развертывания (об этом поговорим далее). Если ваши требования выходят за эти рамки, вам, скорее всего, придется писать собственные скрипты на языке программирования общего назначения (например, Python или Ruby) и использовать средства управления конфигурацией (как Ansible или Chef) или другие инструменты для автоматизации серверов (скажем, Capistrano).

Самое сложное в написании такого рода скриптов — это обработка сбоев. Например, что произойдет, если компьютер, выполняющий ваш скрипт развертывания, потеряет интернет-соединение или сломается посреди этого процесса? Не так-то просто написать скрипт, который был бы идемпотентным и мог бы восстановиться после сбоя и корректно завершить

развертывание. Вам, возможно, придется записывать куда-то состояние скрипта (хотя иногда состояние можно вывести, обращаясь к инфраструктуре) и создать конечный автомат, способный справиться с любым возможным состоянием — начальным и переходным.

Стратегии развертывания

Существует целый ряд разных стратегий, которые можно использовать для развертывания приложений в зависимости от конкретных требований. Представьте, что у вас есть пять запущенных копий старой версии вашего приложения и вы хотите выкатить новую версию. Вот несколько самых распространенных стратегий, которые могут вам в этом помочь.

•*Скользящие развертывания с заменой*. Удалите старую копию приложения, разверните вместо нее новую, подождите, пока она не пройдет проверку работоспособности, направьте к ней реальный трафик и повторяйте этот процесс, пока не будут заменены все старые копии. Эта стратегия развертывания гарантирует, что у вас никогда не будет запущено больше пяти копий вашего приложения, что может быть полезным, если ваши ресурсы ограничены (например, если каждая копия приложения выполняется на отдельном физическом сервере) или если вы имеете дело с системой, которая хранит свое состояние и идентифицирует каждую копию уникальным образом (это часто случается в консенсусных системах, таких как Apache ZooKeeper). Стоит отметить, что эта стратегия может заменять сразу несколько копий приложения (при условии, что вы сможете выдержать нагрузку и не потеряете данные при работе меньшего числа копий) и в ходе развертывания у вас будут одновременно запущены как старые, так и новые версии.

• *Скользящие развертывания без замены*. Разверните одну новую копию приложения, подождите, пока она не пройдет проверку работоспособности, направьте к ней реальный трафик, удалите старую копию и затем повторяйте этот процесс, пока не будут заменены все старые копии. Эта стратегия развертывания годится только в том случае, если ресурсы можно инициализировать динамически (например, когда ваше приложение находится в облаке, в котором в любой момент можно запустить новые виртуальные серверы) и если ваше приложение допускает выполнение более пяти копий одновременно. Преимущество этого подхода в том, что у вас всегда запущено не менее пяти копий приложения и во время развертывания емкость вашей системы не понижается. Стоит отметить, что эта стратегия может заменять сразу несколько копий приложения (как это делал ваш модуль asg-rolling-deploy) и в ходе развертывания у вас будут одновременно запущены как старые, так и новые версии.

• *«Сине-зеленые» развертывания*. Разверните несколько новых копий приложения, подождите, пока они запустятся и пройдут проверку работоспособности, переключите на них промышленный трафик и затем удалите старые копии. «Сине-зеленые» развертывания работают только в том случае, если ресурсы можно инициализировать динамически (например, когда ваше приложение находится в облаке, в котором в любой момент можно запустить новые виртуальные серверы) и ваше приложение допускает выполнение более пяти копий одновременно. Преимущество этого подхода в том, что пользователям всегда доступна только одна версия вашего приложения и у вас всегда запущено не менее пяти копий, поэтому в ходе развертывания емкость вашей системы не понижается.

• *Канареечные развертывания*. Разверните одну новую копию приложения, подождите, пока она не пройдет проверку работоспособности, направьте к ней реальный трафик и затем приостановите развертывание. Во время этой паузы сравните новую (канареечную) копию приложения с одной из старых (контрольных) копий. Это сравнение можно проводить на разных уровнях: загрузка процессора, использование памяти, латентность, пропускная способность, частота возникновения ошибок в журнальных записях, HTTP-коды ответов и т. д. В идеале оба сервера будут выглядеть идентично. Это даст вам уверенность в том, что новый код будет работать без каких-либо проблем. В этом случае вы возобновляете развертывание и завершаете его с помощью одной из скользящих стратегий. Но, если вы обнаружили какие-то различия в поведении, это может быть признаком проблем в новом коде, поэтому развертывание следует отменить, а канареечную версию удалить, пока ситуация не ухудшилась.

Название этой стратегии пришло из угольной промышленности. Шахтеры брали с собой в шахту канареек: если тоннель был наполнен опасными газами (например, окисью углерода), канарейка умирала до того, как эти газы причиняли вред самим шахтерам. Это служило системой раннего предупреждения об опасности, благодаря которому шахтеры знали, что им следует немедленно покинуть тоннель, пока не случилась беда. Канареечные развертывания работают похожим образом. Они позволяют методично проверять новый код в промышленных условиях, и если что-то пойдет не так, вы узнаете об этом на ранних этапах, когда проблема затронула лишь небольшую часть ваших пользователей. Таким образом, у вас будет достаточно времени, чтобы отреагировать и предотвратить дальнейший ущерб.

Канареечные развертывания часто используют в сочетании с *ротацией функций* — это когда все новые возможности заворачиваются в условное выражение. По умолчанию условное выражение ложное, поэтому при

начальном развертывании кода новая возможность выключена. Благодаря этому канареечный сервер, который вы развернули, должен вести себя точно так же, как контрольный, а любые расхождения можно автоматически считать проблемными и инициировать откат назад. Позже, если проблем не обнаружилось, вы можете включить новую функцию для части ваших пользователей с помощью внутреннего веб-интерфейса. Например, вы можете начать с работников компании: если все работает, функцию можно сделать доступной для 1 % пользователей; если и после этого все идет хорошо, число пользователей можно расширить до 10 % и т. д. Если в какой-то момент возникнет проблема, эту функцию можно будет опять свернуть. Этот процесс позволяет разделить *развертывание* нового кода и *выпуск* новых возможностей.

Сервер развертывания

Развертывание нужно запускать на CI-сервере, а не на компьютере для разработки. Это даст вам несколько преимуществ.

•*Полная автоматизация*. Для запуска развертываний на CI-сервере придется полностью автоматизировать все этапы этого процесса. Таким образом, развертывание будет описано в коде и ни один из этапов не будет пропущен по чьей-то ошибке. Это сделает его быстрым и воспроизводимым.

• *Однородное окружение*. Если запускать развертывания на компьютерах разработчиков, разница в их конфигурации неминуемо приведет к ошибкам. Это, скажем, касается разных операционных систем, версий зависимостей (разные версии Terraform), конфигурационных файлов и того, что на самом деле развертывается (например, разработчик может случайно развернуть изменение, которое не было зафиксировано в системе управления версиями). Вы можете избежать всех этих проблем, если будете развертывать все с одного и того же CI-сервера.

• *Лучшее управление доступом*. Вместо того чтобы раздавать права доступа каждому разработчику, вы можете сделать так, чтобы только CI-сервер мог выполнять развертывание (особенно в промышленной среде). Соблюдать правила безопасности для одного сервера намного легче, чем для десятков и сотен людей с доступом к промышленной системе.

Продвижение артефактов по разным окружениям

Если вы поддерживаете свою инфраструктуру неизменяемой, для выкатывания обновлений одну и ту же версию артефакта необходимо распространить по разным окружениям. Например, если у вас есть окружения для разработки (Dev), финального тестирования (Staging) и

промышленного использования (Production), для выкатывания версии v0.0.4 вашего приложения нужно сделать следующее.

1. Развернуть версию v0.0.4 в Dev.
2. Провести в Dev ручное и автоматическое тестирование.
3. Если версия v0.0.4 хорошо работает в Dev, повторить шаги 1 и 2 для Staging (это называют *продвижением артефакта*).
4. Если версия v0.0.4 хорошо работает в Staging, повторить шаги 1 и 2 для Production.

Мы везде используем один и тот же артефакт, поэтому, если он работает в одном окружении, существует высокая вероятность того, что он заработает и в другом. Но если вы столкнетесь с какими-либо проблемами, то всегда сможете откатиться к более старой версии.

Процесс развертывания инфраструктурного кода

Мы закончили с процессом развертывания кода приложений. Пришло время поговорить о том, как развертывается инфраструктурный код. В этом разделе под инфраструктурным я понимаю код, который написан с использованием любого средства IaC (конечно, включая Terraform) и с помощью которого можно развертывать произвольные изменения инфраструктуры, не ограничиваясь одним приложением.

Рассмотрим рабочий процесс для инфраструктурного кода.

1. Используем систему управления версиями.
2. Выполняем код локально.
3. Вносим изменения в код.
4. Подаем изменения на рассмотрение.
5. Выполняем автоматические тесты.
6. Проводим слияние и выпускаем новую версию.
7. Развертываем.

Внешне это идентично аналогичному процессу для прикладного кода, но все важные отличия — внутри. Развертывание изменений в инфраструктурном коде более сложное, и методики, которые для этого используются, известны не так хорошо. Поэтому, чтобы вам было легче, каждый этап будет привязан к аналогичному этапу из процесса для кода приложений.

Использование системы управления версиями

Весь ваш инфраструктурный код, как и прикладной, должен находиться в системе управления версиями. Значит, как и прежде, для загрузки кода используется команда gitclone. Однако в контексте инфраструктуры существует несколько дополнительных требований:

• отдельные репозитории для *текущей инфраструктуры* и *модулей*;
• золотое правило Terraform;
• проблема с ветками.

Отдельные репозитории для текущей инфраструктуры и модулей

Как уже обсуждалось в главе 4, для кода Terraform обычно нужно как минимум два репозитория: один для модулей, а другой — для текущей инфраструктуры. В первом вы создаете свои универсальные модули с версионированием — как те, которые вы создали в предыдущих главах этой книги (cluster/asg-rolling-deploy, data-stores/mysql, networking/alb и services/hello-world-app). Второй репозиторий определяет вашу текущую инфраструктуру, которую вы развернули в том или ином окружении (Dev, Stage, Prod и т. д.).

Есть один хороший рабочий подход: нужно организовать единую инфраструктурную команду, которая специализируется на создании многоразовых надежных модулей промышленного уровня. Создавая библиотеку модулей, реализующих идеи из главы 6, эта команда может сделать для вашей компании отличное конкурентное преимущество. У каждого модуля будет компонуемый API, основательная документация (в том числе и исполняемая документация в папке examples), комплексный набор автоматических тестов, поддержка версионирования и совместимость со всеми требованиями, которые компания предъявляет к инфраструктуре промышленного уровня (то есть безопасность, соблюдение стандартов и правовых норм, масштабируемость, высокая доступность, мониторинг и т. д.).

Если вы собираете такую библиотеку (или покупаете уже готовую60), все остальные команды в вашей компании смогут пользоваться ею для развертывания и администрирования своей инфраструктуры (чем-то напоминает каталог услуг). При этом: а) никому больше не придется собирать эту инфраструктуру с нуля; б) системные администраторы больше не будут задерживать рабочий процесс, так как они теперь не отвечают за развертывание и администрирование инфраструктуры для каждой команды. Вместо этого сисадмины смогут уделять большую часть своего времени написанию инфраструктурного кода, а остальные команды получат возможность работать автономно, применяя эти модули для подготовки необходимых ресурсов. И, поскольку внутри все используют одни и те же канонические модули, с ростом компании и изменением требований системные администраторы смогут выпускать новые версии этих модулей для всех команд, обеспечивая тем самым согласованность и удобство в обслуживании.

Если быть точным, удобство в обслуживании будет сохраняться до тех пор, пока соблюдается золотое правило Terraform.

Золотое правило Terraform

Так можно быстро проверить работоспособность вашего кода Terraform: откройте репозиторий *текущей* инфраструктуры, выберите наугад несколько папок и выполните команду terraformplan для каждой. Если в каждом случае вывод указывает на отсутствие изменений, все прекрасно: значит, ваш инфраструктурный код совпадает с тем, что у вас на самом деле развернуто. Если вам иногда попадаются небольшие расхождения и вы время от времени слышите от своих коллег оправдания («О, точно, я немного подкрутил вручную эту небольшую штучку и забыл обновить код»), ваш код не соответствует реальности и вскоре это может вызвать проблемы. Если команда terraformplan завершается полной неудачей и возвращает ошибки или каждый раз получается огромное расхождение, ваш код не имеет никакого отношения к реальной жизни и, скорее всего, бесполезный.

Идеальный сценарий (или то, к чему вы на самом деле стремитесь) — это то, что я называю *золотым правилом Terraform*.

Основная ветка репозитория с текущей инфраструктурой должна один в один соответствовать тому, что на самом деле развернуто в промышленной среде.

Разберем это предложение на части, начав с конца.

•*«…что на самом деле развернуто»*. Чтобы гарантировать, что код Terraform в репозитории текущей инфраструктуры — актуальное представление того, что на самом деле развернуто, вы *никогда не должны вносить сторонние изменения*. Начав использовать Terraform, вы должны перестать менять свою инфраструктуру через веб-консоль, ручные API-вызовы или любой другой механизм. Как вы уже видели в главе 5, сторонние изменения не только приводят к сложным ошибкам, но и нивелируют многие преимущества от применения технологии IaC как таковой.

• *«…один в один соответствовать…»*. При просмотре репозитория с текущей инфраструктурой мне нужна возможность быстро определить, какие ресурсы и в какой среде были развернуты. То есть каждый ресурс должен полностью соответствовать какой-то строчке кода, сохраненной в репозитории. На первый взгляд это кажется очевидным, но здесь на удивление легко ошибиться. Например, как я только что упомянул, вы можете внести сторонние изменения, в результате чего в коде будет одно, а в реальности — другое. Менее очевидной ошибкой является использование

рабочих областей Terraform для управления окружениями таким образом, что инфраструктура может быть развернута без соответствующего кода. Иными словами, если вы используете рабочие области, репозиторий вашей текущей инфраструктуры будет содержать лишь одну копию кода, хотя на самом деле этот код мог развернуть 3 или 30 окружений. Поэтому по одному лишь коду нельзя понять, что действительно развернуто, а это непременно приведет к ошибкам и затруднит поддержку. Таким образом, как уже описывалось в подразделе «Изоляция через рабочие области» на с. 114, вместо управления окружениями с помощью рабочих областей каждое окружение следует описывать в отдельной папке, используя отдельные файлы. Так вы сможете точно сказать, какие окружения у вас развернуты, лишь просмотрев репозиторий текущей инфраструктуры. Позже в этой главе вы увидите, как это делать с минимальным дублированием кода.

• *«Основная ветка…»*. Для понимания того, что на самом деле развернуто в промышленной среде, должно быть достаточно просмотра одной-единственной ветки. Обычно это основная ветка, master. Значит, все изменения, которые влияют на промышленную среду, должны сохраняться непосредственно в master (вы можете создавать отдельные ветки, но только для создания запросов на включение внесенных изменений с последующим их слиянием с основной веткой) и команду terraformapply для промышленной среды необходимо выполнять только для этой ветки. Почему так, я объясню в следующем пункте.

Проблема с ветками

В главе 3 вы видели механизм блокирования, встроенный в хранилища Terraform. Благодаря ему, если два члена команды одновременно выполнят terraformapply для одного и того же набора конфигурационных файлов, они не перезапишут изменения друг друга. К сожалению, это решает лишь часть проблемы. Несмотря на то, что хранилища Terraform предоставляют механизм блокирования состояния, они не могут вам помочь с блокированием на уровне кода Terraform. В частности, если два члена команды развертывают один и тот же код в одном и том же окружении, но из разных веток, возникнут конфликты, которые нельзя предотвратить путем блокирования.

Представьте, что ваша коллега, Анна, вносит какие-то изменения в конфигурацию Terraform для приложения под названием foo, которое состоит из единственного сервера Amazon EC2:

```
resource "aws_instance" "foo" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
```

```
}
```

Это приложение получает много запросов, поэтому Анна решает поменять instance_type с t2.micro на t2.medium:

```
resource "aws_instance" "foo" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.medium"
}
```

Это она увидит при выполнении команды terraformplan:

```
$ terraform plan
(...)
Terraform will perform the following actions:
  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                      = "ami-0c55b159cbfafe1f0"
        id                       = "i-096430d595c80cb53"
        instance_state           = "running"
      ~ instance_type            = "t2.micro" -> "t2.medium"
        (...)
    }
Plan: 0 to add, 1 to change, 0 to destroy.
```

Изменения кажутся приемлемыми, поэтому Анна развертывает их в среде Staging.

А тем временем Билл начинает вносить изменения в конфигурацию Terraform того же приложения, но в другой ветке. Он просто хочет добавить тег:

```
resource "aws_instance" "foo" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = {
    Name = "foo"
  }
}
```

Внесенные Анной изменения уже развернуты в Staging, но, поскольку они хранятся в другой ветке, в коде Билла по-прежнему содержится старое значение для поля instance_type. Это видит Билл, когда выполняет команду plan (следующий вывод урезан, чтобы его было легче читать):

```
$ terraform plan
(...)
Terraform will perform the following actions:
  # aws_instance.foo will be updated in-place
```

```
~ resource "aws_instance" "foo" {
    ami                  = "ami-0c55b159cbfafe1f0"
    id                   = "i-096430d595c80cb53"
    instance_state       = "running"
  ~ instance_type        = "t2.medium" -> "t2.micro"
  + tags = {
      + "Name" = "foo"
    }
    (...)
}
```

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

О нет, он собирается отменить изменение поля instance_type, сделанное Анной! Если девушка по-прежнему занимается тестированием в среде Staging, она будет крайне удивлена, когда сервер внезапно развернется заново и начнет вести себя иначе. Но есть и хорошие новости: если Билл внимательно прочитает вывод команды plan, он сможет заметить ошибку еще до того, как она затронет Анну. Тем не менее этот пример иллюстрирует проблемы, которые могут возникнуть при развертывании изменений из разных веток в общую среду.

Установленное хранилищами Terraform блокирование здесь не поможет, так как этот конфликт не имеет никакого отношения к конкурентному изменению файла состояния. Билл и Анна могут применять свои изменения с разницей в несколько недель, но проблема останется прежней. Основная причина в том, что ветвление и Terraform плохо сочетаются между собой. Terraform косвенным образом привязывает код к развернутой в реальном мире инфраструктуре. Поскольку реальность у нас с вами одна, разделение кода Terraform на ветки имеет мало смысла. Поэтому при работе с любой общей средой (например, Stage, Prod) всегда развертывайте код из одной ветки.

Локальное выполнение кода

Следующим шагом после загрузки кода на свой компьютер будет его запуск. Но есть одна загвоздка: в отличие от прикладного код Terraform не имеет такого понятия, как «локальный компьютер». Например, вы не можете развернуть AWS ASG на своем ноутбуке. Как уже обсуждалось в подразделе «Основы ручного тестирования» на с. 255, чтобы протестировать код Terraform вручную, его необходимо запустить в изолированной среде, такой как учетная запись AWS, специально выделенная для разработчиков (еще лучше, если у каждого разработчика будет своя учетная запись AWS).

Подготовив изолированную среду, вы можете начать ручное тестирование, выполнив команду terraformapply:

```
$ terraform apply
(...)
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
Outputs:
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

Чтобы проверить, работает ли развернутая инфраструктура, можно воспользоваться инструментами вроде curl:

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
Hello, World
```

Для автоматических тестов, написанных на Go, в изолированной учетной записи, предназначенной для тестирования, выполняется команда gotest:

```
$ go test -v -timeout 30m
(...)
PASS
ok  terraform-up-and-running  229.492s
```

Внесение изменений в код

Получив возможность запускать свой код Terraform, можно приступить к внесению итеративных изменений — точно так же, как вы это делали с кодом приложения. Для развертывания изменений можно заново выполнить terraformapply, а чтобы убедиться в том, что они работают, можно снова запустить curl:

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
Hello, World v2
```

Чтобы проверить, по-прежнему ли код проходит тесты, можно выполнить gotest:

```
$ go test -v -timeout 30m
(...)
PASS
ok  terraform-up-and-running  229.492s
```

Единственное отличие от прикладного кода в том, что тестирование инфраструктурного кода обычно занимает больше времени, поэтому вам стоит подумать над тем, как сократить этот процесс, чтобы получать результаты как можно быстрее. Из пункта «Стадии тестирования» на с. 299 вы узнали, что можно разбить тест на отдельные стадии и выполнять только те из них, которые вам нужны, что существенно ускоряет обратную связь.

Представьте, что у вас есть тест, который развертывает базу данных и приложение, проверяет их работу и затем удаляет. При его начальном выполнении вы можете пропустить удаление, чтобы база данных и приложение продолжали работать:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)
PASS
ok   terraform-up-and-running   423.650s
```

Затем при каждом изменении приложения вы можете пропустить развертывание базы данных и обе стадии с удалением. Таким образом, тест выполняет лишь развертывание приложения и проверку:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  SKIP_deploy_db=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)
PASS
ok   terraform-up-and-running   13.824s
```

Это сокращает время на получение результатов с нескольких минут до нескольких секунд, что крайне положительно скажется на продуктивности.

При внесении изменений не забывайте регулярно фиксировать свой код:

```
$ git commit -m "Updated Hello, World text"
```

Подача изменений на рассмотрение

Если все работает так, как вы того ожидали, можно создать запрос на включение внесенных изменений (точно так же, как и в случае с прикладным кодом). Ваши коллеги просмотрят ваши изменения в поиске возможных ошибок, заодно исправляя их в соответствии с рекомендациями по *оформлению кода*. В ходе командной разработки, независимо от того, какой именно код вы пишете, все должны следовать общим рекомендациям. Одно из моих любимых определений чистого кода было дано в интервью, которое я взял у Ника Делламаггиора для одной из своих предыдущих книг *Hello, Startup* (http://www.hello-startup.net).

Если смотреть на один файл, написанный десятью разными инженерами, то определить, кому принадлежат те или иные фрагменты, должно быть почти невозможно. Вот что для меня означает чистый код.

Чтобы этого добиться, нужно проводить разбор кода и публиковать рекомендации по его оформлению вместе с шаблонами и идиомами языка. Когда все их изучат, продуктивность существенно повысится, так как каждый будет знать, как писать код одним и тем же образом. После этого вас будет больше заботить то, что вы пишете, а не как вы это делаете.

Ник Делламаггиор, ведущий разработчик инфраструктуры в Coursera

У каждой команды свои правила написания кода Terraform, которые подходят именно ей, поэтому я перечислю лишь те из них, которые могут пригодиться для большинства команд:

• документация;
• автоматические тесты;
• структура каталогов;
• рекомендации по оформлению кода.

Документация

В некотором смысле код Terraform сам по себе является разновидностью документации. Он использует простой язык для описания того, какая именно инфраструктура у вас развернута и как она сконфигурирована. Однако такой вещи, как самодокументируемый код, не существует. Хорошо написанный код может поведать о том, *что* он делает, но никакой из известных мне языков программирования (включая Terraform) не способен объяснить, *зачем* он это делает.

Именно поэтому любому программному обеспечению, в том числе и IaC, помимо самого кода, нужна документация. Существует несколько видов документации для выбора, которые можно сделать обязательными при разборе кода.

•*Письменная документация*. Большинство модулей Terraform должны содержать файл README, который объясняет их назначение, причину их существования и то, как их можно модифицировать. Этот файл лучше написать в первую очередь, еще до какого-либо кода Terraform. Так, прежде чем окунуться в детали реализации, вы будете помнить о том, *что* и *зачем* вы создаете61. Потратив 20 минут на написание README, вы сможете сэкономить часы, которые ушли бы на создание кода, решающего не ту проблему. Помимо этого файла, также следует позаботиться о практических руководствах, документации к API, вики-страницах и проектных документах, которые более глубоко объясняют принцип работы кода и то, почему он так написан.

• *Документация в коде*. Внутри самого кода в качестве документации можно добавлять комментарии. Terraform считает комментарием любой текст, который начинается со знака #. Не пытайтесь объяснить в

комментариях, что делает ваш код; это его работа. Предоставляйте лишь ту информацию, которую нельзя выразить в коде: например, как его следует использовать или почему было выбрано то или иное архитектурное решение. Terraform также позволяет указывать для каждой входной и выходной переменной параметр description, который отлично подходит для описания того, как эти переменные должны использоваться.

• *Примеры кода*. Как уже обсуждалось в главе 6, любой модуль Terraform должен включать в себя примеры того, как его следует применять. Это отличная возможность подчеркнуть предполагаемые сценарии использования, позволить пользователям запустить ваш модуль без написания какого-либо кода и, что самое главное, добавить автоматические тесты.

Автоматические тесты

Глава 7 полностью посвящена тестированию кода Terraform, поэтому скажу лишь, что инфраструктурный код без тестов можно считать неисправным. В связи с этим один из важнейших комментариев, которые вы можете оставить при разборе кода, звучит так: «Как вы это протестировали?»

Структура каталогов

Ваша команда должна выработать правила о том, где должен храниться код Terraform и какой должна быть структура каталогов. Поскольку структура файлов и каталогов определяет то, как Terraform хранит свое состояние, нужно особенно тщательно подумать о влиянии вашей структуры каталогов на возможность предоставления гарантий изоляции, чтобы, например, изменения в среде финального тестирования не вызвали проблем в промышленном окружении. При разборе кода важно следить за соблюдением структуры, описанной в подразделе «Изоляция с помощью описания структуры файлов» на с. 119, которая обеспечивает изоляцию разных окружений (таких как Stage и Prod) и отдельных компонентов (скажем, сетевой топологии для всей среды и отдельного приложения в этой среде).

Рекомендации по оформлению кода

Любая команда должна соблюдать определенные соглашения о стиле оформления кода, включая использование пробелов, переносов строки, отступов, фигурных скобок, имен переменных и т. д. Программисты любят поспорить о том, что лучше: пробелы или табуляция — и где должна находиться фигурная скобка, но сам выбор не так уж и важен. Важно, чтобы

ваша кодовая база была однородной. У большинства редакторов и интегрированных сред разработки есть средства форматирования кода, которые также можно использовать в хуках фиксации вашей системы управления версиями. Все это поможет соблюдать общий стиль.

У Terraform даже есть встроенная команда fmt, которая может автоматически привести код к единому стилю:

```
$ terraform fmt
```

Вы можете запускать эту команду в рамках хука фиксации, чтобы весь код, который вы сохраняете в системе управления версиями, был написан в одном стиле.

Выполнение автоматических тестов

У инфраструктурного кода, как и у прикладного, должны быть хуки, которые запускают автоматические тесты в CI-сервере после каждой фиксации и отображают их результаты на странице запроса на включение внесенных изменений. Вы уже видели в главе 7, как пишутся модульные, интеграционные и сквозные тесты для кода Terraform. Но есть еще одна крайне важная проверка, которую вы должны сделать: terraformplan. Здесь действует простое правило: прежде чем применять изменения, всегда выполняйте команду plan.

Terraform автоматически отображает вывод команды plan, когда вы выполняете apply, поэтому данное правило означает, что вы должны остановиться и прочитать этот вывод! Вы не поверите, какие ошибки можно предотвратить, потратив 30 секунд на анализ расхождений, которые выводит apply. Чтобы поощрять это поведение, команду plan можно интегрировать в процесс разбора кода. Например, открытый инструмент под названием Atlantis (https://www.runatlantis.io/) автоматически выполняет terraformplan при фиксации кода и добавляет вывод plan в виде комментариев к запросам на включение внесенных изменений, как показано на рис. 8.3.

Более того, команда plan позволяет сохранять свой вывод в файл:

```
$ terraform plan -out=example.plan
```

Затем для этого файла можно выполнить команду apply, чтобы она применила именно те изменения, которые вы видели в начале:

```
$ terraform apply example.plan
```

Обратите внимание на то, что, как и состояние Terraform, файлы с планом могут содержать конфиденциальные данные. Например, если вы используете Terraform для развертывания базы данных, в файле с планом может находиться пароль к ней. Поскольку эти файлы не шифруются, для их сколь-либо продолжительного хранения необходимо предусмотреть собственное шифрование.

Слияние и выпуск новой версии

После того как члены вашей команды разобрали внесенные изменения и вывод команды plan и все ваши тесты были успешно пройдены, можно объединить свой код с веткой master и выпустить новую версию. Как и с прикладным кодом, для этого подходят теги Git:

$ git tag -a "v0.0.6" -m "Updated hello-world-example text"

$ git push --follow-tags

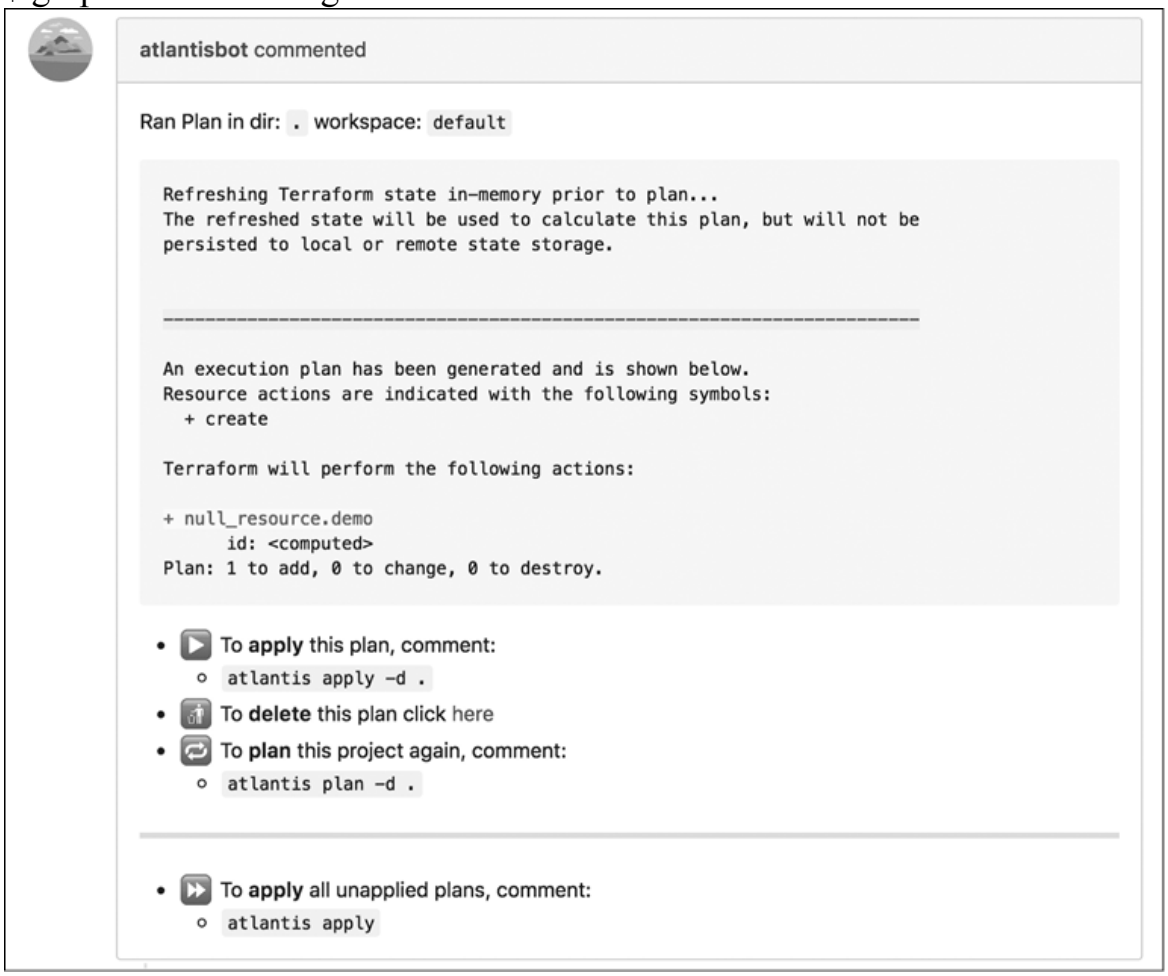


Рис. 8.3. Atlantis умеет автоматически добавлять вывод команды terraform plan в виде комментариев к запросам на включение внесенных изменений

Отличие в том, что прикладной код часто развертывается в виде отдельного артефакта, такого как образ Docker или ВМ, тогда как Terraform умеет самостоятельно загружать код из Git. Таким образом, репозиторий с заданным тегом сам по себе неизменяемый артефакт с поддержкой версионирования, и именно он будет развертываться.

Развертывание

Получив неизменяемый артефакт с поддержкой версионирования, вы можете его развернуть. Вот несколько ключевых моментов, которые следует учитывать при развертывании кода Terraform:

• инструментарий для развертывания;
• стратегии развертывания;
• сервер для развертывания;
• продвижение артефакта по разным окружениям.

Инструментарий для развертывания

Основным средством для развертывания кода Terraform является сам Terraform. Но есть также несколько других инструментов, которые могут пригодиться.

•*Atlantis*. Вы уже видели этот открытый инструмент. Он умеет не только добавлять вывод команды plan в ваши запросы на сохранение внесенных изменений, но и запускать terraformapply в ответ на добавление в ваш запрос специального комментария. За счет этого вы получаете удобный веб-интерфейс для развертывания Terraform. Однако имейте в виду, что он не поддерживает версионирование. Это может затруднить поддержку и отладку крупных проектов.

• *Terraform Enterprise*. Корпоративные продукты от компании HashiCorp имеют веб-интерфейс, с помощью которого можно выполнять команды terraformplan и terraformapply, а также управлять переменными, конфиденциальными данными и правами доступа.

• *Terragrunt*. Это открытая обертка вокруг Terraform, которая заполняет собой некоторые пробелы в данной системе. Чуть позже в этой главе вы увидите, как с ее помощью в разных средах можно развернуть версионируемый код Terraform, минимизировав его дублирование.

• *Скрипты*. Вы можете сделать работу с Terraform более гибкой за счет скриптов, написанных на языках программирования общего назначения, таких как Python, Ruby или bash.

Стратегии развертывания

Система Terraform сама по себе не предлагает никаких стратегий развертывания. У нее нет встроенной поддержки скользящих или «сине-зеленых» обновлений, и она не позволяет использовать ротацию функций для большинства изменений (это, например, касается баз данных; вы либо вносите изменение, либо нет). Вам, в сущности, доступна лишь команда terraformapply, которая просто выполняет конфигурацию, предусмотренную в вашем коде. Конечно, иногда в самом коде можно реализовать

собственную стратегию развертывания; вспомните, скажем, скользящие обновления с нулевым временем простоя в модуле asg-rolling-deploy, который вы создали в предыдущих главах. Но, поскольку Terraform является декларативным языком, ваш контроль за развертываниями довольно ограничен.

Исходя из этих ограничений крайне важно быть готовыми к ситуациям, когда развертывание проходит неудачно. При развертывании приложения от многих видов ошибок спасает выбранная вами стратегия. Например, если приложению не удается пройти проверку работоспособности, балансировщик нагрузки никогда не направит к нему реальный трафик, поэтому пользователи не пострадают. Более того, в случае возникновения ошибок стратегии со скользящими или «сине-зелеными» обновлениями могут автоматически откатываться к предыдущей версии приложения.

Для сравнения, Terraform *не поддерживает автоматический откат в ответ на ошибки*. Это отчасти связано с тем, что в случае с произвольным инфраструктурным кодом это часто оказывается небезопасным или даже невозможным. Например, если приложение не удалось развернуть, его почти всегда можно откатить к более старой версии, но если неудачным оказалось развертывание изменения в коде Terraform, которое должно было удалить базу данных или остановить сервер, откатить его будет не так просто!

Более того, как вы уже видели в разделе «Подводные камни Terraform» на с. 203, ошибки в Terraform далеко не редкость. Поэтому ваша стратегия развертывания должна считать их (относительно) нормальным явлением и уметь на них как следует реагировать.

•*Повторные попытки*. Некоторые ошибки в Terraform временные и сами исчезают при повторном выполнении terraformapply. Инструментарий для развертывания, который вы используете вместе с Terraform, должен обнаруживать известные проблемы и автоматически повторять операцию после небольшой паузы. Автоматическое повторение попыток в ответ на известные ошибки встроено в Terragrunt.

• *Ошибки состояния Terraform*. Время от времени Terraform не удается сохранить состояние при выполнении terraformapply. Предположим, если в ходе работы apply теряется соединение с Интернетом, неудачей завершится не только эта команда, но и попытка записать обновленный файл состояния в удаленное хранилище (такое как Amazon S3). В таких случаях Terraform сохраняет состояние на диск в файл под названием errored.tfstate. Убедитесь, что ваш CI-сервер не удаляет эти файлы (скажем, в процессе очистки рабочей области после сборки)! Если после неудачного развертывания у вас все еще остается доступ к этому файлу, при возобновлении интернет-

соединения вы можете загрузить его в удаленное хранилище (например, в S3), используя команду statepush. Таким образом, информация о состоянии не будет утеряна:

$ terraform state push errored.tfstate

• *Ошибки снятия блокировки*. Иногда Terraform не удается снять блокировку. Например, если ваш CI-сервер выйдет из строя в ходе выполнения terraformapply, состояние так и останется заблокированным. Любой, кто попытается выполнить apply для того же модуля, получит сообщение об ошибке, в котором будет сказано, что состояние заблокировано, и указан идентификатор блокировки. Если вы совершенно точно уверены, что эта блокировка оставлена по случайности, вы можете принудительно ее снять с помощью команды force-unlock, передав ее ID из полученного сообщения об ошибке:

$ terraform force-unlock <LOCK_ID>

Сервер развертывания

Все изменения в инфраструктурном коде, как и в прикладном, должны применяться с CI-сервера, а не с компьютера разработчика. Вы можете запускать команду terraformapply из Jenkins, CircleCI, Terraform Enterprise, Atlantis или с любой другой автоматизированной платформы с достаточным уровнем безопасности. Вы получаете те же преимущества, что и с прикладным кодом: это заставляет вас полностью автоматизировать ваш процесс развертывания, гарантирует, что этот процесс выполняется всегда из одного и того же окружения, и позволяет лучше контролировать доступ к промышленным средам.

Тем не менее инфраструктурный код требует чуть более сложной организации прав на развертывание, чем прикладной. CI-серверу обычно можно выдать минимальный набор прав для развертывания приложений. Например, чтобы развернуть прикладной код в ASG, CI-серверу, как правило, требуется лишь несколько конкретных прав ec2 и autoscaling. Однако для развертывания произвольных изменений в инфраструктурном коде (скажем, код Terraform может попытаться развернуть базу данных, VPC или совершенно новую учетную запись AWS) CI-сервер должен обладать произвольными, то есть администраторскими полномочиями.

Поскольку CI-серверы предназначены для выполнения произвольного кода, их сложно сделать безопасными (попробуйте подписаться на консультативную рассылку Jenkins, чтобы увидеть, насколько часто в нем объявляют о серьезных уязвимостях), поэтому давать им постоянные администраторские полномочия будет рискованно. Для минимизации этого риска можно предпринять несколько шагов.

• Не делайте свой CI-сервер доступным из публичного Интернета. То есть размещайте его в приватных подсетях, без публичного IP-адреса, чтобы он был доступен только по VPN-соединению. Это существенно повысит безопасность, но создаст дополнительные ограничения: у вас перестанут работать веб-хуки для внешних систем, в результате чего, к примеру, GitHub не сможет автоматически инициировать сборку на вашем CI-сервере. Вместо этого нужно сделать так, чтобы ваш CI-сервер сам обращался за обновлениями к системе управления версиями.

• Ограничьте доступ к CI-серверу. Разрешите обращаться к нему только по HTTP, требуйте от пользователей прохождения аутентификации и следуйте рекомендациям по укреплению серверов (например, предусмотрите строгие правила для брандмауэра, установите fail2ban, включите ведение журнала аудита и т. д.).

• Старайтесь не выдавать CI-серверу постоянные администраторские полномочия. Он должен иметь права, достаточные для автоматического развертывания определенных изменений в инфраструктурном коде. Но когда речь идет о чем-то более деликатном (например, о добавлении/удалении пользователей или доступе к конфиденциальным данным), соответствующие временные полномочия (срок действия которых, скажем, истекает через один час) серверу должен предоставить живой системный администратор.

Продвижение артефактов по разным окружениям

Инфраструктурные артефакты, как и прикладные, должны быть неизменяемыми, поддерживать версионирование и продвигаться от одного окружения к другому; например, версию v0.0.6 нужно продвинуть из Dev в Stage, а затем в Prod62. Здесь тоже действует простое правило: *всегда тестируйте изменения в коде Terraform, прежде чем развертывать их в промышленной среде*.

Поскольку в Terraform все и так автоматизировано, проверка изменений в Stage перед выкатыванием их в Prod не требует особых дополнительных усилий, но при этом позволяет выявить огромное количество ошибок. Тестирование в предпромышленном окружении особенно важно по той причине, что, как уже упоминалось в этой главе, Terraform не откатывает изменения в случае ошибки. Если при выполнении terraformapply что-то пойдет не так, придется чинить это самостоятельно. Данный процесс будет менее трудоемким и напряженным, если проводить его еще до развертывания в промышленной среде.

Процесс продвижения кода Terraform по разным окружениям аналогичен продвижению прикладных артефактов, только у него есть

дополнительный этап: выполнение команды terraformplan и ручная проверка ее вывода. В случае с приложениями этот этап обычно не требуется, так как развертывание прикладного кода чаще всего проходит по одному сценарию и не несет в себе большого риска. Однако каждое развертывание инфраструктуры может быть уникальным, а ошибки, которые при этом возникают (вроде удаления базы данных), могут стоить очень дорого. Поэтому возможность в последний раз просмотреть и разобрать вывод plan, безусловно, стоит потраченного времени.

Так, к примеру, выглядит процесс продвижения модуля Terraform версии v0.0.6 по окружениям Dev, Stage и Prod.

1. Обновить среду Dev до v0.0.6 и выполнить terraformplan.
2. Запросить разбор и одобрение плана. Например, послать автоматическое сообщение по Slack.
3. Если план одобрен, развернуть версию v0.0.6 в Dev с помощью terraformapply.
4. Выполнить в Dev ручное и автоматическое тестирование.
5. Если версия v0.0.6 хорошо работает в Dev, повторить шаги 1–4, чтобы продвинуть ее в Stage.
6. Если версия v0.0.6 хорошо работает в Stage, повторить шаги 1–4, чтобы продвинуть ее в Prod.

Осталось разобраться с одной важной проблемой: дублированием кода между разными окружениями в репозитории *текущей* инфраструктуры. Например, взгляните на репозиторий на рис. 8.4.

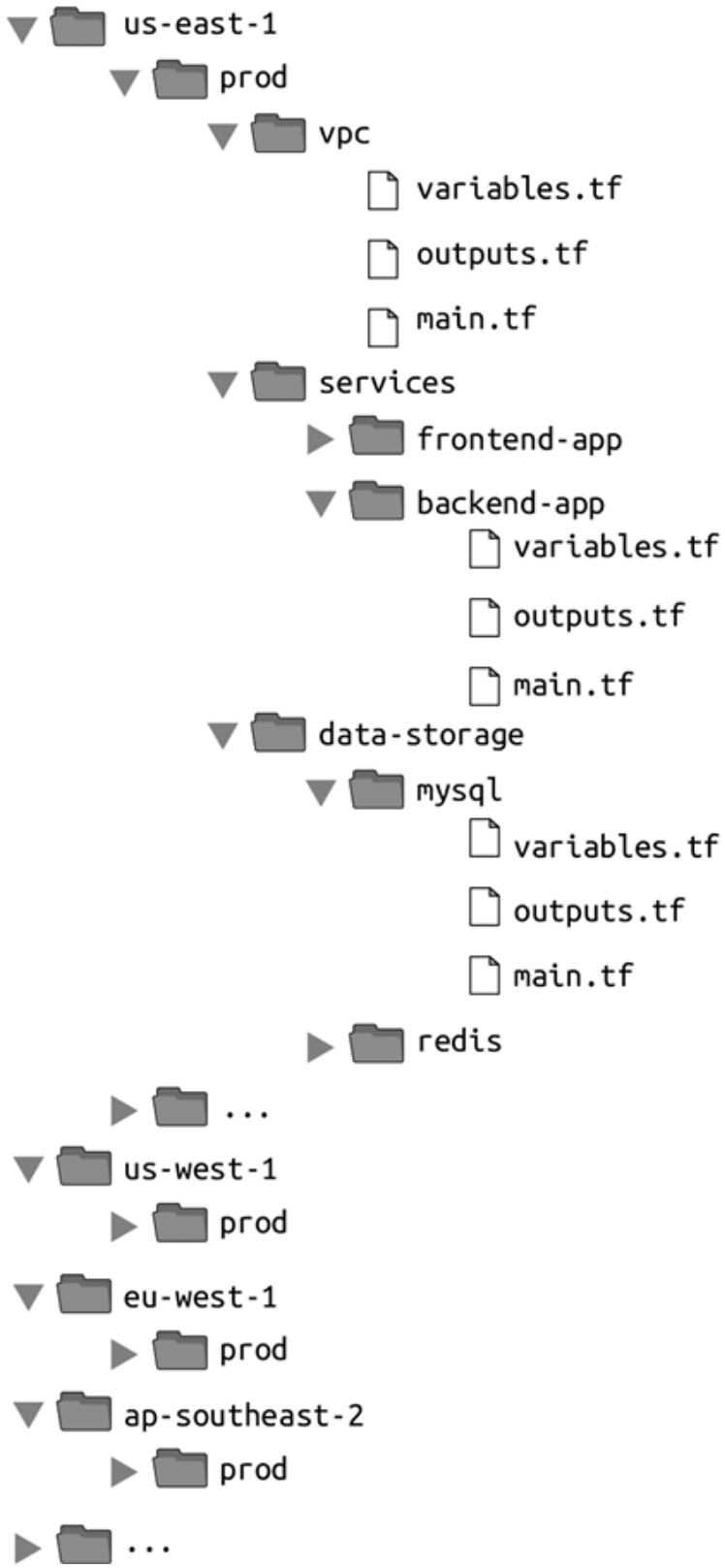


Рис. 8.4. Структура каталогов при активном копировании сред и модулей внутри каждой среды

Репозиторий текущей инфраструктуры содержит большое количество регионов с множеством модулей в каждом из них. И большинство этих модулей были скопированы. Конечно, у каждого из них имеется файл main.tf, который ссылается на модуль в вашем репозитории modules, поэтому ситуация с дублированием могла быть и хуже. Но, даже если вы просто создаете экземпляр одного модуля, вам все равно приходится использовать большой объем шаблонного кода, одинакового для всех окружений:

- конфигурацию provider;
- конфигурацию backend;
- задание всех входных переменных для модуля;
- задание всех выходных переменных, которые возвращает модуль.

В результате в каждом вашем модуле могут накопиться десятки и сотни строчек в основном идентичного кода, который копируется из одной среды в другую. Чтобы минимизировать дублирование по принципу DRY и облегчить продвижение кода Terraform по разным средам, вы можете использовать открытый инструмент под названием Terragrunt (https://github.com/gruntwork-io/terragrunt), о котором я уже упоминал ранее. Terragrunt представляет собой тонкую обертку вокруг Terraform. После его установки (инструкции по установке — в файле README по адресу https://github.com/gruntwork-io/terragrunt#install-terragrunt) вы сможете выполнять стандартные команды terraform, применяя исполняемый файл terragrunt:

```
$ terragrunt plan
$ terragrunt apply
$ terragrunt output
```

Terragrunt запускает Terraform с заданной вами командой, но с некоторым дополнительным поведением на основе конфигурации, указанной в файле terragrunt.hcl. Основная идея в том, что весь код Terraform находится в единственном экземпляре в репозитории modules, тогда как репозиторий *текущей* инфраструктуры содержит файлы terragrunt.hcl, которые позволяют конфигурировать и развертывать каждый модуль в любой среде без дублирования. Это проиллюстрировано на рис. 8.5.

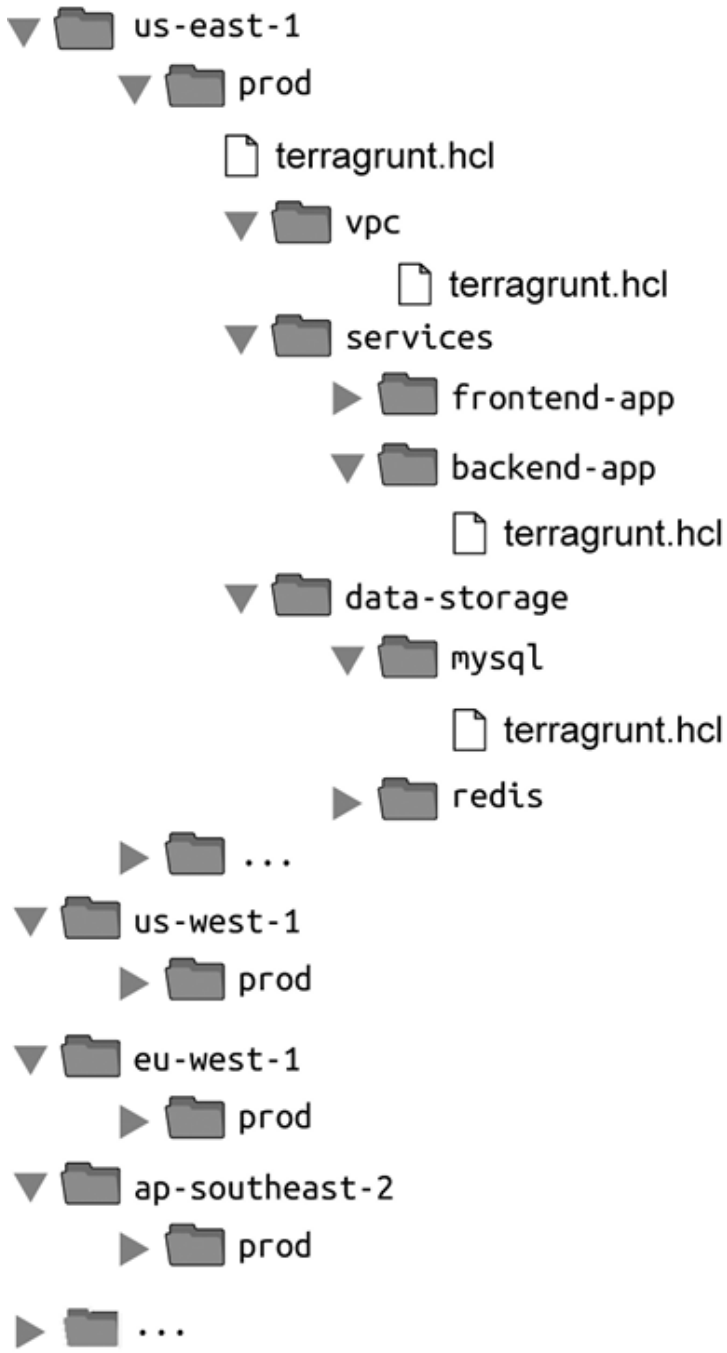


Рис. 8.5. Структура файлов и каталогов при использовании Terragrunt

Для начала добавьте конфигурацию provider в файлы modules/data-stores/mysql/main.tf и modules/services/hello-world-app/main.tf:

```
provider "aws" {
  region = "us-east-2"
  # Разрешаем любую версию провайдера AWS вида 2.x
  version = "~> 2.0"
}
```

Затем добавьте конфигурацию backend в файлы modules/data-stores/mysql/main.tf и modules/services/hello-world-app/main.tf, но оставьте блок config пустым (вскоре вы увидите, как наполнить его с помощью Terragrunt):

```
terraform {
  # Подойдет любая версия Terraform
  # вида 0.12.x
  required_version = ">= 0.12, < 0.13"
  # Частичная конфигурация. Остальное
  # добавит Terragrunt.
  backend "s3" {}
}
```

Зафиксируйте эти изменения и выпустите новую версию своего репозитория modules:

```
$ git add modules/data-stores/mysql/main.tf
$ git add modules/services/hello-world-app/main.tf
$ git commit -m "Update mysql and hello-world-app for Terragrunt"
$ git tag -a "v0.0.7" -m "Update Hello, World text"
$ git push --follow-tags
```

Теперь перейдите в репозиторий текущей инфраструктуры и удалите все файлы .tf. Этот код Terraform, который вы скопировали ранее, будет заменен файлом terragrunt.hcl для каждого модуля. Например, так выглядит terragrunt.hcl для live/stage/data-stores/mysql/terragrunt.hcl:

```
terraform {
  source = "github.com/<OWNER>/
  modules//data-stores/
  mysql?ref=v0.0.7"
}
inputs = {
  db_name     = "example_stage"
  db_username = "admin"
        # Установите пароль с помощью переменной среды
TF_VAR_db_password
}
```

Как видите, файлы terragrunt.hcl используют тот же синтаксис HCL (HashiCorp Configuration Language), что и Terraform. Когда вы запустите команду terragruntapply, она найдет в файле terragrunt.hcl параметр source и сделает следующее.

1. Загрузит во временную папку код, находящийся по заданному URL-адресу. Здесь поддерживается тот же синтаксис URL-адресов, что и в

параметре source модулей Terraform. Поэтому вы можете использовать локальные пути, обычные и версионированные адреса Git (с помощью параметра ref, как показано в предыдущем примере) и т. д.

2. Выполнит во временной папке команду terraformapply, передав ей входные переменные, которые вы указали в блоке inputs={…}.

Преимущество такого подхода в том, что весь код каждого модуля в репозитории *текущей* инфраструктуры сводится к единственному файлу terragrunt.hcl, который содержит указатель на используемый модуль (с указанием определенной версии) и входные переменные, установленные для определенной среды. Таким образом, мы следуем принципу DRY настолько строго, насколько это возможно.

Terragrunt также позволяет избавиться от дублирования конфигурации backend. Чтобы не указывать bucket, key, dynamodb_table и другие параметры в каждом модуле, вы можете разместить их в файле terragrunt.hcl для той или иной среды. Например, создайте в файле live/stage/terragrunt.hcl следующую конфигурацию:

```
remote_state {
  backend = "s3"
  config = {
    bucket         = "<YOUR BUCKET>"
    key            = "${path_relative_to_include()}/terraform.tfstate"
    region         = "us-east-2"
    encrypt        = true
    dynamodb_table = "<YOUR_TABLE>"
  }
}
```

Конфигурация backend размещается в блоке remote_state как обычно, только для значения key используется встроенная в Terragrunt функция под названием path_relative_to_include(). Эта функция возвращает относительный путь между этим корневым файлом terragrunt.hcl и любым дочерним модулем, который его подключает. Например, чтобы подключить этот корневой файл в live/stage/data-stores/mysql/terragrunt.hcl, просто добавьте блок include:

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"
}
include {
  path = find_in_parent_folders()
}
inputs = {
```

```
  db_name     = "example_stage"
  db_username = "admin"
      # Установите пароль с помощью переменной среды TF_VAR_db_password
}
```

Блок include находит корневой файл terragrunt.hcl с помощью функции find_in_parent_folders(), встроенной в Terragrunt, и автоматически наследует все его параметры, включая конфигурацию remote_state. В результате этот модуль mysql будет использовать те же настройки backend, что и корневой файл, а полю key будет автоматически присвоено значение data-stores/mysql/terraform.tfstate. Это означает, что для хранения состояния Terraform будет применяться та же структура каталогов, что и для репозитория текущей инфраструктуры, благодаря чему вам будет проще разобраться в том, какие файлы состояния сгенерировал тот или иной модуль.

Чтобы развернуть этот модуль, выполните terragruntapply:

```
$ terragrunt apply
[terragrunt] Reading Terragrunt config file at terragrunt.hcl
[terragrunt] Downloading Terraform configurations from
        github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7
[terragrunt] Running command: terraform init -backend-config=(...)
(...)
[terragrunt] Running command: terraform apply
(...)
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

В этом выводе видно, как Terragrunt считывает ваш файл terragrunt.hcl, загружает заданный вами модуль, запускает terraforminit, чтобы сконфигурировать ваш блок backend (он даже создаст для вас бакет S3 и таблицу DynamoDB, если их еще не существует), и затем разворачивает все это с помощью команды terraformapply.

Теперь вы можете развернуть модуль hello-world-app в среде Staging, добавив файл live/stage/services/hello-world-app/terragrunt.hcl и запустив terragruntapply:

```
terraform {
    source = "github.com/<OWNER>/modules//services/hello-world-app?ref=v0.0.7"
}
include {
  path = find_in_parent_folders()
}
```

```
inputs = {
  environment = "stage"
  min_size = 2
  max_size = 2
  enable_autoscaling = false
  db_remote_state_bucket = "<YOUR_BUCKET>"
  db_remote_state_key    = "<YOUR_KEY>"
}
```

Этот модуль также использует блок include, чтобы взять все параметры из корневого файла terragrunt.hcl. Таким образом, он наследует все то же содержимое backend, кроме поля key, которому, как можно было бы ожидать, автоматически присваивается services/hello-world-app/terraform.tfstate. Если в среде Staging все работает хорошо, вы можете создать аналогичные файлы terragrunt.hcl в папке live/prod и продвинуть ту же версию артефакта v0.0.7 в промышленную среду, выполнив в каждом модуле команду terragruntapply.

Собираем все вместе

Вы познакомились с тем, как прикладной и инфраструктурный код проходят свой путь от стадии разработки до промышленной среды. В табл. 8.1 приводится сравнение этих двух рабочих процессов.

Таблица 8.1. Рабочие процессы для доставки прикладного и инфраструктурного кода

Процесс	**Прикладной код**	**Инфраструктурный код**
Использование системы управления версиями	git clone	По одному репозиторию на приложение.
Используйте ветки	git clone	Репозитории live и modules.
Не используйте ветки	Локальное выполнение кода	Выполняйте на локальном компьютере.
ruby web-server.rb	ruby web-server-test.rb	Выполняйте в изолированной среде.
terraform apply		

go test
Внесение изменений в код
Отредактируйте код.
ruby web-server.rb
ruby web-server-test.rb
Отредактируйте код.
terraform apply
go test
Используйте стадии тестирования
Подача изменений на рассмотрение
Подайте запрос на сохранение внесенных изменений.
Убедитесь, что соблюдаются рекомендации по написанию кода
Подайте запрос на сохранение внесенных изменений.
Убедитесь, что соблюдаются рекомендации по написанию кода
Выполнение автоматических тестов
Тесты выполняются на CI-сервере.
Модульные тесты.
Интеграционные тесты.
Сквозные тесты.
Статический анализ
Тесты выполняются на CI-сервере.
Модульные тесты.
Интеграционные тесты.
Сквозные тесты.
Статический анализ.
terraform plan
Слияние и выпуск новой версии
git tag
Создайте неизменяемый артефакт с поддержкой версионирования
git tag
Используйте репозиторий в качестве неизменяемого артефакта с поддержкой версионирования
Развертывание
Развертывайте с помощью Terraform, средства оркестрации (такого как Kubernetes, Mesos), скриптов.
Множество стратегий развертывания: скользящие, «сине-зеленые», канареечные обновления.
Выполняйте развертывание на CI-сервере.
Выделите CI-серверу ограниченные права доступа.

Продвигайте неизменяемые артефакты с поддержкой версионирования по разным средам

Развертывайте с помощью Terraform, Atlantis, Terraform Enterprise, Terragrunt, скриптов.

Ограниченный набор стратегий развертывания. Не забудьте об обработке ошибок: повторные попытки, errored.tfstate!

Выполняйте развертывание на CI-сервере.

Выделите CI-серверу администраторские права доступа.

Продвигайте неизменяемые артефакты с поддержкой версионирования по разным средам

Следуя этому процессу, вы сможете выполнить и протестировать свой прикладной и инфраструктурный код, провести его разбор, упаковать его в неизменяемые артефакты с поддержкой версионирования и затем продвинуть эти артефакты от одного окружения к другому, как показано на рис. 8.6.

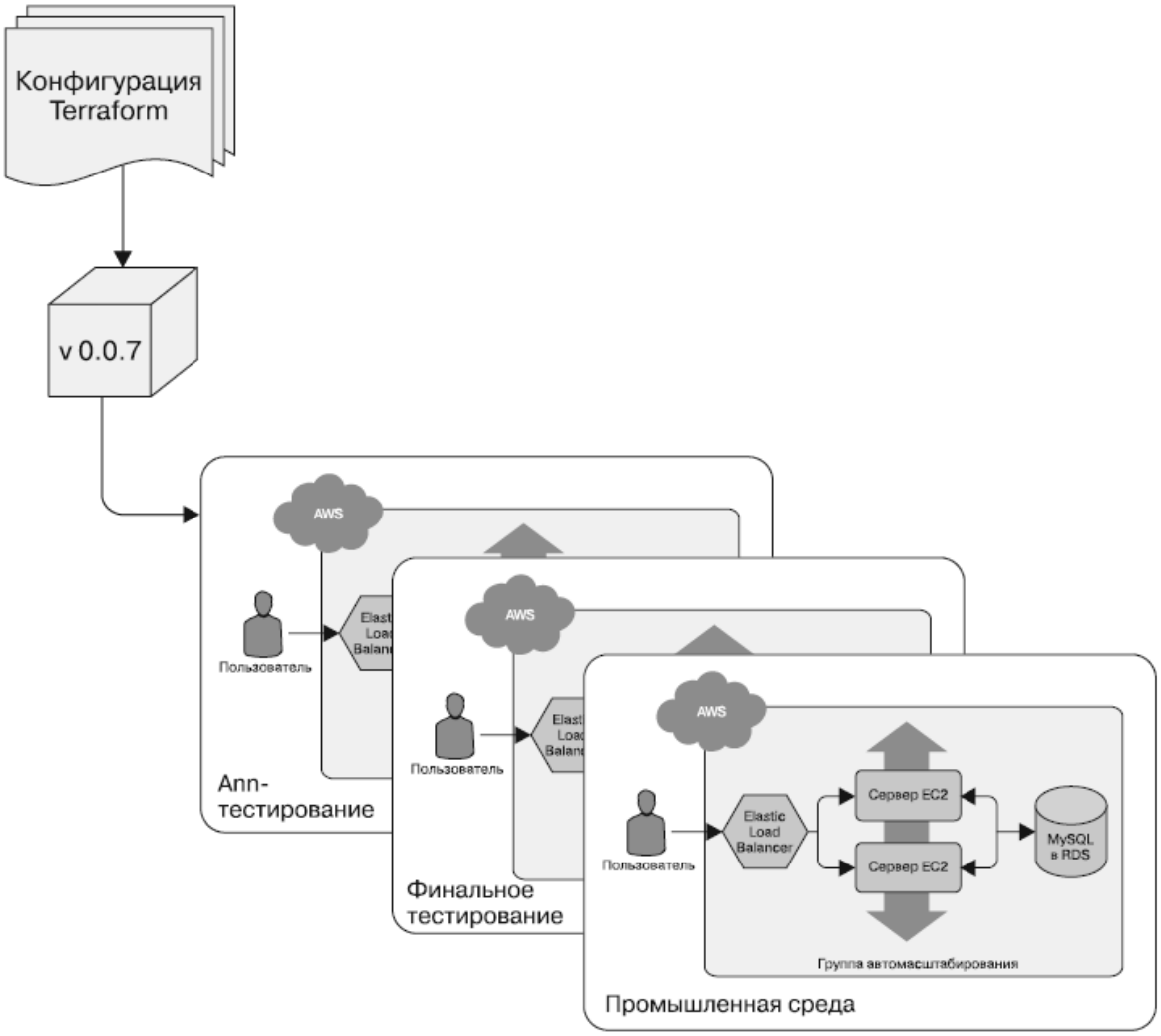


Рис. 8.6. Продвижение неизменяемого версионируемого артефакта с кодом Terraform от одной среды к другой

Резюме

Если вы дочитали до этого места, у вас есть все необходимые знания для использования Terraform в реальных условиях, включая написание кода, управление состоянием, создание универсальных модулей, эмуляцию циклов и условных выражений, выполнение развертывания, создание кода промышленного уровня, тестирование и работу с Terraform в команде. Мы рассмотрели примеры доставки и администрирования серверов, кластеров, балансировщиков нагрузки, баз данных, планируемых действий, оповещений CloudWatch, пользователей IAM, универсальных модулей, развертываний с нулевым временем простоя, автоматических тестов и т. д. Уф! Главное, когда закончите, не забудьте сделать terraformdestroy для каждого модуля.

В случае корректного применения Terraform ваша команда сможет быстрее развертывать свой код и реагировать на изменения. Надеюсь, процесс развертывания станет для вас рутинным и скучным, что в мире системного администрирования считается большим плюсом. Если вы делаете свою работу как следует, а не тратите все время на ручное управление ресурсами, ваша команда сможет уделять все больше внимания улучшению инфраструктуры, что позволит вам работать еще быстрее.

На этом книга подходит к концу, но ваше путешествие в мир Terraform только начинается. Чтобы узнать больше о Terraform, IaC и DevOps, переходите дальше — к приложению, в котором содержится список рекомендуемой литературы. Если вы хотите оставить отзыв или задать вопрос, обязательно свяжитесь со мной по адресу jim@ybrikman.com. Спасибо за прочтение!

58 The Standish Group, CHAOS Manifesto 2013: Think Big, Act Small. 2013. www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf.

59 *Milstein D.* How To Survive a Ground-Up Rewrite Without Losing Your Sanity // OnStartups.com, 8 апреля 2013 года. onstartups.com/tabid/3339/bid/97052/How-To-Survive-a-Ground-Up-Rewrite-Without-LosingYour-Sanity.aspx.

60 GruntworkInfrastructureasCodeLibrary (gruntwork.io/infrastructure-as-code-library/) — это коллекция более чем из 300 000 строк универсального инфраструктурного кода промышленного уровня с коммерческой поддержкой, который был проверен в реальных условиях сотнями компаний.

61 Написание README в первую очередь называется Readme-ориентированной разработкой (bit.ly/1p8QBor).

62 Идея продвижения кода Terraform по разным средам принадлежит Кифу Моррису: Using Pipelines to Manage Environments with Infrastructure as Code (bit.ly/2lJmus8).

Приложение. Дополнительные ресурсы

Ниже перечислены одни из лучших ресурсов о DevOps и IaC, которые мне удалось найти, включая книги, статьи, информационные рассылки и доклады.

Книги

•*Morris K.* Infrastructure as Code: Managing Servers in the Cloud. — O'Reilly, 2016.

• *Бейер Б., Джоунс К., Петофф Д., Мёрфи Н.Р.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2019.

• *Хамбл Д., Уиллис Д., Дебуа П., Ким Д.* Руководство по DevOps. Как добиться гибкости, надежности и безопасности мирового уровня в технологических компаниях. — М.: Манн, Иванов и Фербер, 2018.

• *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2019.

• *Humble J., Farley D.* Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation. — Addison-Wesley Professional, 2010.

• *Нейгард М.* Release it! Проектирование и дизайн ПО для тех, кому не все равно. — СПб.: Питер, 2016.

• *Лукша М.* Kubernetes в действии Action. — М.: ДМК-Пресс, 2019.

• *Gruver G., Mouser T.* Leading the Transformation: Applying Agile and DevOps Principles at Scale. — IT Revolution Press, 2015.

• *Behr K., Kim G., SpaffordG.* Visible Ops Handbook. — Information Technology Process Institute, 2004.

• *Дэвис Д., Дэниелс К.* Философия DevOps. Искусство управления IT. — СПб.: Питер, 2017.

• *Humble J., Molesky J., O'Reilly B.* Lean Enterprise. — O'Reilly, 2014.

• *Brikman Y.* Hello, Startup: A Programmer's Guide to Building Products, Technologies and Teams. — O'Reilly, 2015.

Блоги

• High Scalability (highscalability.com).

• Code as Craft (codeascraft.com).

• dev2ops (dev2ops.org).
• Блог AWS (aws.amazon.com/blogs/aws).
• Kitchen Soap (www.kitchensoap.com).
• Блог Пола Хамманта (paulhammant.com).
• Блог Мартина Фаулера (martinfowler.com/bliki).
• Блог Gruntwork (blog.gruntwork.io).
• Блог Евгения Брикмана (www.ybrikman.com/writing).

Лекции

• Евгений Брикман. *Reusable, composable, battle-tested Terraform modules* (bit.ly/32b28JD).
• Евгений Брикман. *5 Lessons Learned From Writing Over 300,000 Lines of Infrastructure Code* (bit.ly/2ZCcEfi).
• Евгений Брикман. *Infrastructure as code: running microservices on AWS using Docker, Terraform and ECS* (bit.ly/30TYaVu).
• Евгений Брикман. *Agility Requires Safety* (bit.ly/2YJuqJb).
• Джез Хамбл. *Adopting Continuous Delivery* (youtu.be/ZLBhVEo1OG4).
• Майкл Рембетси и Патрик Макдоннел. *Continuously Deploying Culture* (vimeo.com/51310058).
• Джон Оллспой и Пол Хаммонд. *10+ Deploys Per Day: Dev and Ops Cooperation at Flickr* (youtu.be/LdOe18KhtT4).
• Рэйчел Потвин. *Why Google Stores Billions of Lines of Code in a Single Repository* (youtu.be/W71BTkUbdqE).
• Рич Хикки. *The Language of the System* (youtu.be/ROor6_NGIWU).
• Бен Кристенсен. *Don't Build a Distributed Monolith* (youtu.be/-czp0Y4Z36Y).
• Глен Вандербург. *Real Software Engineering* (youtu.be/NP9AIUT9nos).

Информационные рассылки

• DevOps Weekly (www.devopsweekly.com).
• DevOpsLinks (devopslinks.com).
• Gruntwork Newsletter (www.gruntwork.io/newsletter).
• Terraform: Up & Running Newsletter (bit.ly/32dnRAW).

Онлайн-форумы

• Terraform Google Group (bit.ly/2L5mazu).
• DevOps subreddit (www.reddit.com/r/devops).

Об авторе

Евгений (Джим) Брикман обожает программировать, писать и путешествовать. Он соучредитель компании Gruntwork, которая

предоставляет DevOps в качестве услуги, и автор еще одной книги, опубликованной издательством O'Reilly Media под названием *Hello, Startup: A Programmer's Guide to Building Products, Technologies and Teams*. Ранее он работал программистом в LinkedIn, TripAdvisor, Cisco Systems и Thomson Financial. Свои степени бакалавра и магистра наук он получил в Корнеллском университете. Больше о нем можно узнать по адресу ybrikman.com.

Об обложке

На обложке этой книги изображен летучий дракон *(Draco volans)* — небольшая рептилия, получившая имя из-за способности планировать в воздухе за счет широких кожистых складок, натянутых между шестью «ложными» ребрами. Летучий дракон водится во многих странах Юго-Восточной Азии, включая Индонезию, Вьетнам, Таиланд, Филиппины и Сингапур.

Летучий дракон питается насекомыми и может достигать 20 сантиметров в длину. Он в основном обитает в лесистых регионах, планируя с дерева на дерево в поисках добычи и держась подальше от хищников. Самки спускаются с деревьев только для того, чтобы отложить яйца в скрытых норах в земле. Самцы ревностно охраняют свою территорию, преследуя соперников по деревьям.

Когда-то летучие драконы считались ядовитыми, но в реальности они не представляют опасности для человека и иногда выступают в роли домашних животных.

Многие животные на обложках издательства O'Reilly находятся под угрозой исчезновения; все они важны для нашей планеты. О том, как им помочь, можно узнать на сайте animals.oreilly.com.