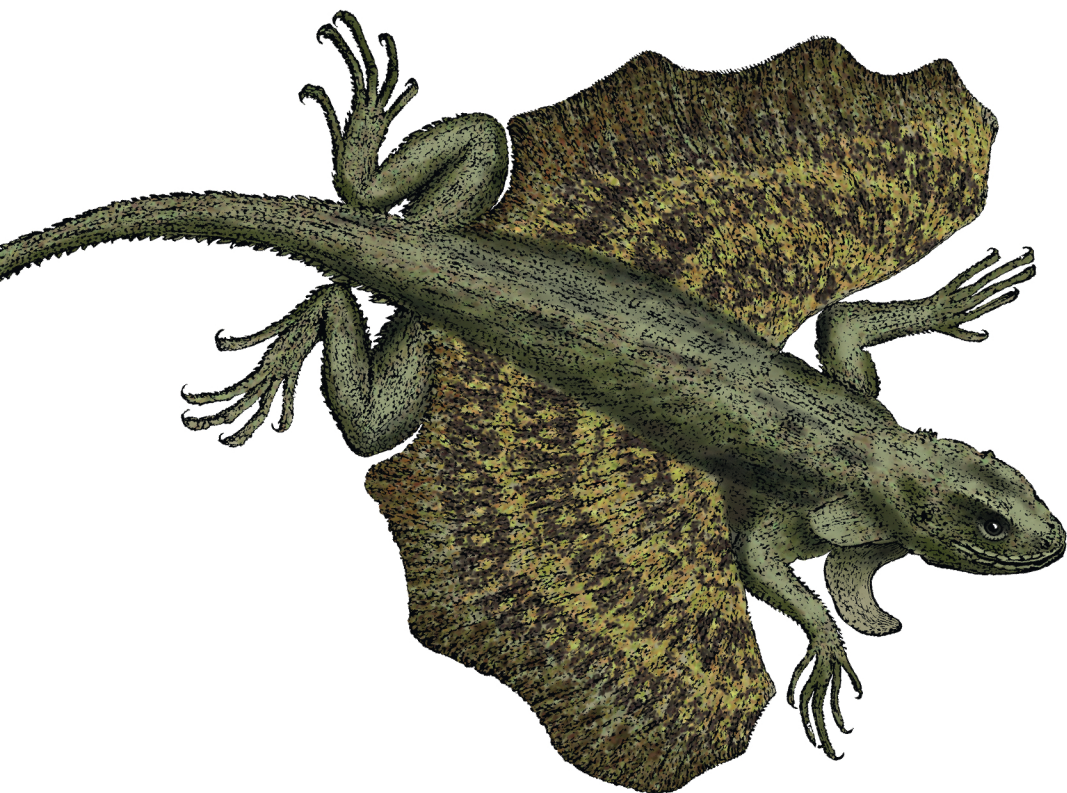


O'REILLY®

Третье
Издание



Terraform

инфраструктура на уровне кода

SPRINT
book

Евгений Брикман

THIRD EDITION

Terraform: Up & Running

Writing Infrastructure as Code

Yevgeniy Brikman

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Terraform

инфраструктура на уровне кода

ТРЕТЬЕ ИЗДАНИЕ

Евгений Брикман

SPRINT 2024
book

ББК 32.973
УДК 004
Б87

Брикман Евгений

Б87 Terraform: инфраструктура на уровне кода. 3-е межд. изд. — Астана: «Спринт Бук», 2024. — 464 с.: ил.

ISBN 978-601-08-3642-6

Terraform — настоящая звезда в мире DevOps. Эта технология позволяет управлять облачной инфраструктурой как кодом (IaC) на облачных платформах и платформах виртуализации, включая AWS, Google Cloud, Azure и др. Третье издание было полностью переработано и дополнено, чтобы вы могли быстро начать работу с Terraform.

Евгений (Джим) Брикман знакомит вас с примерами кода на простом декларативном языке программирования Terraform, иллюстрирующими возможность развертывания инфраструктуры и управления ею с помощью команд. Умудренные опытом системные администраторы, инженеры DevOps и начинающие разработчики быстро перейдут от основ Terraform к использованию полного стека, способного поддерживать трафик огромного объема и большую команду разработчиков.

ББК 32.973
УДК 004

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1098116743 англ.

Authorized Russian translation of the English edition of Terraform:
Up and Running: Writing Infrastructure as Code 3rd Edition
ISBN 978-1098116743 © 2022 Yevgeniy Brikman.

ISBN 978-601-08-3642-6

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
© Перевод на русский язык Спринт Бук, 2024
© Издание на русском языке, оформление Спринт Бук, 2024

Краткое содержание

Введение	12
Глава 1. Почему Terraform	28
Глава 2. Приступаем к работе с Terraform	67
Глава 3. Как управлять состоянием Terraform	109
Глава 4. Многократное использование инфраструктуры с помощью модулей Terraform	142
Глава 5. Работа с Terraform: циклы, условные выражения, развертывание и подводные камни	166
Глава 6. Управление конфиденциальными данными	218
Глава 7. Работа с несколькими провайдерами	251
Глава 8. Код Terraform промышленного уровня	307
Глава 9. Как тестировать код Terraform	349
Глава 10. Как использовать Terraform в команде	413
Приложение. Дополнительные ресурсы	456
Об авторе	459
Иллюстрация на обложке	460

Оглавление

Введение	12
Целевая аудитория книги.....	13
Почему я написал эту книгу.....	14
Структура издания.....	15
Что нового в третьем издании	17
Что нового во втором издании	20
Чего нет в этой книге	22
Примеры с открытым исходным кодом	23
Использование примеров кода.....	24
Условные обозначения.....	25
Благодарности.....	26
Глава 1. Почему Terraform	28
Восход DevOps.....	28
Что такое инфраструктура как код.....	31
Самодельные скрипты	31
Средства управления конфигурацией	32
Средства шаблонизации серверов	35
Средства оркестрации.....	39
Средства выделения ресурсов	41
Преимущества инфраструктуры как кода	43
Как работает Terraform	45
Сравнение Terraform с другими средствами IaC	47
Управление конфигурацией или выделение ресурсов.....	48
Выбор между изменяемой и неизменяемой инфраструктурой.....	49
Выбор между процедурными и декларативными языками	50
Выбор между универсальными и предметно-ориентированными языками.....	53
Наличие или отсутствие ведущего сервера.....	55
Наличие или отсутствие агентов.....	56

Платные и бесплатные предложения	58
Размер сообщества	59
Выбор между зрелостью и новизной	62
Использование нескольких инструментов вместе	62
Резюме	65
Глава 2. Приступаем к работе с Terraform	67
Подготовка учетной записи в AWS	68
Установка Terraform	71
Развертывание одного сервера	73
Развертывание одного веб-сервера	81
Развертывание конфигурируемого веб-сервера	88
Развертывание кластера веб-серверов	95
Развертывание балансировщика нагрузки	99
Удаление ненужных ресурсов	107
Резюме	108
Глава 3. Как управлять состоянием Terraform	109
Что представляет собой состояние Terraform	110
Общее хранилище для файлов состояния	112
Ограничения хранилищ Terraform	119
Изоляция файлов состояния	121
Изоляция через рабочие области	123
Изоляция с помощью размещения файлов	128
Источник данных terraform_remote_state	132
Резюме	140
Глава 4. Многократное использование инфраструктуры с помощью модулей Terraform	142
Что такое модуль	145
Входные параметры модуля	147
Локальные переменные модуля	151
Выходные переменные модуля	153
Подводные камни	156
Пути к файлам	156
Вложенные блоки	157
Версионирование модулей	160
Резюме	165

Глава 5. Работа с Terraform: циклы, условные выражения, развертывание и подводные камни	166
Циклы.....	167
Циклы с параметром count	167
Циклы с выражениями for_each	175
Циклы на основе выражений for	181
Циклы с использованием строковой директивы for	184
Условные выражения	186
Условные выражения с использованием параметра count.....	186
Условная логика с использованием выражений for_each и for	192
Условные выражения с использованием строковой директивы if	193
Развертывание с нулевым временем простоя.....	195
Подводные камни Terraform	206
Параметры count и for_each имеют ограничения	206
Ограничения развертываний с нулевым временем простоя	208
Даже хороший план может оказаться неудачным	211
Рефакторинг может иметь свои подвохи	213
Резюме.....	217
Глава 6. Управление конфиденциальными данными	218
Основы управления секретами	219
Инструменты управления секретами	220
Типы хранимых секретов	221
Как хранить секреты	221
Интерфейс доступа к секретам	222
Сравнение инструментов управления секретами	223
Использование инструментов управления секретами в комплексе с Terraform	224
Провайдеры	224
Ресурсы и источники данных.....	235
Файлы состояний и файлы планов	246
Резюме.....	248
Глава 7. Работа с несколькими провайдерами	251
Работа с одним провайдером.....	252
Что такое провайдер.....	252
Как устанавливаются провайдеры	253
Как используются провайдеры	256

Работа с несколькими копиями одного провайдера.....	257
Работа с несколькими регионами AWS	257
Работа с несколькими учетными записями AWS.....	269
Создание модулей, способных работать с несколькими провайдерами.....	276
Работа с несколькими провайдерами	280
Краткий курс Docker.....	281
Краткий курс по Kubernetes.....	285
Развертывание контейнеров Docker в AWS с помощью Elastic Kubernetes Service (EKS)	297
Резюме.....	306
Глава 8. Код Terraform промышленного уровня.....	307
Почему построение инфраструктуры промышленного уровня требует так много времени.....	309
Требования к инфраструктуре промышленного уровня.....	312
Инфраструктурные модули промышленного уровня.....	314
Мелкие модули.....	314
Компонуемые модули.....	319
Тестируемые модули.....	325
Версионирование модулей	332
За границами модулей Terraform.....	340
Резюме.....	348
Глава 9. Как тестировать код Terraform	349
Ручные тесты	350
Основы ручного тестирования.....	351
Очистка ресурсов после тестов.....	354
Автоматические тесты.....	354
Модульные тесты.....	356
Интеграционные тесты	383
Сквозные тесты.....	397
Другие подходы к тестированию	400
Резюме.....	410
Глава 10. Как использовать Terraform в команде.....	413
Внедрение концепции IaC внутри команды	414
Убедите свое начальство	414

Сделайте переход постепенным	417
Дайте своей команде время на обучение	419
Процесс развертывания кода приложений	420
Использование системы управления версиями	421
Локальное выполнение кода	422
Внесение изменений в код	422
Подача изменений на рассмотрение	423
Выполнение автоматических тестов	424
Слияние и выпуск новой версии	425
Развертывание	426
Процесс развертывания инфраструктурного кода	430
Использование системы управления версиями	430
Локальное выполнение кода	435
Внесение изменений в код	436
Подача изменений на рассмотрение	437
Выполнение автоматических тестов	439
Слияние и выпуск новой версии	441
Развертывание	441
Собираем все вместе	452
Резюме	454
Приложение. Дополнительные ресурсы	456
Книги	456
Блоги	457
Лекции	457
Информационные рассылки	458
Онлайн-форумы	458
Об авторе	459
Иллюстрация на обложке	460

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу:
comp@sprintbook.kz (издательство «SprintBook», компьютерная редакция).
Мы будем рады узнать ваше мнение!

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 для администрирования: *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/администрировать ее в различных публичных облачных сервисах (скажем, Amazon Web Services, Microsoft Azure, Google Cloud Platform, DigitalOcean), а также закрытых 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-0c55b159cbfaffe1f0"
  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 для управления инфраструктурой надежным автоматизированным инструментом, 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.

Изучив всего несколько глав, вы пройдете путь от простейшего примера «Hello, World» (на самом деле вы только что видели его!) до развертывания полного технологического стека (виртуальных серверов, кластера серверов, балансировщиков нагрузки, базы данных), рассчитанного на огромный трафик и крупную команду разработчиков. Это практическое руководство не только научит принципам DevOps и инфраструктуры как кода (infrastructure as code, IaC), но и проведет вас через десятки примеров кода, которые можно опробовать дома. Поэтому держите свой 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-инженерам, инженерам по мониторингу, инженерам DevOps, 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, выделением ресурсов и шаблонизацией серверов?
- Когда следует использовать Terraform, Chef, Ansible, Puppet, Pulumi, CloudFormation, Docker, Packer или Kubernetes?
- Как работает система Terraform и как с ее помощью управлять инфраструктурой?
- Как создавать многократно используемые модули Terraform?
- Как безопасно управлять конфиденциальной информацией при работе с Terraform?
- Как использовать Terraform с несколькими регионами, учетными записями и облаками?
- 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-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 позволило нам извлечь много непростых уроков. Я хочу поделиться с вами ими, чтобы вы могли сократить этот долгий процесс и овладеть Terraform в считанные 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

Вот список тем, которые освещаются в книге.

- *Глава 1 «Почему Terraform».* Как DevOps меняет наш подход к выполнению ПО; краткий обзор инструментов IaC, включая управление конфигурацией, шаблонизацию серверов, оркестрацию и выделение ресурсов; преимущества IaC; сравнение Terraform, Chef, Puppet, Ansible, Pulumi, 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 состояние гонки; 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 «Многократное использование инфраструктуры с помощью модулей Terraform».* Что такое модули; как создать простой модуль; как сделать модуль конфигурируемым с помощью входящих и исходящих значений; локальные переменные; версионирование модулей; 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 и как безопасно осуществлять рефакторинг кода Terraform.
- *Глава 6 «Управление конфиденциальными данными».* Введение в управление секретами; обзор различных типов секретов, разных способов их хранения и доступа к ним; сравнение распространенных инструментов управления секретами, таких как HashiCorp Vault, AWS Secrets Manager и Azure Key Vault; как управлять секретами при работе с провайдерами, включая аутентификацию через переменные окружения, роли IAM и OIDC; управление секретами при работе с ресурсами и источниками данных, в том числе с использованием переменных окружения, зашифрованных файлов и централизованных хранилищ секретов; безопасное обращение с файлами состояния и файлами планов.
- *Глава 7 «Работа с несколькими провайдерами».* Подробный обзор особенностей работы с провайдерами, поддерживающими Terraform, в том числе порядок выбора и установки определенной версии и ее использование в коде. Применение нескольких копий одного и того же провайдера, в том числе развертывание в нескольких регионах AWS, развертывание в нескольких учетных записях AWS и создание повторно используемых модулей, способных использовать несколько провайдеров. Как задействовать несколько разных провайдеров вместе, например, для запуска кластера Kubernetes (EKS) в AWS и развертывания контейнеров Docker в кластере.
- *Глава 8 «Код Terraform промышленного уровня».* Почему проекты DevOps всегда развертываются дольше, чем ожидается; что характеризует инфраструктуру, готовую к промышленному использованию; как создавать модули Terraform для промышленного применения; небольшие модули; сборные модули; тестируемые модули; готовые к выпуску модули; реестр Terraform;

проверка переменных; управление версиями Terraform, провайдеров Terraform, модулей Terraform и Terragrunt; «аварийные люки» в Terraform.

- *Глава 9 «Как тестировать код Terraform».* Ручное тестирование кода Terraform; создание и удаление тестового окружения; автоматизированное тестирование кода Terraform; Terratest; модульные тесты; интеграционные тесты; сквозные тесты; внедрение зависимостей; параллельное выполнение тестов; этапы тестирования; попытки; пирамида тестирования; статический анализ; план тестирования; тестирование сервера.
- *Глава 10 «Как использовать Terraform в команде».* Как внедрить Terraform в командную работу; как убедить начальство; рабочий процесс развертывания прикладного кода; рабочий процесс развертывания инфраструктурного кода; управление версиями; золотое правило Terraform; рецензирование 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 в третьем издании

Первое издание этой книги вышло в 2017 году, второе вышло в 2019 году, и хотя мне сложно в это поверить, но сейчас я работаю уже над третьим изданием. Время летит. Удивительно, как много изменилось за эти годы!

Если вы читали второе издание книги и хотите узнать, что нового появилось в этом издании, или вам просто интересно увидеть, как Terraform развивался между 2019 и 2022 годами, то ниже перечислены некоторые основные различия между вторым и третьим изданиями.

- *Сотни страниц обновленного содержания.* Третье издание книги примерно на 100 страниц объемнее второго. Также, по моим оценкам, примерно от трети до половины страниц второго издания были обновлены. Почему так много новой информации? Потому что с момента выхода второго издания вышло шесть новых версий Terraform со значительными изменениями: 0.13, 0.14, 0.15, 1.0, 1.1 и 1.2. Более того, многие провайдеры Terraform провели собственные серьезные обновления, в том числе AWS, который на момент выхода второго издания был представлен версией 2, а к моменту выхода третьего — версией 4. Кроме того, за последние несколько лет значительно

выросло сообщество Terraform, что привело к появлению множества новых приемов, инструментов и модулей. Я постарался отразить как можно больше этих изменений, добавив две новые главы и внося существенные обновления в остальные, как описано ниже.

- *Новые функциональные возможности провайдеров.* В Terraform значительно улучшена поддержка провайдеров. В третье издание я добавил совершенно новую главу (главу 7), где рассказывается, как работать с несколькими провайдерами: например, как организовать развертывание в нескольких регионах, нескольких учетных записях и нескольких облаках. Кроме того, по многочисленным просьбам в эту главу включен совершенно новый набор примеров, показывающих, как использовать Terraform, Kubernetes, Docker, AWS и EKS для запуска контейнерных приложений. Наконец, я обновил все остальные главы, чтобы осветить новые функциональные возможности провайдеров, появившиеся в последних нескольких версиях, включая блок `require_providers`, добавленный в Terraform 0.13 (предлагает лучший способ установки и версионирования официальных и неофициальных провайдеров Terraform, а также управления ими); файлы блокировки, внедренные в Terraform 0.14 (помогает гарантировать использование одних и тех же версий провайдеров всеми членами вашей команды) и параметр `configuration_aliases`, появившийся в Terraform 0.15 (улучшает управление провайдерами внутри модулей).
- *Улучшенное управление секретами.* При использовании кода Terraform часто приходится иметь дело со многими видами секретов: паролями баз данных, ключами API, учетными данными облачного провайдера, сертификатами TLS и т. д. Поэтому я добавил в третье издание новую главу (главу 6), посвященную этой теме, где сравниваются распространенные инструменты управления секретами и представлены примеры кода, иллюстрирующие разные методы безопасного использования секретов в Terraform, включая переменные окружения, зашифрованные файлы, централизованные хранилища секретов, роли IAM, OIDC и многое другое.
- *Новые функциональные возможности модулей.* В Terraform 0.13 появилась возможность использовать `count`, `for_each` и `depend_on` в блоках `module`, что делает модули более мощными, гибкими и пригодными для повторного применения. Примеры использования этих новых возможностей вы найдете в главах 5 и 7.
- *Новые возможности проверки.* В главе 8 я добавил примеры использования функции `validation`, появившейся в Terraform 0.13 и выполняющей простые проверки переменных (например, превышение минимальных или максимальных значений), а также функций `precondition` и `postcondition`, появившихся в Terraform 1.2 и выполняющих простые проверки ресурсов и источников данных перед запуском `apply` (например, для принудительного использования AMI при выборе пользователем архитектуры x86_64) либо после запуска `apply`

(например, проверка успешности шифрования тома EBS). В главе 6 я покажу, как использовать параметр `sensitive` (добавлен в Terraform 0.14 и 0.15), не позволяющий выводить секреты при запуске `plan` или `apply`.

- **Новые возможности рефакторинга.** В Terraform 1.1 появился блок `moved`, обеспечивающий лучший способ выполнения таких операций рефакторинга, как переименование ресурса. Раньше для этого пользователь должен был вручную запускать операции `terraform state mv`, а теперь, как показано в новом примере в главе 5, этот процесс можно полностью автоматизировать, что делает обновление модулей более безопасным и удобным.
- **Дополнительные возможности тестирования.** Инструменты автоматического тестирования кода Terraform продолжают совершенствоваться. В главе 9 я добавил пример кода и привел сравнение инструментов статического анализа, включая `tfsec`, `tfint`, `terrascan` и `validate`, инструментов тестирования `plan`, включая `Terratest`, `OPA` и `Sentinel`, а также инструментов тестирования серверов, включая `inspec`, `serverspec` и `goss`. Я также добавил сравнение существующих подходов к тестированию, чтобы вы могли выбрать наиболее подходящий для вас.
- **Улучшенная стабильность.** Выпуск версии Terraform 1.0 стал важной вехой для Terraform, означающей не только достижение определенного уровня зрелости, но и гарантии совместимости — все выпуски 1.x будут обратно совместимы, поэтому обновление между выпусками v1.x больше не должно требовать изменений кода, рабочих процессов или файлов состояния. Файлы состояния Terraform теперь перекрестно совместимы с Terraform 0.14, 0.15 и всеми выпусками 1.x, а источники данных удаленного состояния Terraform перекрестно совместимы с Terraform 0.12.30, 0.13.6, 0.14.0, 0.15.0 и всеми версиями 1.x. Я также обновил главу 8, добавив примеры, показывающие как лучше управлять версиями Terraform (включая использование `tfenv`), `Terragrunt` (включая использование `tgswitch`) и провайдеров Terraform (включая использование файла блокировки).
- **Взросшая зрелость.** Terraform был загружен более 100 миллионов раз, насчитывает более 1500 участников и используется почти в 79 % компаний из списка Fortune 500¹, поэтому можно с уверенностью сказать, что экосистема значительно выросла за последние несколько лет и стала более зрелой. Сейчас проект Terraform насчитывает больше разработчиков, провайдеров, повторно используемых модулей, инструментов, плагинов и классов. Появилось больше книг и учебных пособий. Кроме того, HashiCorp, компания, создавшая Terraform, провела IPO (первичное публичное размещение акций) в 2021 году, поэтому теперь Terraform больше не является небольшим

¹ Согласно HashiCorp S1 (<https://www.sec.gov/Archives/edgar/data/0001720671/000119312521319849/d205906ds1.htm>).

стартапом — это крупная и устойчивая компания, продающая свои акции, для которой Terraform является крупнейшим предложением.

- Множество других изменений. Кроме перечисленного выше, произошло множество других изменений, включая запуск Terraform Cloud (веб-интерфейс для использования Terraform); возросшую зрелость инструментов, популярных в сообществе, таких как Terragrunt, Terratest и tfenv; добавление множества новых возможностей провайдеров (в том числе новых способов развертывания с нулевым временем простоя, таких как обновление экземпляра, о чем рассказывается в главе 5); появление новых функций (в частности, в главе 5 я добавил примеры использования функции `one` и в главе 7 — функции `try`); прекращение поддержки многих старых возможностей (например, источника данных `template_file`, многих параметров `aws_s3_bucket`, `list` и `map`, внешних ссылок в `destroy`) и многое другое.

Что нового во втором издании

Возвращаясь еще раньше в прошлое, замечу, что второе издание книги стало примерно на 150 страниц больше первого издания. Вот краткий перечень этих изменений, иллюстрирующий, как эволюционировал проект Terraform в период с 2017 по 2019 год.

- *Четыре больших выпуска Terraform.* Когда вышла первая книга, стабильной версией Terraform была 0.8. Спустя четыре основных выпуска 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. Во втором издании я добавил новую главу (глава 9 в этом издании), затрагивающую такие темы, как модульное, интеграционное и сквозное тестирование, внедрение зависимостей, распараллеливание тестов, статический анализ и др.
- *Улучшения в модулях.* Инструментарий и методики создания модулей Terraform тоже заметно эволюционировали. Во втором издании я добавил новую главу (глава 8 в этом издании), где вы найдете руководство по написанию испытанных модулей промышленного уровня с возможностью повторного использования — таких, которым можно доверить благополучие своей компании.

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

- *Улучшения в рабочем процессе.* Глава 8 (в этом издании — глава 10) была полностью переписана согласно тем изменениям, которые произошли в процедуре интеграции Terraform в рабочий процесс команд. Там, помимо прочего, можно найти подробное руководство, описывающее основные этапы создания прикладного и инфраструктурного кода: 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, развитые ограничители типов, условные выражения с отложенным вычислением, поддержку выражений `null`, `for_each` и `for`, встроенные блоки и др. Все примеры кода во втором издании были адаптированы для HCL2, а новые возможности языка подробно рассматриваются в главах 5 и 6 (в этом издании — глава 8).
- *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 `terraform init` каждый раз, когда вы начинаете работать с новым модулем. Об этом пойдет речь в главах 2 и 7 (в этом издании — глава 9).
- *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 больше¹. 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, PostgreSQL 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 лучшее покрытие: скажем, провайдер для AWS охватывает большинство сервисов

¹ Список провайдеров для Terraform можно найти на странице <https://registry.terraform.io/browse/providers>.

этой платформы, а поддержка новых сервисов часто появляется даже раньше, чем у CloudFormation!

- *Реестр Terraform.* В 2017 году компания HashiCorp представила реестр Terraform (<https://registry.terraform.io/>) — пользовательский интерфейс, который облегчает просмотр и загрузку открытых универсальных модулей Terraform, разрабатываемых сообществом. В 2018 году была добавлена поддержка частных реестров, позволяющая организациям создавать свои закрытые реестры. В Terraform 0.11 появился полноценный синтаксис для загрузки модулей из реестра. Подробнее о реестре рассказывается в главе 8.
- *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 теперь содержат путь к файлу, номер строки и фрагмент кода.
- *Много других мелких изменений.* Было сделано много менее значительных изменений, включая появление локальных переменных (см. раздел «Локальные переменные модуля» главы 4), новые «аварийные люки» для взаимодействия с внешним миром с помощью скриптов (например, раздел «За границами Terraform» главы 8), выполнение `plan` в рамках команды `apply`, исправление циклических проблем с `create_before_destroy`, значительное улучшение параметра `count`, которое позволяет ссылаться в нем на источники данных и ресурсы, 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, с концепцией «инфраструктура как код» или с 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 этой 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, и вам не придется ничего платить за выполнение примеров). Именно поэтому примеры кода не охватывают и не требуют платных услуг HashiCorp, Terraform Cloud или Terraform Enterprise. И именно поэтому я опубликовал все примеры с открытым исходным кодом.

Примеры с открытым исходным кодом

Все доступные в книге листинги с кодом можно найти по адресу <https://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 находятся в папке `code` и разбиты по главам, имена которых начинаются с названия инструмента или языка (например, Terraform, Packer, OPA) и заканчиваются номером главы. Единственное исключение — код на Go, используемый в автоматических тестах в главе 9, который находится в папке `terraform` и соответствует рекомендациям по использованию структуры папок `examples`, `modules` и `test`, данным в этой главе.

В табл. В.1 приводятся несколько примеров, показывающих, как искать код разного типа в репозитории.

Таблица В.1. Где в репозитории искать примеры кода разного типа

Тип кода	Глава	Папка в репозитории с примерами
Terraform	Глава 2	<code>code/terraform/02-intro-to-terraform-syntax</code>
Terraform	Глава 5	<code>code/terraform/05-tips-and-tricks</code>
Packer	Глава 1	<code>code/packer/01-why-terraform</code>
OPA	Глава 9	<code>code/opa/09-testing-terraform-code</code>
Go	Глава 9	<code>code/terraform/09-testing-terraform-code/test</code>

Стоит отметить, что большинство примеров демонстрирует состояние кода на момент *завершения* главы. Если вы хотите научиться как можно большему, весь код лучше писать самостоятельно, с нуля.

Писать код вы начнете в главе 2, где научитесь развертывать кластер веб-серверов с помощью Terraform от начала и до конца. После этого следуйте инструкциям в каждой новой главе, развивая и улучшая этот пример. Вносите изменения так, как указано в книге, пытайтесь писать весь код самостоятельно и используйте примеры из репозитория на GitHub, только чтобы свериться или прояснить непонятные моменты.



Версии

Все примеры в книге проверены в версии Terraform 1.x и AWS Provider 4.x, которые на момент написания были самыми свежими. Поскольку Terraform — относительно новый инструмент, 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

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Условные обозначения

В книге используются следующие типографические обозначения.

Курсив

Обозначает новые термины и важные слова.

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, утверждения 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 рисунок обозначает совет или предложение.



Этот значок указывает на примечание общего характера.



Этот значок указывает на предупреждение или предостережение.

Благодарности

Джошу Паднику

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 изменило мою жизнь, и я горжусь тем, что вы помогаете мне делиться некоторыми из моих текстов с другими. Отдельная благодарность Брайану Андерсону, Вирджинии Уилсон и Корбин Коллинз за помощь в работе над первым, вторым и третьим изданиями соответственно.

Компьющикам Gruntwork

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

Клиентам 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 сопутствовал успех (в этой 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.

Рецензентам

Спасибо Кифу Моррису, Сету Варго, Маттиасу Гису, Рокардо Феррейре, Акашу Махаяну, Морицу Хейберу, Тейлору Долежалю и Антону Бабенко за вычитку первых черновиков книги и за большое количество подробных и конструктивных отзывов. Ваши советы улучшили эту книгу.

Читателям первого и второго изданий

Те из вас, кто купил первое и второе издание, сделали возможным выход третьего. Спасибо. Ваши отзывы, вопросы, предложения относительно исходного кода и постоянная жажда новостей послужили мотивацией для добавления нового содержимого и обновления существующего. Надеюсь, обновленный текст окажется полезным, и с нетерпением жду дальнейшего давления со стороны читателей.

Маме, папе, Ларисе, 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@sprintbook.kz (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

ГЛАВА 1

Почему Terraform

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

Доставка ПО подразумевает выполнение множества действий для того, чтобы сделать код доступным для клиента. К ним относятся запуск кода на промышленных серверах, обеспечение его устойчивости к перебоям в работе и всплескам нагрузки, защита от злоумышленников. Прежде чем погружаться в мир Terraform, стоит сделать шаг назад и поговорить о роли этого инструмента в области доставки программного обеспечения.

В этой главе мы подробно обсудим следующие темы.

- Восход DevOps.
- Что такое инфраструктура как код.
- Преимущества инфраструктуры как кода.
- Как работает Terraform.
- Сравнение Terraform с другими инструментами для работы с инфраструктурой как с кодом.

Восход DevOps

Если бы в недалеком прошлом вы захотели создать компанию для разработки 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: разработчики (developers, Devs), которые занимались написани-

ем программ, и системные администраторы (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 программная работа системных администраторов, такая как установка приложений и их зависимостей, часто требовала выполнения команд на сервере.

До поры до времени этого было достаточно, но по мере роста компании начинали возникать проблемы. Обычно выглядело это 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 звонков по ночам 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 двухгодичного 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 происходит глубинный сдвиг. 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, Docker и Kubernetes. 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 слияния веток вы регулярно интегрируете свой код, поддерживая его в постоянной готовности к развертыванию. Вместо ежемесячных развертываний вы можете доставлять свой код десятки раз в день или даже после каждой фиксации. Вместо того чтобы бороться 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 производственных происшествий 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 проблем¹.

Движение DevOps основано на четырех принципах: культуре, автоматизации, измерении и разделении (в английском языке иногда используется аббревиатура CASM — culture, automation, measurement, sharing). Эта книга не задумывалась как комплексный обзор DevOps (рекомендуемая литература приводится в приложении), поэтому сосредоточусь лишь на одном из этих 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, IaC).

¹ Согласно книге Kim G., Humble J., Debois P., Willis J. The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations. — IT Revolution Press, 2016. (Ким Дж., Дебуа П., Уиллис Дж., Хамбл Дж. Руководство по DevOps. Как добиться гибкости, надежности и безопасности мирового уровня в технологических компаниях.)

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:

- самодельные скрипты;
- средства управления конфигурацией;
- средства шаблонизации серверов;
- средства оркестрации;
- средства выделения ресурсов.

Рассмотрим каждую из них.

Самодельные 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 (например, Bash, Ruby, Python), и выполняете получившийся скрипт на своем сервере, как показано на рис. 1.1.

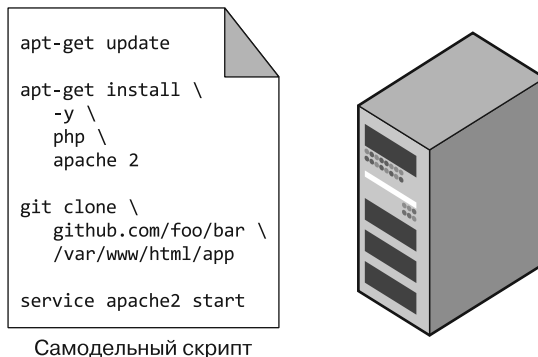


Рис. 1.1. Выполнение самодельного скрипта на сервере

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

В отличие от инструментов, специально созданных для IaC и предоставляющих 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 самодельных скриптах каждый разработчик использует собственный стиль и делает вещи по-своему. Если речь идет о скрипте из восьми строчек, который устанавливает Apache, обе эти 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. Самодельные 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 и Ansible 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, размещения файла, параметров с понятными именами, управления конфиденциальными данными и т. д. Если каждый разработчик организует свои самодельные скрипты по-разному, то большинство средств управления конфигурацией имеют набор правил и соглашений, которые упрощают навигацию по коду.
- *Идемпотентность.* Написать рабочий самодельный 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 убедиться в отсутствии этой строчки. И всегда, собираясь запустить программу, надо определить, не 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 в него множество условных выражений. Для сравнения: большинство функций Ansible идемпотентно по умолчанию. Например, роль `Ansible web-server.yaml` установит сервер Apache, только если он еще не установлен, и попытается его запустить лишь при условии, что он еще не запущен.

- *Распределенность.* Самодельные 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.

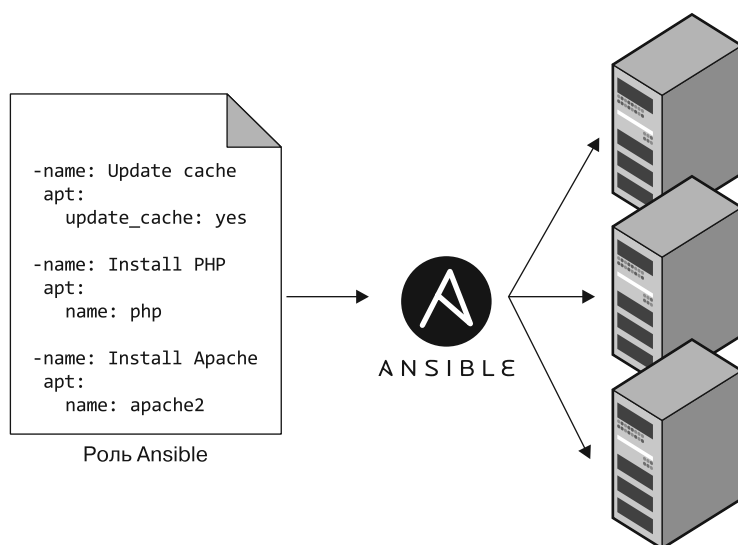


Рис. 1.2. Такие средства управления конфигурацией, как Ansible, способны выполнять ваш код на большом количестве серверов

Например, чтобы применить роль `web-server.yml` к пяти серверам, нужно сначала создать файл под названием `hosts`, содержащий 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 *сценарий Ansible*:

```

- hosts: webservers
  roles:
    - webserver
  
```

И наконец, выполнить его:

```
ansible-playbook playbook.yml
```

Это заставит Ansible параллельно сконфигурировать все пять серверов. В качестве альтернативы в сценарии можно указать параметр `serial`. Это позволит выполнить *пакетное развертывание*. Например, если присвоить `serial` значение 2, Ansible будет обновлять сразу по два сервера, пока не обновит все пять. Дублирование любой части этой логики в самодельных 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 запускать 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 установить на все ваши серверы, используя другие инструменты IaC.

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

- *Виртуальные машины (ВМ)* 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 совершенно одинаково во всех средах (например, на вашем компьютере, сервере проверки качества и промышленном сервере). Недостаток в том, что виртуализация всего этого оборудования и выполнение совершенно отдельной ОС на каждой ВМ требует больших накладных расходов с точки зрения потребления процессора и памяти, а также времени запуска. Образы ВМ можно описывать в виде кода, применяя такие инструменты, как Packer и Vagrant.

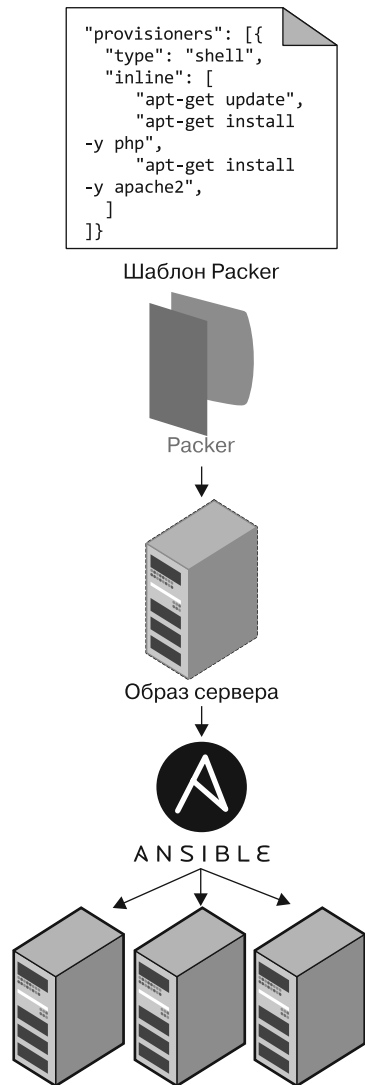


Рис. 1.3. С помощью таких средств шаблонизации, как Packer, можно создавать самодостаточные образы серверов. Затем, используя другие инструменты, такие как Ansible, эти образы можно установить на все ваши серверы

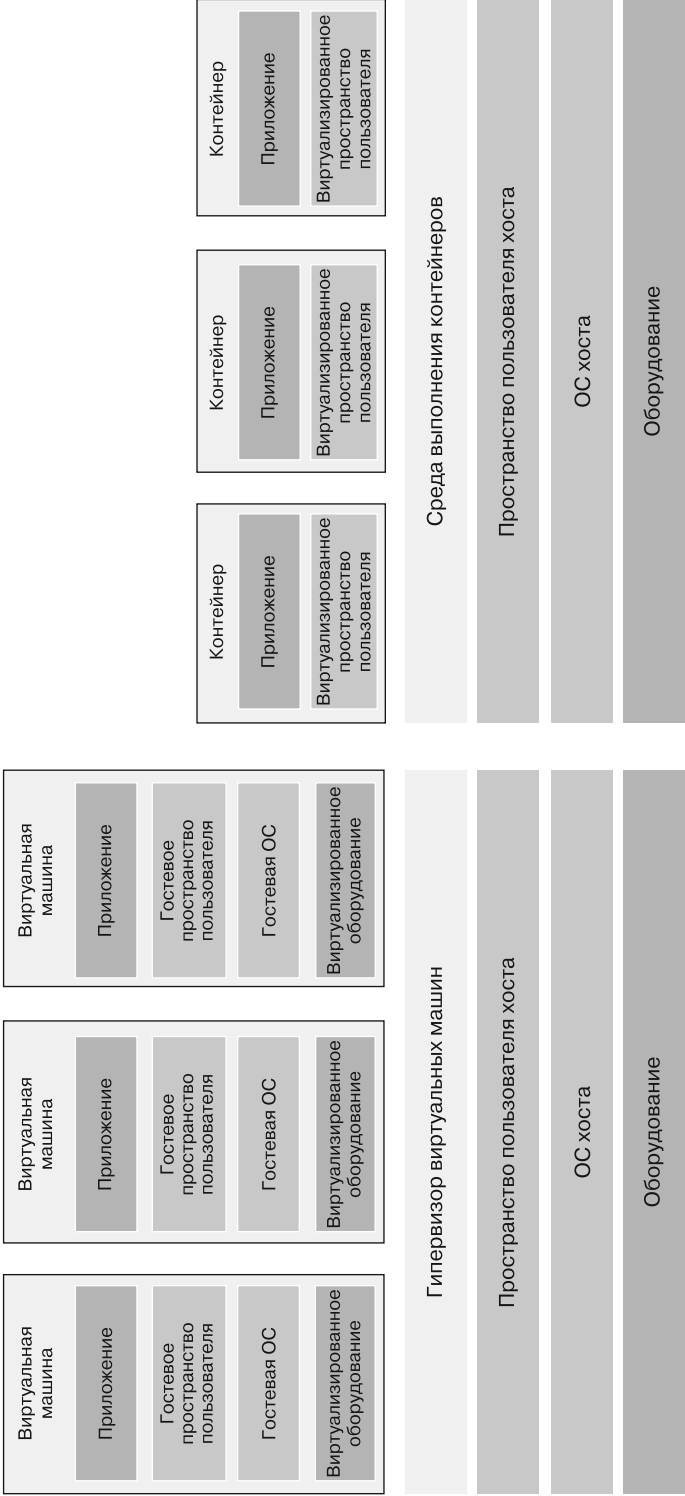


Рис. 1.4. Существует два вида образов: VM (слева) и контейнеры (справа). VM виртуализируют оборудование, тогда как контейнеры — только пользовательское пространство

- *Контейнеры* эмулируют пользовательское пространство ОС¹. 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 этой 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, промышленном 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 сложнее². Поскольку применяются общие ядро и оборудование, ваши контейнеры могут загружаться в считанные 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. Кстати, пример с использованием Docker я приведу в главе 7.

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-0fb653ca2d3203ac1",
```

¹ В современных операционных системах код выполняется в одном из двух «пространств»: пространстве ядра и пространстве пользователя. Код, работающий 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 потому 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.

² 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.

```

    "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"
    ]
  }
  "pause_before": "60s"
}
}]
}
```

Шаблон Packer настраивает тот же веб-сервер Apache, который мы видели в файле `setup-webserver.sh`, и использует тот же код на Bash¹. Единственное отличие от предыдущего кода в том, что Packer не запускает веб-сервер Apache (с помощью команды вроде `sudo service apache2 start`). 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).

Создать АМІ из этого шаблона можно командой `packer build web server.json`. Когда сборка завершится, полученный образ АМІ можно установить на все ваши серверы в AWS и сконфигурировать Apache для запуска во время загрузки компьютера (как показано в примере далее в этом разделе). В результате все они будут работать абсолютно одинаково.

Имейте в виду, что разные средства шаблонизации серверов имеют различное назначение. Packer обычно используется для создания образов, выполняемых непосредственно поверх промышленных серверов, таких как АМІ (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 АМІ, у которых внутри установлен Docker Engine; дальше эти образы развертываются на кластере серверов в вашей учетной записи AWS, и затем

¹ В качестве альтернативы Bash для настройки образов в Packer можно использовать средства управления конфигурацией, такие как Ansible и Chef.

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 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 нельзя внести 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 нужно выбрать какой-то способ выполнения следующих действий.

- Развертывать ВМ и контейнеры с эффективным применением оборудования.
- Накатывать обновления на многочисленные ВМ и контейнеры, используя такие стратегии, как пакетные, сине-зеленые и канареечные развертывания.
- Следить за работоспособностью ВМ и контейнеров, автоматически заменяя неисправные (*автовосстановление*).
- Масштабировать количество ВМ и контейнеров в обе стороны в зависимости от нагрузки (*автомасштабирование*).
- Распределять трафик между ВМ и контейнерами (*балансировка нагрузки*).
- Позволять ВМ и контейнерам находить друг друга и общаться между собой по сети (*обнаружение сервисов*).

Все эти задачи находятся 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 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, можно описать поведение контейнера Docker в виде кода внутри YAML-файла:

```
apiVersion: apps/v1

# Используем объект Deployment для развертывания нескольких реплик
# Docker-контейнера (возможно, больше одного) и декларативного
# накатывания обновлений для него
kind: Deployment

# Метаданные этого объекта, включая его имя
metadata:
  name: example-app

# Спецификация, которая конфигурирует это развертывание
spec:
  # Благодаря этому объект Deployment знает, как найти контейнер
  selector:
    matchLabels:
      app: example-app

  # Приказываем объекту Deployment развернуть три реплики Docker-контейнера
  replicas: 3

  # Определяет способ обновления развертывания. Здесь указываем
  # пакетное обновление
  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 объект `Deployment`, который декларативно описывает следующие вещи.

- Один или несколько контейнеров Docker для одновременного запуска. Эта группа контейнеров называется *Pod-оболочкой*. Pod-оболочка, описанная выше, содержит единственный контейнер, который запускает Apache.
- Настройки каждого контейнера Docker в Pod-оболочке. В нашем примере Pod-оболочка настраивает Apache для прослушивания порта 80.
- Сколько копий (*реplik*) Pod-оболочки должно быть запущено в 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 Pod-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 контейнеру), производительности (в частности, выбираются наименее загруженные серверы) и т. д. Кроме того, Kubernetes постоянно следит за тем, чтобы в кластере всегда выполнялось три реплики. Для этого автоматически заменяется любая Pod-оболочка, вышедшая из строя или переставшая отвечать.
- Как развертывать обновления. Когда выходит новая версия контейнера Docker, наш код запускает три новые реплики, ждет, когда они начнут работу, и затем удаляет три старые копии.

Так много возможностей всего в нескольких строчках на YAML! Чтобы развернуть приложение в Kubernetes, нужно выполнить команду `kubectl apply -f example-app.yml`. Чтобы накатить обновления, можно отредактировать YAML-файл и снова запустить `kubectl apply`. Также есть возможность управлять кластером Kubernetes и приложением с помощью Terraform, как будет показано в главе 7.

Средства выделения ресурсов

В отличие от инструментов управления конфигурацией, шаблонизации серверов и оркестрации, код которых выполняется на каждом сервере, *средства выделения ресурсов*, такие как Terraform, CloudFormation, OpenStack Heat и Pulumi, 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 инфраструктуры (рис. 1.5).

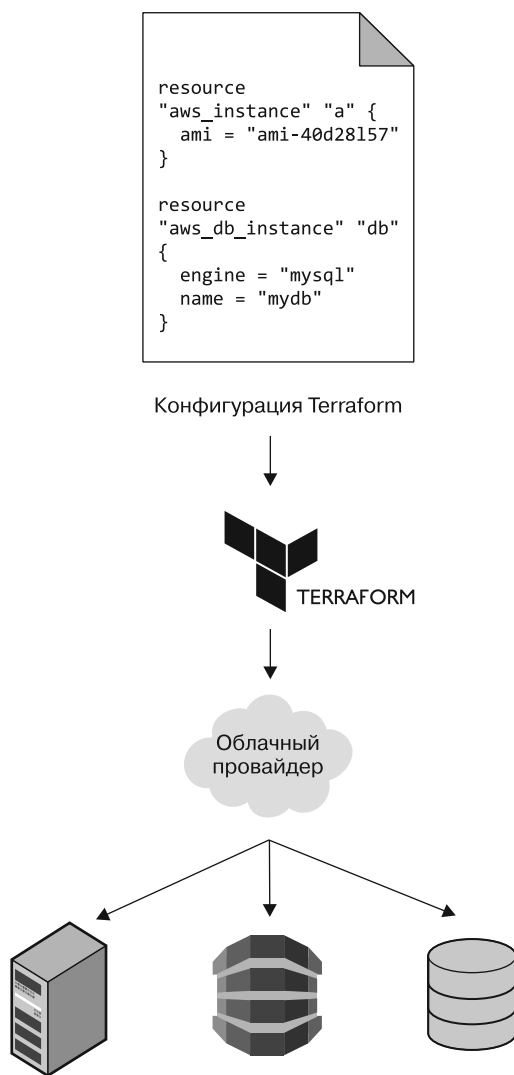


Рис. 1.5. Средства выделения ресурсов можно использовать в связке с вашим облачным провайдером, чтобы создавать серверы, базы данных, балансировщики нагрузки и любые другие элементы инфраструктуры

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

```
resource "aws_instance" "app" {
  instance_type = "t2.micro"
  availability_zone = "us-east-2a"
  ami = "ami-0fb653ca2d3203ac1"
```

```
user_data = <<-EOF
    #!/bin/bash
    sudo service apache2 start
EOF
}
```

Не нужно волноваться, если вам непонятны какие-то элементы 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 предыдущем разделе. В нем содержится PHP, Apache и исходный код приложения.
- **user_data**. Этот Bash-скрипт выполняется при загрузке веб-сервера. В предыдущем примере он используется для запуска Apache.

Иными словами, этот код демонстрирует, как можно объединить выделение ресурсов и шаблонизацию серверов, что является распространенной практикой в неизменяемой инфраструктуре.

Преимущества инфраструктуры как кода

Теперь, после знакомства со всевозможными разновидностями IaC, у вас может появиться вопрос: зачем это нужно? Зачем изучать кучу новых языков и инструментов, обрекая себя еще бóльшим количеством кода, который нужно поддерживать?

Дело в том, что код довольно мощный. Усилия, направленные на преобразование ручных процессов в код, вознаграждаются огромным расширением ваших возможностей по доставке ПО. Согласно докладу о состоянии DevOps за 2016 год (<https://puppet.com/resources/report/2016-state-devops-report>), 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 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, развертывающих 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 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.* Если состояние инфраструктуры хранится в голове одного сисадмина, который может уйти в отпуск, уволиться или попасть под автобус¹, то в какой-то момент вы можете оказаться в ситуации, когда дальнейшее управление инфраструктурой становится невозможным. С другой стороны, вы можете описать его в виде кода, 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 журнал фиксаций кода будет записываться 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 инфраструктуре. Вслед за этим проблему можно решить за счет простого отката к предыдущей версии кода IaC, в которой вы уверены.
- *Проверка.* Если состояние инфраструктуры описано в файле, то 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 развертывания каждого продукта в каждой среде с нуля у вас будет возможность использовать в качестве основы известные, задокументированные и проверенные на практике компоненты².
- *Счастье.* Есть еще одна очень важная причина, почему следует использовать IaC, которую часто упускают из виду, — счастье. 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

¹ Именно отсюда происходит термин «коэффициент автобуса». Коэффициент автобуса команды — это количество людей, которых вы можете потерять (например, из-за того, что их сбил автобус), прежде чем бизнес остановится. Никто не хотел бы, чтобы коэффициент автобуса был равен 1.

² В качестве примера можно взять библиотеку IaC от компании Gruntwork (<https://bit.ly/2H3Y7yT>).

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 осознаете важность IaC. Следующий вопрос: является ли Terraform лучшим средством IaC именно для вас? Чтобы ответить на него, мы кратко рассмотрим принцип работы Terraform, а затем сравним 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 собрать сервер (или любой другой компьютер), 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*, описывающие желаемую инфраструктуру. В концепции «*инфраструктура как код*» эти файлы играют роль кода. Вот пример конфигурации Terraform:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrrdatas     = [aws_instance.example.public_ip]
}
```

Даже если вы никогда раньше не видели код Terraform, вы без особого труда поймете его. Этот фрагмент заставляет Terraform выполнить API-вызовы к двум провайдерам: к AWS, чтобы развернуть там сервер, и к Google Cloud, чтобы создать DNS-запись с IP-адресом сервера в 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 развернуть с помощью определенных команд, таких как `terraform apply`. 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).

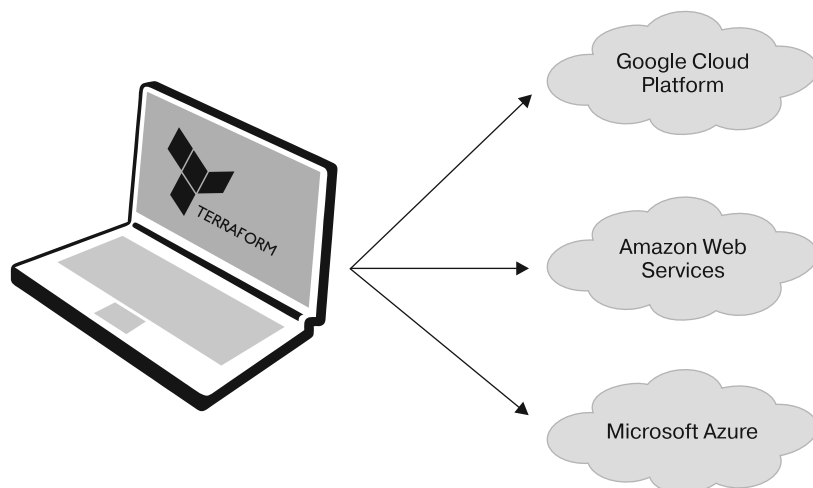


Рис. 1.6. Terraform — это утилита, которая преобразует содержимое конфигурационных файлов в API-вызовы к облачным провайдерам

Если кто-то в вашей команде захочет изменить инфраструктуру, то, вместо того чтобы делать это вручную прямо на серверах, они отредактируют конфигурационные файлы Terraform, проверят их с помощью автоматических тестов и разбора кода, зафиксируют обновленный код в системе управления версиями и затем выполнят команду `terraform apply`, чтобы выполнить API-вызовы, необходимые для развертывания изменений.



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, то сможете 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. Нельзя развернуть «идентичную инфраструктуру» в разных облаках, поскольку инфраструктуры, предоставляемые облачными провайдерами, разнятся! Серверы, балансировщики нагрузки и базы данных, предлагаемые в AWS, Azure и Google Cloud, сильно отличаются с точки зрения возможностей, конфигурации, управления, безопасности, масштабируемости, доступности, наблюдаемости и т. д. Не существует простого и «прозрачного» способа преодолеть эти отличия, особенно учитывая, что некоторые функции одного облачного провайдера часто отсутствуют у 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, то вам может быть непонятно, 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. Теоретически это действительно так, но в этом мало пользы. 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 выделения ресурсов: Terraform, Chef, Puppet, Ansible, Pulumi, CloudFormation и OpenStack Heat. Моя цель — помочь вам понять, является ли Terraform хорошим выбором. Для этого я объясню, почему моя компания, Gruntwork, выбрала Terraform в качестве средства IaC и в каком-то

смысле почему я написал эту книгу¹. Как и с любым техническим решением, все сводится к компромиссам и приоритетам. Даже если ваши задачи отличаются от моих, надеюсь, что описанный здесь процесс мышления поможет вам принять собственное решение.

Вам придется выбирать между такими вещами, как:

- управление конфигурацией или выделение ресурсов;
- изменяемая инфраструктура или неизменяемая инфраструктура;
- процедурный язык или декларативный язык;
- универсальный язык или предметно-ориентированный язык;
- наличие или отсутствие ведущего сервера;
- наличие или отсутствие агента;
- платные или бесплатные предложения;
- большое сообщество или маленькое сообщество;
- зрелость или новизна;
- использование нескольких инструментов вместе.

Управление конфигурацией или выделение ресурсов

Как упоминалось ранее, Chef, Puppet и Ansible управляют конфигурацией, тогда как CloudFormation, Terraform, OpenStack Heat и Pulumi выделяют ресурсы.

Это не совсем четкое разделение, так как средства управления конфигурацией обычно в какой-то степени поддерживают выделение ресурсов (например, вы можете развернуть сервер с помощью Ansible), а средства выделения ресурсов управляют конфигурацией до некоторой степени (скажем, на каждом сервере, выделенном с помощью Terraform, можно запускать конфигурационные скрипты). Поэтому следует выбирать тот инструмент, который лучше всего подходит для вашего случая.

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

¹ Docker, Packer и Kubernetes не участвуют в сравнении, поскольку их можно использовать в сочетании с любыми средствами управления конфигурацией или выделения ресурсов.

выделения ресурсов. В главе 7 вы увидите пример совместного использования Terraform и Docker, что в наши дни является особенно популярной комбинацией.

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

Выбор между изменяемой и неизменяемой инфраструктурой

Обычно в средствах управления конфигурацией, таких как Chef, Puppet и Ansible, по умолчанию применяется парадигма изменяемой инфраструктуры.

Например, если приказать Chef установить новую версию OpenSSL, он запустит процесс обновления ПО 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 от остальных серверов, 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 промышленном сервере, который, в отличие от тестовой среды, месяцами накапливал обновления.

Если вы применяете средства выделения 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 проблем 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 выделения 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, Puppet, Open Stack Heat и Pulumi 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 отличие, рассмотрим пример. Представьте, что вам нужно развернуть десять серверов (*экземпляров, или инстансов, EC2* в терминологии AWS), используя образ AMI с идентификатором `ami-0fb653ca2d3203ac1` (Ubuntu 20.04). Так выглядит шаблон Ansible, который делает это в процедурном стиле:

```
- ec2:
  count: 10
  image: ami-0fb653ca2d3203ac1
  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-0fb653ca2d3203ac1"
  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, 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 стано-

вится 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-0fb653ca2d3203ac1
  instance_type: t2.micro
```

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

```
resource "aws_instance" "example" {
  count      = 15
  ami       = "ami-0fb653ca2d3203ac1"
  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-0fb653ca2d3203ac1"
  + instance_type = "t2.micro"
  + (...)
}

# aws_instance.example[12] will be created
+ resource "aws_instance" "example" {
  + ami           = "ami-0fb653ca2d3203ac1"
  + instance_type = "t2.micro"
  + (...)
}

# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
  + ami           = "ami-0fb653ca2d3203ac1"
  + instance_type = "t2.micro"
  + (...)
}
```

```
# aws_instance.example[14] will be created
+ resource "aws_instance" "example" {
+   ami           = "ami-0fb653ca2d3203ac1"
+   instance_type = "t2.micro"
+   (...)
}
```

Plan: 5 to add, 0 to change, 0 to destroy.

А если вы хотите развернуть другую версию приложения, например АМІ 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 значение в параметре `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, и 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* применения этих шаблонов. 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 приходится 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 не нужно учитывать текущее состояние окружающего мира. Вместо этого можно сосредоточиться лишь на описании нужного состояния, а Terraform автоматически сообразит, как к нему перейти. Поэтому кодовая база Terraform обычно остается компактной и понятной.

Выбор между универсальными и предметно-ориентированными языками

Для управления инфраструктурой как кодом Chef и Pulumi позволяют использовать *универсальный язык программирования* (general-purpose programming language, GPL): Chef поддерживает Ruby; Pulumi поддерживает широкий спектр универсальных языков, включая JavaScript, TypeScript, Python, Go, C#, Java и др. Terraform, Puppet, Ansible, CloudFormation и OpenStack Heat, напротив, используют *предметно-ориентированные языки* (domain-specific language, DSL): Terraform использует HCL; Puppet — Puppet Language; Ansible, CloudFormation и OpenStack Heat — YAML (CloudFormation также поддерживает JSON).

Разделение между универсальными и предметно-ориентированными языками не совсем четкое — это скорее полезная мысленная модель, чем отдельная категоризация, но основная идея заключается в том, что DSL предназначены для использования в одной конкретной области, тогда как GPL могут использоваться в разных областях. Например, код на HCL, который вы пишете для Terraform, работает только с Terraform и ограничивается функциями, поддерживаемыми Terraform, такими как развертывание инфраструктуры. Это главное отличие от использования GPL, такого как JavaScript в Pulumi, код на котором может не только управлять инфраструктурой с помощью библиотек Pulumi, но и выполнять практически любые другие задачи программирования, например запускать веб-приложения (фактически Pulumi предлагает API автоматизации, который можно использовать для встраивания Pulumi в код вашего приложения), выполнять сложную логику управления (циклы, условия и абстракцию, которые проще реализовать в GPL, чем в DSL), запускать различные проверки и тесты, интегрироваться с другими инструментами и API и т. д.

DSL имеют несколько преимуществ перед GPL.

- *Проще в освоении.* Поскольку DSL по своему замыслу предназначены для использования в одной конкретной предметной области, они, как правило, имеют меньший размер и более простой синтаксис, чем GPL, поэтому они проще в освоении. Большинство разработчиков смогут изучить HCL в Terraform быстрее, чем, скажем, Java.
- *Более четкие и лаконичные.* Поскольку DSL создаются для одной конкретной цели и все ключевые слова в языке подчинены этой цели, код, написанный на DSL, обычно более компактный и простой для понимания, чем код на GPL, решающий ту же задачу. Код на HCL в Terraform, реализующий развертывание одного сервера в AWS, обычно короче и проще, чем аналогичный код на Java.
- *Более единообразные.* Большинство DSL ограничены в своих возможностях. У этой особенности есть недостатки, о которых я упомяну ниже, но одно из преимуществ заключается в том, что код, написанный на DSL, обычно имеет единообразную и предсказуемую структуру, поэтому в нем легче ориентироваться, чем в коде, написанном на GPL, где каждый разработчик может решить одну и ту же задачу совершенно по-разному. На самом деле развернуть сервер в AWS с помощью Terraform можно только одним способом, но организовать то же самое на Java можно сотней способов.

GPL, в свою очередь, имеют несколько преимуществ перед DSL.

- *Иногда нет необходимости изучать что-то новое.* Поскольку GPL используются во многих областях, есть вероятность, что вам вообще не придется учить новый язык. Особенно это касается Pulumi, поддерживающего несколько самых популярных языков в мире, включая JavaScript, Python и Java. Если вы уже знаете Java, то сможете освоить Pulumi быстрее, чем если бы вам пришлось изучать HCL для использования Terraform.
- *Более широкая экосистема и более зрелые инструменты.* Поскольку GPL используются во многих областях, они имеют гораздо более многочисленные сообщества и более зрелые инструменты, чем типичные DSL. Количество и качество интегрированных сред разработки, библиотек, шаблонов, инструментов тестирования и прочего для Java значительно превосходят то, что доступно для Terraform.
- *Более мощные.* GPL, в принципе, могут использоваться для решения практически любых задач программирования, поэтому они предлагают гораздо более широкие возможности и функциональность, чем DSL. Определенные задачи, такие как логика управления (циклы и условия), автоматическое тестирование, повторное использование кода, абстракция и интеграция с другими инструментами, на Java реализуются намного проще, чем на HCL в Terraform.

Наличие или отсутствие ведущего сервера

Chef и Puppet по умолчанию требуют наличия *ведущего сервера* для хранения состояния инфраструктуры и распространения обновлений. Каждый раз, чтобы что-то обновить в инфраструктуре, необходимо использовать клиента (например, утилиту командной строки), чтобы передать новые команды ведущему серверу, который накатит обновления на остальные серверы или позволит им их загружать на регулярной основе.

Наличие ведущего сервера дает несколько преимуществ. Во-первых, это единое централизованное место, где можно просматривать и администрировать состояние 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 ведущие серверы умеют работать непрерывно, в фоновом режиме, обеспечивая соблюдение вашей конфигурации. Таким образом, если кто-то поменяет состояние узла вручную, ведущий сервер может откатить это изменение, предотвращая тем самым дрейф конфигурации.

Однако использование ведущего сервера имеет серьезные недостатки.

- *Дополнительная инфраструктура.* Вам нужно развернуть дополнительный сервер или даже кластер дополнительных серверов (для высокой доступности и масштабируемости).
- *Обслуживание.* Ведущий сервер нуждается в обслуживании, обновлении, резервном копировании, мониторинге и масштабировании.
- *Безопасность.* Вам нужно сделать так, чтобы клиент мог общаться с ведущим 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 и Puppet есть поддержка нескольких режимов работы без ведущих 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 ведущего сервера). Это существенно снижает количество вовлеченных компонентов, но, как вы увидите в следующем подразделе, все равно оставляет без ответа ряд вопросов, особенно касающихся выделения серверов и установки самих агентов.

У Ansible, CloudFormation, Heat, Terraform и Pulumi по умолчанию нет ведущего сервера. Или, точнее говоря, некоторые из них работают с ведущим 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 ведущими 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 дополнительные инфраструктура или механизмы аутентификации (то есть вы просто используете свои SSH-ключи).

Наличие или отсутствие агентов

Chef и Puppet требуют установки своих *агентов* (вроде Chef Client, Puppet Agent) 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 выделение 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 из образов AMI, в которых уже установлен агент). У других предусмотрен специальный подготовительный процесс, в ходе которого выполняются однократные команды для выделения серверов (с помощью API облачного провайдера) и установки на них агентов (по SSH).
- *Необходимо выполнять обслуживание.* Вам следует тщательно и регулярно обновлять программное обеспечение агента, чтобы синхронизировать его с ведущим сервером (если таковой имеется). Также нужно следить за агентами и перезапускать их, если они выйдут из строя.
- *Следует обеспечить безопасность.* Если ПО агента загружает конфигурацию с ведущего сервера (или какого-то другого сервера, если у вас нет ведущего), вам придется открыть исходящие порты на каждом узле. Если ведущий 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 и Puppet предлагают поддержку разных режимов работы без агента, но все они выглядят так, будто о них вспомнили задним числом, и ни одному из них не доступен полный набор возможностей по управлению конфигурацией.

В связи с этим в реальных условиях стандартный или идиоматический способ использования Chef и Puppet подразумевает наличие агента и, как правило, ведущего сервера (рис. 1.7).

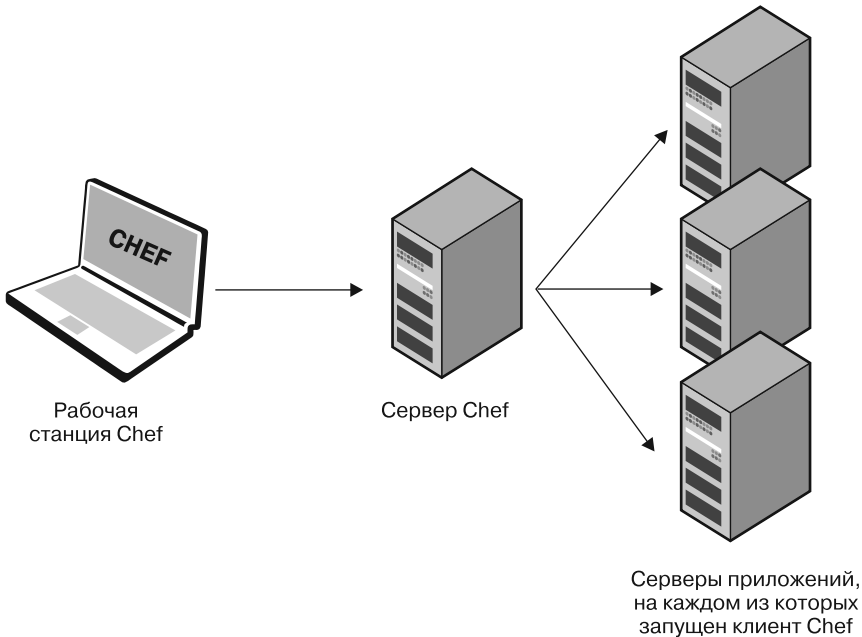


Рис. 1.7. Типичная архитектура Chef и Puppet состоит из множества компонентов. Например, в стандартной конфигурации клиент Chef, запущенный на вашем компьютере, общается с ведущим сервером Chef, который разворачивает изменения, взаимодействуя с клиентами Chef на всех остальных серверах

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 ведущий (-е) сервер (-ы), или в процесс взаимодействия между ведущим (-и) сервером (-ами) и клиентами или остальными серверами, или же...

Ansible, CloudFormation, Heat, Terraform и Pulumi 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 запускаете 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.

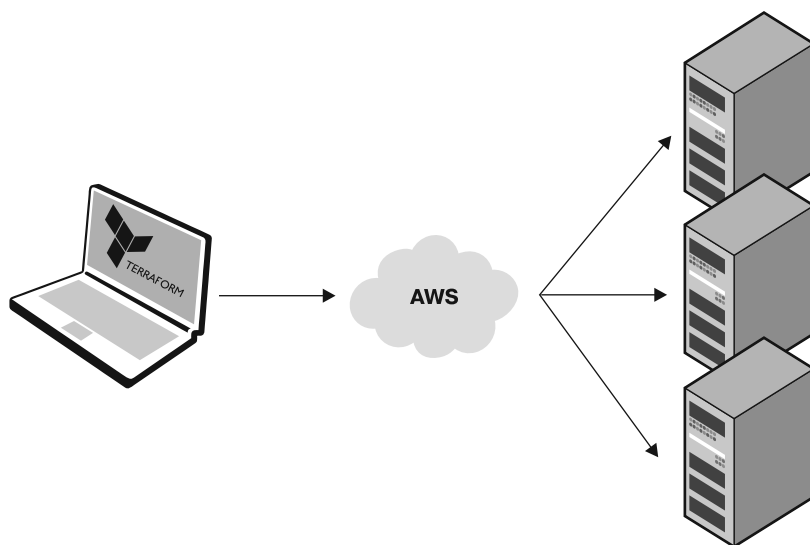


Рис. 1.8. Архитектура Terraform не требует ни агентов, ни ведущего сервера. Вам достаточно запустить клиент Terraform, а тот уже позаботится обо всем остальном, используя API облачных провайдеров, таких как AWS

Платные и бесплатные предложения

CloudFormation и OpenStack Heat доступны совершенно бесплатно: развертываемые с их помощью ресурсы могут стоить денег, но вы ничего не платите за использование самих инструментов. Terraform, Chef, Puppet, Ansible и Pulumi имеют платные и бесплатные версии: например, вы можете применять бесплатную версию Terraform с открытым исходным кодом отдельно или использовать ее с платным продуктом Terraform Cloud. Обсуждение цен, форматов дистрибутивов и возможных вариантов платных версий выходит за рамки этой книги. Единственный вопрос, на котором я хочу акцентировать внимание: настолько ли ограничена бесплатная версия, что в реальных производственных сценариях вы вынуждены будете использовать платное предложение.

Сразу внесем ясность: нет ничего плохого в том, что компания предлагает платное обслуживание для одного из своих инструментов. На самом деле, если вы используете эти инструменты в производстве, я настоятельно рекомендую обратить внимание на платные услуги, потому что многие из них стоят своих денег. Однако вы должны понимать, что платные сервисы вам неподконтрольны — их поддержка может прекратиться, они могут быть приобретены другими компаниями (например, Chef, Puppet и Ansible прошли такой этап приобретения, оказавший существенное влияние на их предложения платных продуктов), изменить свою модель ценообразования (например, Pulumi изменила цены в 2021 году, что принесло пользу некоторым пользователям, но увеличило цены примерно в десять раз для других), изменить продукт или полностью прекратить его выпуск — поэтому важно знать, будет ли выбранный вами инструмент работать, даже если по какой-то причине вы не сможете воспользоваться одной из платных услуг.

По моему опыту, бесплатные версии Terraform, Chef, Puppet и Ansible можно с успехом использовать в производственных целях; платные услуги способны сделать эти инструменты еще лучше, но, если они окажутся недоступны, вы все равно сможете продолжить работу. Pulumi, напротив, сложнее использовать в производстве без платного предложения, известного как Pulumi Service.

Ключевая часть управления инфраструктурой как кодом — управление состоянием (о том, как Terraform управляет состоянием, вы узнаете в главе 3), а Pulumi по умолчанию использует Pulumi Service в качестве серверной части для хранения состояния. Вы можете переключиться на серверные решения для хранения состояния, такие как AWS S3, Azure Blob Storage, Google Cloud Storage, но в документации по серверной части Pulumi (<https://www.pulumi.com/docs/intro/concepts/state/>) объясняется, что только Pulumi Service поддерживает контрольные точки транзакций (они обеспечивают отказоустойчивость и возможность восстановления), параллельную блокировку состояния (для предотвращения повреждения состояния инфраструктуры в командной среде), а также шифрование состояния при передаче и хранении. На мой взгляд, без этих функций Pulumi имеет меньшую практическую ценность в любой производственной среде (то есть с более чем одним разработчиком), поэтому, если вы собираетесь использовать Pulumi, вам, скорее всего, придется воспользоваться платным предложением Pulumi Service.

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. От величины сообщества зависит количество людей, помогающих проекту, разнообразие

плагинов и расширений, простота получения помощи в Интернете (например, благодаря статьям или вопросам на StackOverflow) и проблематичность найма 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 июне 2022 года. Здесь учитывается, имеет ли инструмент открытый исходный код, с какими провайдерами он совместим, общее количество участников проекта и звезд на GitHub, сколько открытых библиотек доступно для этого инструмента и количество вопросов о нем на StackOverflow¹.

Таблица 1.1. Сравнение сообществ IaC

	Код	Облака	Участники	Звезды	Библиотеки	StackOverflow
Chef	Открытый	Все	640	6910	3695 ²	8295
Puppet	Открытый	Все	571	6581	6871 ³	3996
Ansible	Открытый	Все	5328	53 479	31 329 ⁴	22 052
Pulumi	Открытый	Все	1402	12 723	15 ⁵	327
CloudFormation	Закрытый	AWS	?	?	369 ⁶	7252
Heat	Открытый	Все	395	379	0 ⁷	103
Terraform	Открытый	Все	1621	33 019	9641 ⁸	13 370

¹ Большая часть этих данных, включая количество участников и звезд, была взята из репозитория открытого исходного кода (в основном на GitHub) каждого отдельного инструмента. Поскольку проект CloudFormation имеет закрытый исходный код, некоторые из этих сведений для него недоступны.

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

³ Количество модулей в Puppet Forge (<https://forge.puppet.com/>).

⁴ Количество универсальных ролей в Ansible Galaxy (<https://galaxy.ansible.com/>).

⁵ Количество пакетов в Pulumi Registry (<https://www.pulumi.com/registry/>).

⁶ Количество шаблонов в AWS Quick Starts (<https://aws.amazon.com/quickstart/>).

⁷ Мне не удалось найти ни одной коллекции шаблонов Heat, подготовленной сообществом.

⁸ Это количество модулей в Terraform Registry (<https://registry.terraform.io/>).

Естественно, это неидеальное сравнение равнозначных показателей. Некоторые инструменты имеют больше одного репозитория. Например, в Terraform в 2017 году выделили код для разных провайдеров (AWS, Google Cloud, Azure и т. д.) в отдельные репозитории, поэтому оценка активности сообщества в табл. 1.1 значительно занижена, а некоторые используют другие варианты для общения в сообществе, помимо StackOverflow, и т. д.

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 явно лидируют проекты Ansible 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 (обратите внимание, что Pulumi не включен в эту таблицу, так как не участвовал в сравнении в первом издании этой книги).

Таблица 1.2. Как изменились сообщества IaC с сентября 2016 по июнь 2022 года

	Код	Облака	Участники	Звезды	Библиотеки	StackOverflow
Chef	Открытый	Все	+34 %	+56 %	+21 %	+98 %
Puppet	Открытый	Все	+32 %	+58 %	+55 %	+51 %
Ansible	Открытый	Все	+258 %	+183 %	+289 %	+507 %
CloudFormation	Закрытый	AWS	?	?	+54 % ¹	+1083 %
Heat	Открытый	Все	+40 %	+34 %	0	+98 %
Terraform	Открытый	Все	+148 %	+476 %	+24 003 %	+10 106 %

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

¹ В предыдущих изданиях я использовал шаблоны CloudFormation из репозитория awslabs на GitHub, но сейчас их, похоже, больше нет, поэтому здесь я использовал AWS Quick Starts. Как результат, эти цифры нельзя сравнивать напрямую.

Выбор между зрелостью и новизной

Еще один ключевой фактор при выборе любой технологии — ее зрелость. Это давно существующая технология, и ее модели, практики, проблемы и режимы использования хорошо понятны? Или это новая технология, которую вам придется осваивать с нуля? В табл. 1.3 приводятся год выпуска начальной версии каждого инструмента IaC, их версии на данный момент (по состоянию на июнь 2022 года), а также моя субъективная оценка зрелости каждого из инструментов IaC.

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

	Начальный выпуск	Текущая версия	Оценка зрелости
Chef	2009	17.10.3	Высокая
Puppet	2005	7.17.0	Высокая
Ansible	2012	5.9.0	Средняя
Pulumi	2017	3.34.1	Низкая
CloudFormation	2011	???	Средняя
Heat	2012	18.0.0	Низкая
Terraform	2014	1.2.3	Средняя

Здесь сравниваются не совсем равнозначные вещи: возраст сам по себе не определяет зрелость; не влияет и большой номер версии (разные инструменты имеют разные схемы управления версиями), но некоторые тенденции бросаются в глаза. Pulumi — самый молодой инструмент IaC в этом списке и, пожалуй, наименее зрелый: это становится очевидно при попытке найти документацию, передовые практики, модули, написанные сообществом, и т. д. Современный Terraform стал более зрелым: набор его инструментов расширился, практика применения стала более понятной, доступно гораздо больше обучающих ресурсов (включая эту книгу!), и теперь, по достижении рубежа 1.0.0, он стал значительно стабильнее и надежнее, чем на момент выхода первого и второго изданий этой книги.

Использование нескольких инструментов вместе

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 вариант, подходящий для ваших задач.

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

Выделение ресурсов плюс управление конфигурацией

Пример: Terraform и Ansible. Terraform используется для развертывания всей внутренней инфраструктуры, включая топологию сети (то есть виртуальные закрытые облака (virtual private cloud, 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.

Выделение ресурсов плюс шаблонизация серверов

Пример: Terraform и Packer. Packer используется для упаковки 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.

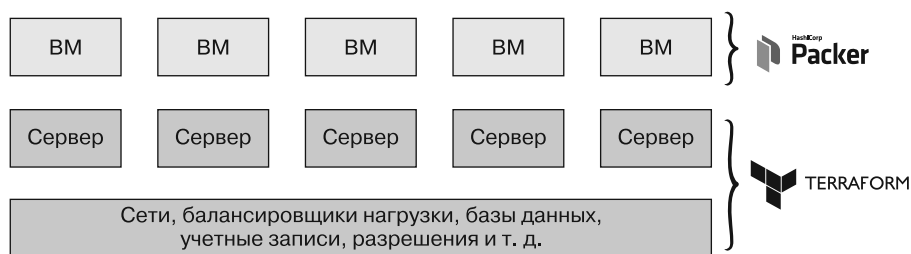


Рис. 1.10. Совместное применение Terraform и Packer

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 и простую в обслуживании инфраструктуру. 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.

Выделение ресурсов плюс шаблонизация серверов плюс оркестрация

Пример: Terraform, Packer, Docker и Kubernetes. Packer используется для создания образов ВМ с установленными агентами 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, в котором вы будете запускать и администрировать свои приложения в виде контейнеров Docker (рис. 1.11).

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.

Пример этого подхода вы увидите в главе 7.

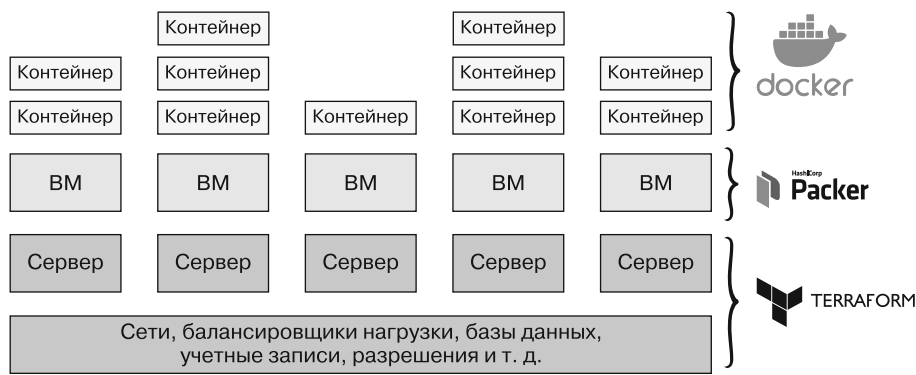


Рис. 1.11. Совместное использование Terraform, Packer, Docker и Kubernetes

Резюме

Ниже показана сводная таблица наиболее 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 ведущего сервера, а Puppet поддерживает неизменяемую инфраструктуру).

В компании Gruntwork нам нужно было открытое, не привязанное к конкретному облаку средство выделения ресурсов, которое поддерживает неизменяемую инфраструктуру, декларативный язык, архитектуру, не требующую ведущих серверов и агентов, и имеет большое сообщество и зрелую кодовую базу. Как показывает табл. 1.4, проект Terraform хоть и не идеальный, но именно он лучше всего удовлетворяет нашим критериям.

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

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

	Chef	Puppet	Ansible	Pulumi	CloudFormation	Heat	Terraform
Код	Открытый	Открытый	Открытый	Открытый	Закрытый	Открытый	Открытый
Облако	Все	Все	Все	Все	AWS	Все	Все
Тип	Управление конфигурацией	Управление конфигурацией	Управление конфигурацией	Выделение ресурсов	Выделение ресурсов	Выделение ресурсов	Выделение ресурсов
Инфраструктура	Изменяемая	Изменяемая	Изменяемая	Неизменяемая	Неизменяемая	Неизменяемая	Неизменяемая
Подход	Процедурный	Декларативный	Процедурный	Декларативный	Декларативный	Декларативный	Декларативный
Язык	GPL	DSL	DSL	GPL	DSL	DSL	DSL
Ведущий сервер	Есть	Есть	Нет	Нет	Нет	Нет	Нет
Агент	Есть	Есть	Нет	Нет	Нет	Нет	Нет
Платная поддержка	По выбору	По выбору	По выбору	Обязательна	—	—	По выбору
Сообщество	Большое	Большое	Огромное	Маленькое	Маленькое	Маленькое	Огромное
Зрелость	Высокая	Высокая	Средняя	Низкая	Средняя	Низкая	Средняя

Приступаем к работе с Terraform

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

Terraform умеет выделять инфраструктуру как в публичных облаках, вроде Amazon Web Services (AWS), Azure, Google Cloud и DigitalOcean, так и в закрытых 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 32 %, что больше, чем у трех ближайших конкурентов (Microsoft, Google, IBM), вместе взятых¹.
- AWS предоставляет широчайший спектр надежных и масштабируемых облачных сервисов, 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 в кластере².

¹ Источник: Canalys, 2021 (<https://www.canalys.com/newsroom/global-cloud-services-q3-2021>).

² Если терминология AWS кажется вам запутанной, то обязательно почитайте статью AWS in Plain English (<https://bit.ly/2KuLD4a>).

- В первый год использования AWS предлагает щедрый бесплатный тариф, которого достаточно для опробования всех этих примеров без денежных затрат¹. Даже если вы уже исчерпали свои бесплатные кредиты, работа с примерами из книги все равно обойдется вам 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 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 <https://github.com/brikis98/terraform-up-and-running-code>.

Подготовка учетной записи в AWS

Если у вас нет учетной записи в AWS, пройдите на страницу <https://aws.amazon.com/> и зарегистрируйтесь. Сразу после регистрации вы войдете в систему с привилегиями *суперпользователя root*. Эта учетная запись позволяет делать что угодно, поэтому с точки зрения безопасности ее лучше не использовать регулярно. Она будет нужна *только* для создания других пользовательских учетных записей с ограниченными правами, на одну из которых вы должны немедленно переключиться².

¹ За подробностями обращайтесь к документации AWS Free Tier (<https://aws.amazon.com/free/>).

² Подробнее о рекомендуемых подходах к управлению пользователями в AWS можно почитать по ссылке <https://amzn.to/2lvJ8Rf>.

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 установите флажок **Access key – Programmatic access** (Ключ доступа — программный доступ), 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.

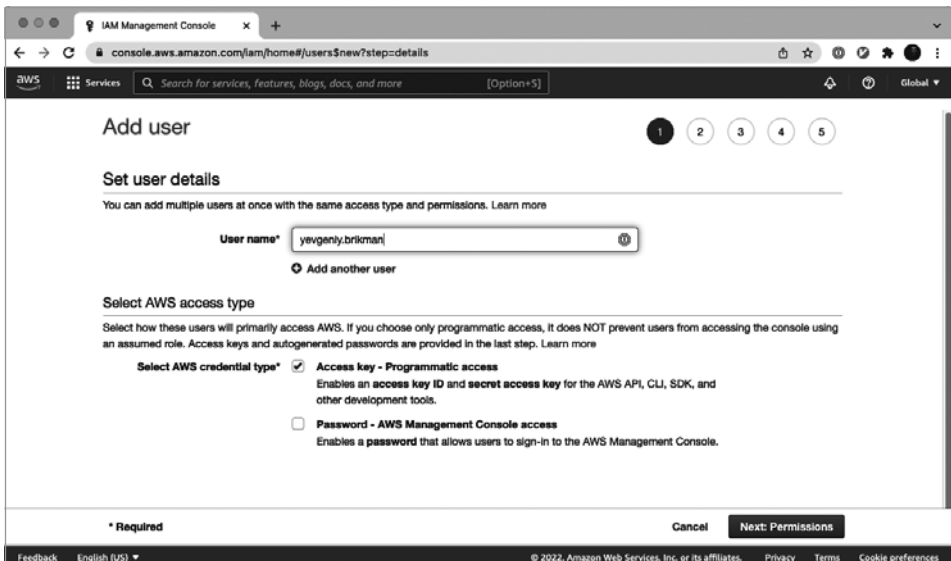


Рис. 2.1. Создание нового пользователя IAM

Нажмите кнопку **Next** (Далее). AWS предложит дать пользователю разрешения. По умолчанию новые пользователи IAM не имеют никаких разрешений и не могут выполнять никаких операций в учетной записи AWS. Чтобы выдать пользователю IAM какие-либо разрешения, нужно 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 собственные политики или обойтись уже готовыми, которые называются *управляемыми политиками*¹.

¹ Больше о политиках IAM можно узнать на сайте AWS (<https://amzn.to/2lQs1MA>).

Для опробования примеров из этой книги просто назначьте своему пользователю IAM управляемую политику `AdministratorAccess` (рис. 2.2)¹.

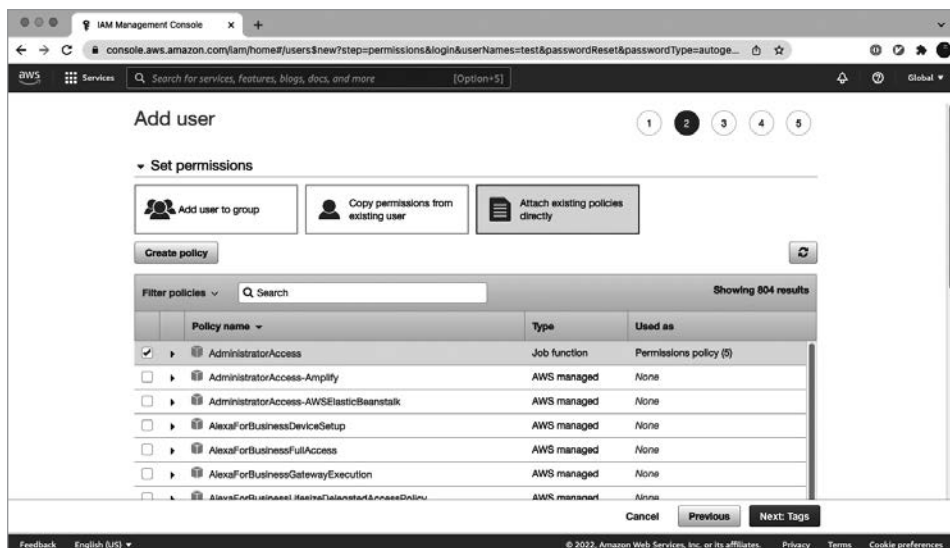


Рис. 2.2. Назначение управляемой политики `AdministratorAccess` новому пользователю IAM

Нажмите кнопку `Next` (Далее) еще пару раз и затем кнопку `Create` (Создать). AWS покажет вам закрытые учетные данные этого пользователя, которые, как видно на рис. 2.3, состоят из *ID ключа доступа* (`Access key ID`) и *закрытого ключа доступа* (`Secret access key`). Их следует немедленно сохранить, поскольку их больше никогда не покажут, а они вам 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 macOS Keychain) и никогда никому не давайте.

Сохранив свои учетные данные, нажмите кнопку `Close` (Закрыть). Теперь вы готовы приступить к использованию Terraform.

¹ Я предполагаю, что вы будете опробовать примеры из этой книги в учетной записи AWS, созданной исключительно для обучения и тестирования, поэтому выдача широких разрешений управляемой политики `AdministratorAccess` не представляет большого риска. Если вы собираетесь опробовать примеры в более конфиденциальной среде (чего я не рекомендую!) и умеете создавать собственные политики IAM, то более ограниченный набор разрешений вы найдете в примерах кода для этой книги (<https://github.com/brikis98/terraform-up-and-running-code/blob/master/code/json/02-intro-to-terraform-syntax/iam-policy.json>).

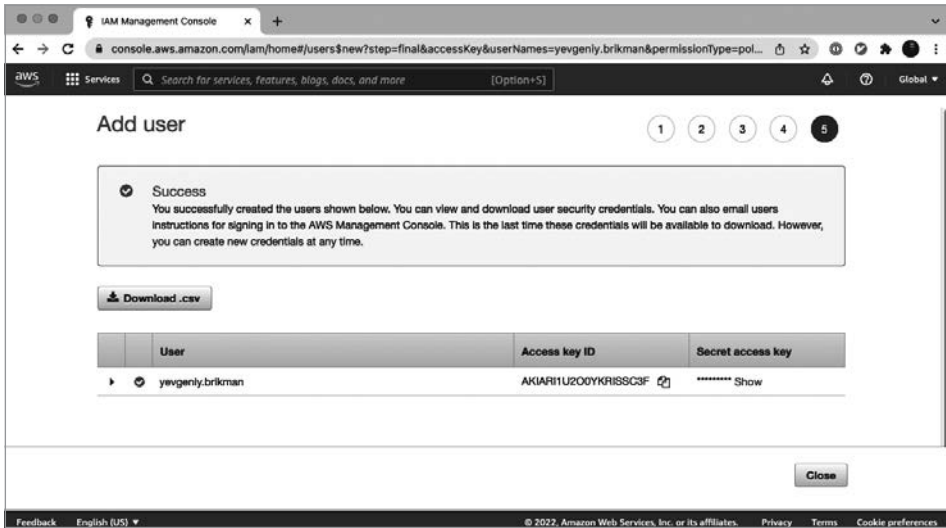


Рис. 2.3. Храните свои учетные данные AWS в надежном месте. Никому их не показывайте (не волнуйтесь, те, что на снимке экрана, — ненастоящие)



Замечание о виртуальных закрытых облаках по умолчанию

Если вы используете существующую учетную запись AWS, у нее уже должно быть облако VPC по умолчанию. VPC (Virtual Private Cloud — виртуальное закрытое 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

Самый простой способ установить Terraform — использовать диспетчер пакетов операционной системы. Например, в macOS, если вы пользуетесь диспетчером Homebrew, можно выполнить такие команды:

```
$ brew tap hashicorp/tap
$ brew install hashicorp/tap/terraform
```

В Windows, если вы пользуетесь диспетчером Chocolatey, запустите:

```
$ choco install terraform
```

Описание способов установки с помощью диспетчеров пакетов, в том числе в Linux, вы найдете в документации Terraform (<https://learn.hashicorp.com/tutorials/terraform/install-cli?in=terraform/aws-get-started>).

Terraform также можно установить вручную. Для этого перейдите на домашнюю страницу проекта (<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 с именем `terraform`, путь к которому следует добавить в переменную среды `PATH`.

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

```
$ terraform
Usage: terraform [global options] <subcommand> [args]
```

```
The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
```

Main commands:

<code>init</code>	Prepare your working directory for other commands
<code>validate</code>	Check whether the configuration is valid
<code>plan</code>	Show changes required by the current configuration
<code>apply</code>	Create or update infrastructure
<code>destroy</code>	Destroy previously-created infrastructure

(...)

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=(ваш ID ключа доступа)
$ export AWS_SECRET_ACCESS_KEY=(ваш закрытый ключ доступа)
```

А вот так то же самое можно сделать в командном окне Windows:

```
$ set AWS_ACCESS_KEY_ID=(ваш ID ключа доступа)
$ set AWS_SECRET_ACCESS_KEY=(ваш закрытый ключ доступа)
```

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 команду `aws configure`, или роли IAM, которые можно назначить почти любому ресурсу в AWS. Подробнее об этом рассказывается в статье *A Comprehensive Guide to Authenticating to AWS on the Command Line* (<https://bit.ly/2M11muR>). Дополнительную информацию об аутентификации у поставщиков Terraform вы также найдете в главе 6.

Развертывание одного сервера

Код Terraform пишется на *языке конфигурации HashiCorp* (HashiCorp Configuration Language или HCL) и хранится в файлах с расширением `.tf`¹. 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 практически в любом текстовом редакторе. Если поискать, можно найти подсветку синтаксиса Terraform во многих редакторах (обратите внимание, что вам, возможно, придется 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 шагом 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 содержимым:

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

Это определение сообщит Terraform, 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

¹ Код Terraform можно также писать на чистом JSON и хранить в файлах с расширением `.tf.json`. Больше о синтаксисе HCL и JSON можно узнать в документации Terraform (<https://www.terraform.io/language/syntax/configuration>).

центров, известных как *зоны доступности*: например, `us-east-2a`, `us-east-2b` и т. д.¹. У этого провайдера можно настроить множество других параметров, но пока не будем усложнять задачу, а более подробно рассмотрим конфигурацию провайдера в главе 7.

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 (например, `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-0fb653ca2d3203ac1"  
  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. Код, представленный выше, назначает параметру `ami` идентификатор AMI Ubuntu 20.04 в `us-east-2`. Этот образ можно использовать бесплатно. Обратите внимание, что идентификаторы AMI различаются для разных регионов AWS, поэтому, если вы измените значение параметра `region` на какое-либо другое, отличное от `us-east-2`, вам понадобится самостоятельно найти соответствующий идентификатор AMI Ubuntu для этого региона² и скопировать его в параметр `ami`. В главе 7 вы увидите, как автоматически получить идентификатор AMI.

¹ Дополнительную информацию о регионах и зонах доступности AWS можно найти на веб-сайте AWS (<https://bit.ly/1NATGqS>).

² Найти идентификаторы AMI на удивление сложно, но можно, как описано здесь: <https://blog.gruntwork.io/locating-aws-ami-owner-id-and-image-name-for-packer-builds-7616fe46b49a>.

- **instance_type.** Тип сервера EC2, который нужно запустить. У каждого типа есть свой объем ресурсов: количество процессоров, объем памяти и дискового пространства, пропускная способность сети. Все доступные варианты перечислены на соответствующей странице *EC2 Instance Types* по адресу <https://amzn.to/2H49EON>. В примере выше указан тип `t2.micro`, который имеет один виртуальный процессор, 1 Гбайт памяти и доступен в бесплатном тарифе AWS.



Читайте 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 регулярно сверяйтесь с документацией, чтобы посмотреть, какие ресурсы доступны и как их использовать. Например, вот документация для ресурса `aws_instance` по адресу <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance>. Несмотря на свой многолетний опыт использования Terraform, я продолжаю обращаться к этой документации по несколько раз в день!

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

```
$ terraform init
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

- Reusing previous version of hashicorp/aws from the dependency lock file
- Using hashicorp/aws v4.19.0 from the shared cache directory

```
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 команду `terraform init`, чтобы он просканировал ваш код, определил, с какими провайдерами вы работаете, и загрузил 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 `terraform plan`:

```
$ terraform plan
```

```
(...)
```

Terraform will perform the following actions:

```
# aws_instance.example will be created
+ resource "aws_instance" "example" {
  + ami                        = "ami-0fb653ca2d3203ac1"
  + 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. Своим выводом 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 можно видеть, что Terraform планирует создать лишь один сервер EC2 и ничего другого — именно то, что нам нужно.

Чтобы инициировать создание сервера, нужно выполнить команду `terraform apply`:

```
$ terraform apply
```

```
(...)
```

Terraform will perform the following actions:

```
# aws_instance.example will be created
+ resource "aws_instance" "example" {
  + ami                        = "ami-0fb653ca2d3203ac1"
```

```

+ 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 10), 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 только что 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.

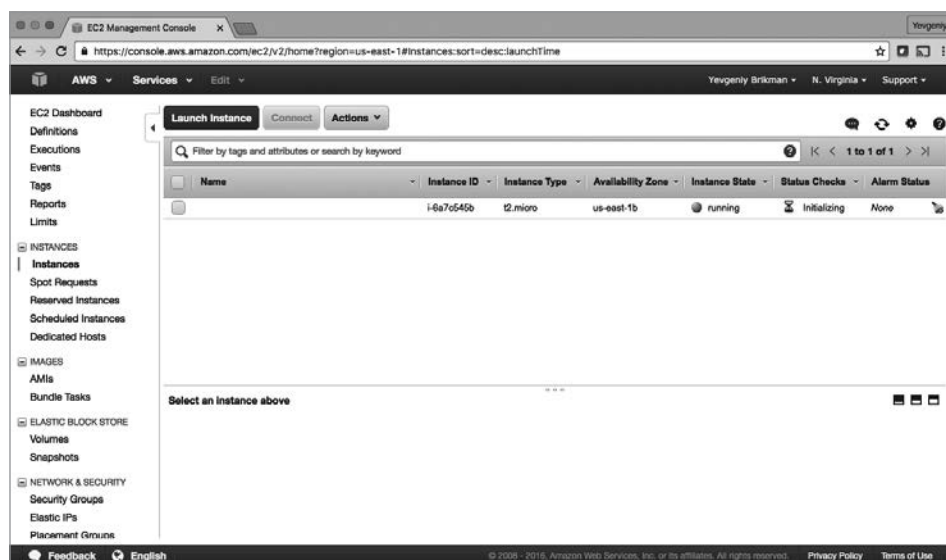


Рис. 2.4. Один сервер EC2

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

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

  tags = {
    Name = "terraform-example"
  }
}
```

Чтобы увидеть последствия этих изменений, еще раз выполните команду `terraform apply`:

```
$ 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-0fb653ca2d3203ac1"
  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 *Refreshing state...* (Обновление состояния...) в выводе 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 декларативного кода перед процедурным; см. раздел «Сравнение Terraform с другими средствами IaC» главы 1). В предыдущем сравнении видно, что Terraform предполагает создать один тег *Name*. Это именно то, что вам нужно, поэтому введите *yes* и нажмите клавишу *Enter*.

Обновив консоль EC2, вы увидите что-то похожее на рис. 2.5.

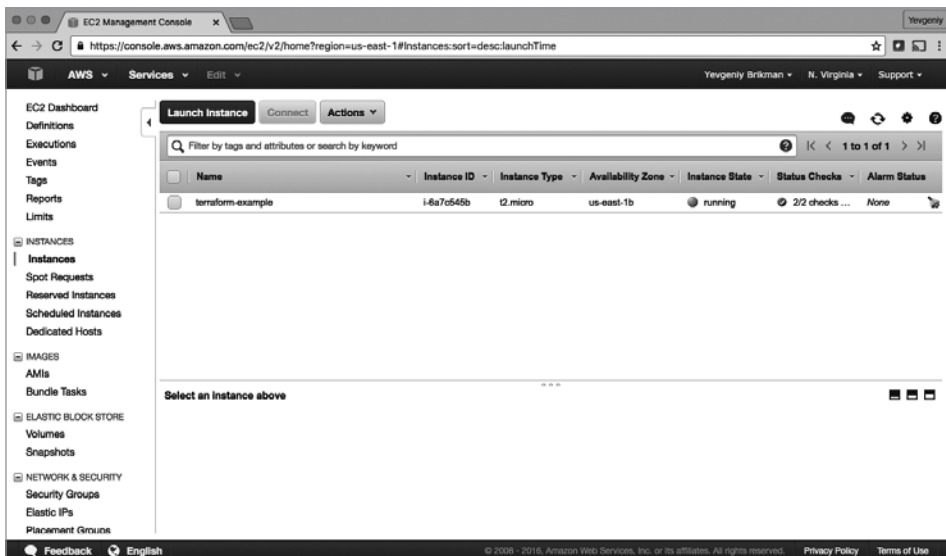


Рис. 2.5. У сервера EC2 теперь есть тег с именем

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 показано, как создать локальный Git-репозиторий и сохранить в нем конфигурационный файл Terraform и файл блокировки (с файлом блокировки вы познакомитесь в главе 8; а пока просто знайте, что его следует добавить в систему управления версиями вместе с вашим кодом):

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

Также следует создать файл `.gitignore` с таким содержимым:

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

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

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

Чтобы поделиться этим кодом со своими коллегами, создайте общий Git-репозиторий, к которому вы все сможете 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 конечной точки под названием `origin` для своего локального Git-репозитория:

```
git remote add origin git@github.com:<ВАШЕ_ИМЯ_ПОЛЬЗОВАТЕЛЯ>/<ИМЯ_ВАШЕГО_РЕПОЗИТОРИЯ>.git
```

Теперь, когда вы захотите поделиться своими изменениями с коллегами, *отправьте* их в репозиторий `origin`:

```
git push origin master
```

Если вы захотите посмотреть, какие изменения внесли ваши коллеги, *извлеките* их из репозитория `origin`:

```
git pull origin master
```

При дальнейшем чтении этой книги и в целом при применении Terraform не забывайте регулярно фиксировать (`git commit`) и отправлять (`git push`) свои изме-

нения. Это позволит работать над кодом совместно с членами вашей команды. К тому же все изменения в вашей инфраструктуре будут записываться в журнал фиксаций, что придется очень кстати в случае отладки. Больше о командной работе с Terraform вы узнаете в главе 10.

Развертывание одного веб-сервера

Следующий шаг: развертывание веб-сервера на только что созданном виртуальном сервере. Мы попытаемся развернуть простейшую веб-архитектуру: один веб-сервер, способный отвечать на HTTP-запросы (рис. 2.6).

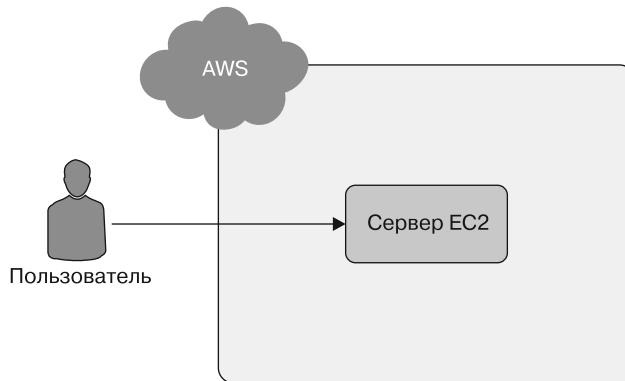


Рис. 2.6. Начнем с простой архитектуры: одного веб-сервера, запущенного в AWS, который отвечает на HTTP-запросы

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

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

Это Bash-скрипт, записывающий текст `Hello, World` в файл `index.html`, который затем раздается веб-сервером через порт 8080, запущенным с использованием инструмента под названием `busybox` (в Ubuntu установлен по умолчанию). Я указал команду `busybox` между `nohup` и `&`, чтобы веб-сервер постоянно работал в фоне даже после завершения Bash-скрипта.

¹ На GitHub по адресу <https://bit.ly/2OIrr2> можно найти полезный список однострочных HTTP-серверов.



Номера портов

В этом примере вместо стандартного HTTP-порта 80 используется 8080. Дело в том, что все порты меньше 1024 требуют администраторских привилегий. Использовать эти порты небезопасно, 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 обслуживал порт 80 и перенаправлял трафик на порты ваших серверов.

Как запустить этот скрипт на сервере EC2? Как уже упоминалось в подразделе «Средства шаблонизации серверов» главы 1, 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*, можно обойтись стандартным образом Ubuntu 20.04 и запустить скрипт «Hello, World» в рамках конфигурации *пользовательских данных* сервера EC2. Вы можете передать параметр *user_data* в код Terraform и все, что в нем указано, будет выполнено при загрузке сервера: это может быть скрипт командной оболочки или директива *cloud-init*. Вариант со скриптом показан ниже:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

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

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example"
  }
}
```

Вот пара замечаний к предыдущему коду.

1. `<<-EOF` и `EOF` — элементы синтаксиса *встроенных документов* (heredoc), который позволяет создавать многострочные строковые литералы без использования множества символов `\n` перевода строки.
2. Параметру `user_data_replace_on_change` присвоено значение `true`, поэтому, когда вы измените параметр `user_data` и запустите `apply`, Terraform завершит

работу исходного экземпляра и запустит совершенно новый. По умолчанию Terraform обновляет исходный экземпляр на месте, но, поскольку пользовательские данные запускаются только при первой загрузке, а исходный экземпляр уже прошел этот процесс, необходимо принудительно создать новый экземпляр, чтобы гарантировать выполнение сценария в пользовательских данных.

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 через порт 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 указывает, что эта группа разрешает принимать TCP-запросы через порт 8080 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 запросы через порт 8080 с любого IP¹.

Создания группы безопасности как таковой недостаточно. Нужно, чтобы сервер EC2 ее использовал. Для этого следует передать ее идентификатор через аргумент `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-0fb653ca2d3203ac1"`) и числа (скажем, `5`). Terraform поддерживает много других разновидностей выражений, которые будут встречаться на страницах этой книги.

¹ Больше о принципе работы CIDR можно узнать на странице в Википедии по адресу <https://bit.ly/2l8Ki9g>. Для преобразования диапазонов IP-адресов в CIDR и обратно можно использовать веб-калькулятор <https://cidr.xyz/> или установить команду `ipcalc` и вызывать ее в своем терминале.

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-0fb653ca2d3203ac1"
  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

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example"
  }
}
```

Ссылаясь в одном ресурсе на другой, вы создаете *неявную зависимость*. Terraform анализирует такие зависимости, строит их граф и применяет его для автоматического определения порядка создания ресурсов. Например, если бы этот код разворачивался с нуля, система Terraform знала бы, что группу безопасности нужно создать раньше, чем сервер EC2, поскольку последний использует ID этой группы. При желании граф зависимостей можно вывести 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)"
}

```

Она выводит код на языке описания графов под названием DOT, который можно преобразовать в изображение, как показано на рис. 2.7, используя настольное приложение Graphviz или его веб-версию GraphvizOnline по адресу <https://bit.ly/2mPbxmg>.

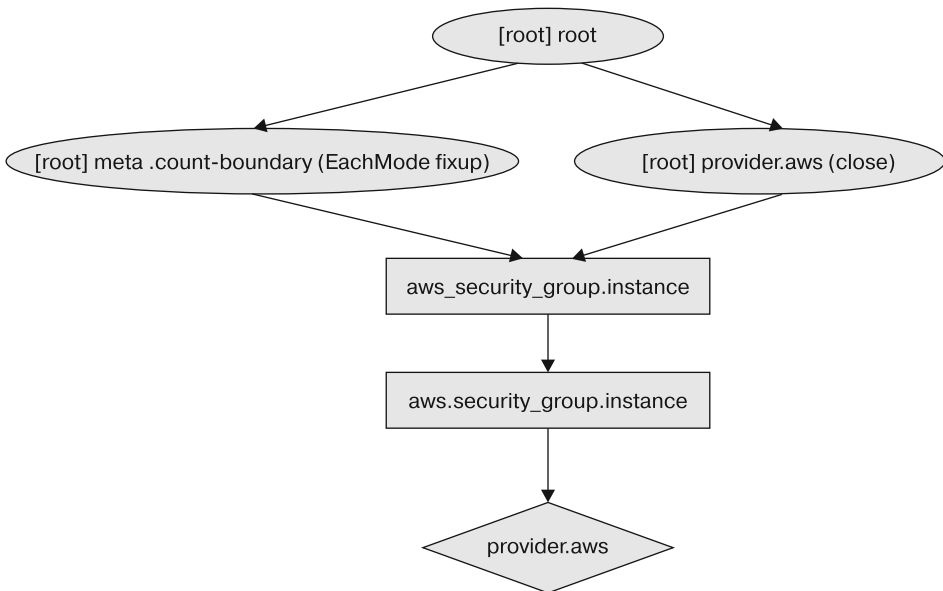


Рис. 2.7. Граф зависимостей для сервера EC2 и его группы безопасности

Выполняя обход дерева зависимостей, Terraform старается по возможности 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.

Выполнив команду `apply`, вы увидите, что Terraform предлагает создать группу безопасности и заменить имеющийся сервер EC2 новым с новыми пользовательскими данными:

```
$ terraform apply
```

```
(...)
```

Terraform will perform the following actions:

```
# aws_instance.example must be replaced
-/+ resource "aws_instance" "example" {
    ami                = "ami-0fb653ca2d3203ac1"
    ~ 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:

-/+ в выводе плана означает «заменить». Чтобы понять, чем продиктовано то или иное изменение, поищите в выводе плана словосочетание `forces replacement` (вынуждает выполнить замену). Изменение многих аргументов ресурса `aws_instance` приводит к замене. Это означает, что имеющийся сервер EC2 будет удален, а его место займет совершенно новый сервер. Это пример парадигмы неизменяемой инфраструктуры, которую мы обсуждали в подразделе «Средства шаблонизации серверов» в главе 1. 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).

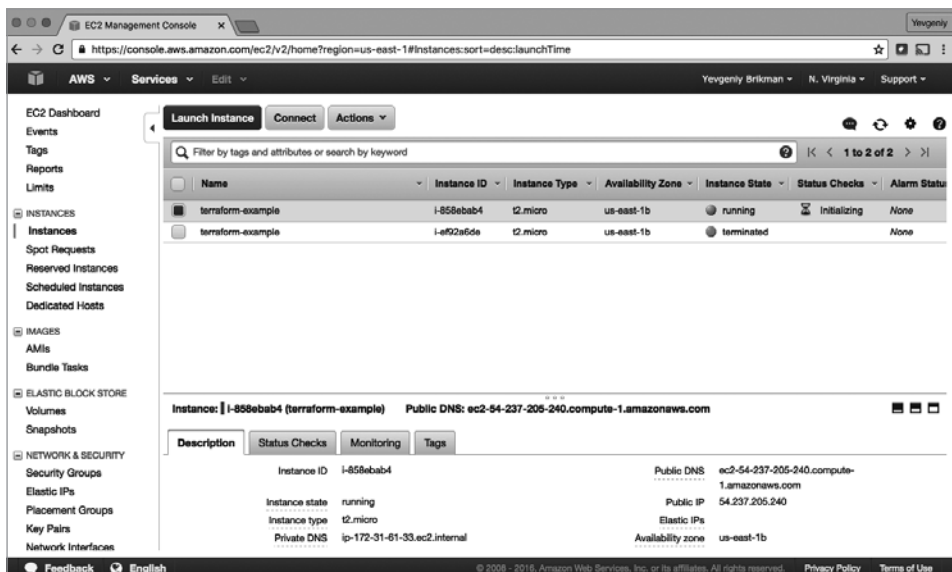


Рис. 2.8. Вместо старого сервера EC2 мы получаем новый, с кодом веб-сервера

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 отправьте HTTP-запрос по этому адресу в порт 8080, используя браузер или утилиту вроде `curl`:

```
$ curl http://<ПУБЛИЧНЫЙ_IP_ЭКЗЕМПЛЯРА_EC2>:8080
Hello, World
```

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



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.

По этой причине 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, как развернуть балансировщик нагрузки).

Развертывание конфигурируемого веб-сервера

Вы, наверное, заметили, что код веб-сервера дублирует порт 8080 в группе безопасности и в конфигурации в пользовательских данных. Это противоречит принципу «*Не повторяйся*» (Don't Repeat Yourself, DRY): каждый элемент информации в системе должен иметь единое, однозначное и достоверное представление¹. Если номер порта указан в двух местах, легко оказаться в ситуации, когда одно из значений будет обновлено, а другое — нет.

¹ Из книги Хант Э., Томас Д. Программист-прагматик. Путь от подмастерья к мастеру (The Pragmatic Programmer).

Чтобы сделать 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 следующие необязательные параметры.

- **description.** Этот параметр всегда желательно указывать для описания использования переменной. 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.** Присвоить значение переменной можно 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`. Всегда старайтесь явно указывать тип, чтобы с помощью механизма ограничений типов выявлять элементарные ошибки. Если тип не указан, Terraform воспринимает значение как `any`.
- **validation.** Позволяет определять собственные правила проверки значения входной переменной, дополняющие базовые проверки типов, например принудительное применение минимальных или максимальных значений к числовым переменным. Примеры таких проверок вы увидите в главе 8.
- **sensitive.** Если присвоить этому параметру значение `true`, Terraform не будет выводить значение переменной при выполнении команд `plan` и `apply`. Используйте этот параметр во всех переменных, через которые передаются конфиденциальные данные, такие как пароли, ключи API и т. д. Подробнее тему конфиденциальности я затрону в главе 6.

Вот пример входной переменной, требующей, чтобы в ней передавалось число:

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

А вот пример переменной, требующей, чтобы в ней передавался список:

```
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 в этом списке 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, требующий, чтобы все значения были строками:

```
variable "map_example" {
  description = "An example of a map in Terraform"
  type       = map(string)

  default = {
    key1 = "value1"
    key2 = "value2"
    key3 = "value3"
  }
}
```

Можно также 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
  }
}
```

Предыдущий пример создаст входную переменную, 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, не соответствующее этому типу, то Terraform немедленно вернет ошибку несоответствия типов. В следующем примере демонстрируется попытка присвоить ключу `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, передайте значение для 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`, то укажите 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, можно добавить выражение другого типа — *ссылку на переменную*. Оно имеет следующий синтаксис:

```
var.<ИМЯ_ПЕРЕМЕННОЙ>
```

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"]
  }
}
```

Неплохо было бы использовать ту же переменную для задания порта в скрипте `user_data`. Сослаться на переменную внутри строкового литерала можно с помощью *выражения интерполяции*, которое имеет следующий синтаксис:

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

Внутри фигурных скобок можно разместить любую корректную ссылку, и Terraform преобразует ее в строку. Например, вот как можно сослаться на переменную `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*, поддерживая такой синтаксис:

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

`NAME` — это имя выходной переменной, а в качестве `VALUE` можно указать любое выражение Terraform, которое вы хотите вернуть. `CONFIG` может иметь следующие дополнительные (и необязательные) параметры.

- **description.** Этот параметр всегда желательно добавлять для описания использования выходной переменной.
- **sensitive.** Если присвоить этому параметру `true`, Terraform не станет выводить значение этой переменной при выполнении команд `plan` и `apply`. Это полезно, когда переменная содержит конфиденциальные данные, такие как пароль или закрытый ключ. Обратите внимание: если выходная переменная ссылается на входную переменную или атрибут ресурса с параметром `sensitive = true`, то вам также необходимо добавить в выходную переменную параметр `sensitive = true`, чтобы указать, что вы намеренно возвращаете секрет.
- **depends_on.** Обычно Terraform автоматически создает граф зависимостей на основе ссылок в коде, но в редких ситуациях приходится давать ему дополнительные подсказки. Например, у вас может быть выходная переменная, возвращающая IP-адрес сервера, но этот IP-адрес не будет доступен до настройки группы безопасности (брандмауэра) для этого сервера. В таких случаях можно с помощью `depend_on` явно указать, что существует зависимость между выходной переменной с IP-адресом и ресурсом группы безопасности.

Например, вместо того, чтобы вручную исследовать консоль ЕС2 в поисках IP-адреса своего сервера, можно получить его через выходную переменную:

```
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 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 `terraform apply` 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 `terraform output`, 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 `terraform output <ИМЯ_ПЕРЕМЕННОЙ>`:

```
$ terraform output public_ip  
"54.174.13.5"
```

Это особенно полезно при написании скриптов. Например, вы можете создать скрипт развертывания, который развертывает веб-сервер с помощью команды `terraform apply`, получает его публичный IP-адрес из `terraform output public_ip` и обращается к этому адресу, используя `curl`. В итоге получится проверка по горячим следам, которая подтвердит успешность развертывания.

Входные и выходные переменные также являются неотъемлемыми ингредиентами 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 можно перенаправить к другому серверу, а размер самого кластера можно увеличивать и уменьшать в зависимости от трафика¹.

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 (Auto Scaling Group, 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.

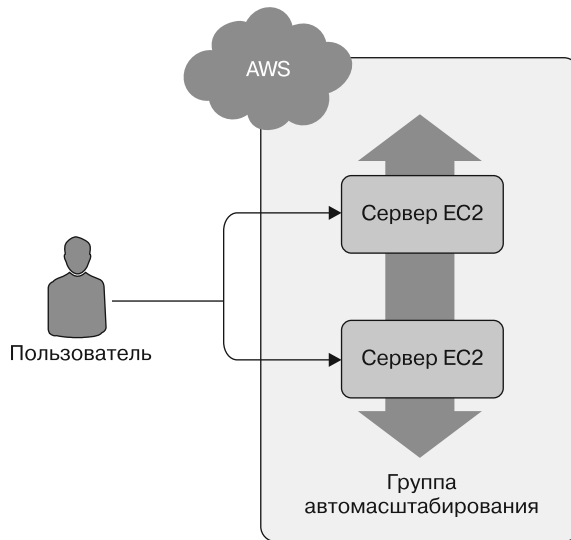


Рис. 2.9. Вместо одного веб-сервера группа автомасштабирования запускает кластер веб-серверов

¹ Более глубокий взгляд на построение высокодоступных и масштабируемых систем в AWS приводится в статье по адресу <https://bit.ly/2mpSXUZ>.

Первое, что нужно сделать при создании ASG, — написать *конфигурацию запуска*, которая определяет настройки каждого сервера EC2 в группе¹. 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`), но не поддерживает теги (позже мы будем работать с ними в ресурсе `aws_autoscaling_group`) и параметр `user_data_replace_on_change` (по умолчанию ASG всегда запускает новые экземпляры, поэтому этот параметр не нужен):

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0fb653ca2d3203ac1"
  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 можно 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 будет запускать от двух до десяти серверов EC2 (но сначала она запустит только два), каждый из которых имеет тег `terraform-asg-example`. Обратите внимание, что ASG ссылается на имя конфигурации запуска. Это приводит к проблеме: конфигурация запуска неизменяема, поэтому, если изменить любой ее параметр, Terraform попытается заменить ее целиком. Обычно при замене ресурса Terraform сначала удаляет его старую версию и затем

¹ На самом деле в наши дни лучше использовать шаблон запуска (и ресурс `aws_launch_template`) с ASG, а не конфигурацию запуска. Однако в примерах этой книги я остановился на конфигурации запуска, потому что на ее примере удобно изучать некоторые концепции, описанные в разделе «Развертывание с нулевым временем простоя» главы 5.

создает новую, но, поскольку ASG продолжает ссылаться на старый ресурс, Terraform не сможет его удалить.

Чтобы решить эту проблему, можно воспользоваться параметром *жизненного цикла*. Он поддерживается всеми ресурсами Terraform и определяет, как они создаются, обновляются и/или удаляются. 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, 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-0fb653ca2d3203ac1"
  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 конфигурации запуска
  # вместе с группой автомасштабирования.
  # 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 в примечании «Сетевая безопасность» выше в этой главе). Каждая подсеть находится в изолированной зоне доступности AWS (то есть в отдельном вычислительном центре), поэтому, развертывая серверы в разных подсетях, 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 сложно поддерживать и переносить. Вместо этого их лучше сохранить в учетной записи AWS, используя *источники данных*.

Источник данных — это фрагмент информации, доступный только 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, а просто получаете 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 источников данных очень похож на синтаксис ресурса:

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

PROVIDER — имя провайдера (например, `aws`), TYPE — тип источника данных (скажем, `vpc`), NAME — идентификатор, с помощью которого можно сослаться на этот источник данных в коде Terraform, а CONFIG содержит аргументы, поддерживаемые этим источником. Вот как с помощью источника данных `aws_vpc` можно запросить информацию о вашем облаке VPC по умолчанию (справочную информацию ищите во врезке «Замечание о виртуальных закрытых облаках по умолчанию» выше в этой главе):

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

Стоит отметить, что в аргументах источникам данных обычно передаются фильтры, которые указывают, какую информацию вы ищете. Источнику данных `aws_vpc` мы должны передать только один фильтр, `default = true`, который требует от Terraform отыскать VPC по умолчанию в вашей учетной записи AWS.

Для получения данных из источника используется следующий синтаксис доступа к атрибутам:

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

Например, вот как можно получить идентификатор VPC из источника данных `aws_vpc`:

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

Можно добавить еще один источник данных, `aws_subnet_ids`, чтобы найти подсети внутри этого облака VPC:

```
data "aws_subnet_ids" "default" {
  filter {
    name = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

А затем извлечь идентификаторы подсетей из источника `aws_subnet_ids` и передать их в аргументе с довольно странным именем `vpc_zone_identifier`, чтобы заставить группу ASG использовать 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 серверами, и сообщить всем вашим пользователям IP-адрес (или, если быть точным, доменное имя) балансировщика. 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).

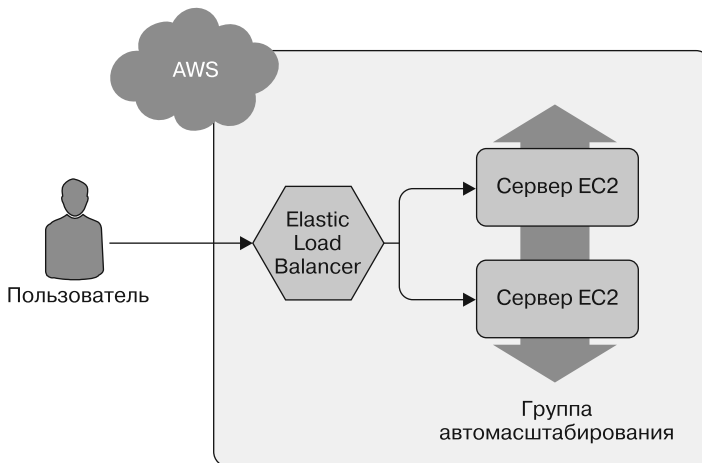


Рис. 2.10. Использование Amazon ELB для распределения трафика по группе автомасштабирования

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. В отличие от ALB, он быстрее масштабируется 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 (уровни 7 и 4) 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 работоспособным узлам.

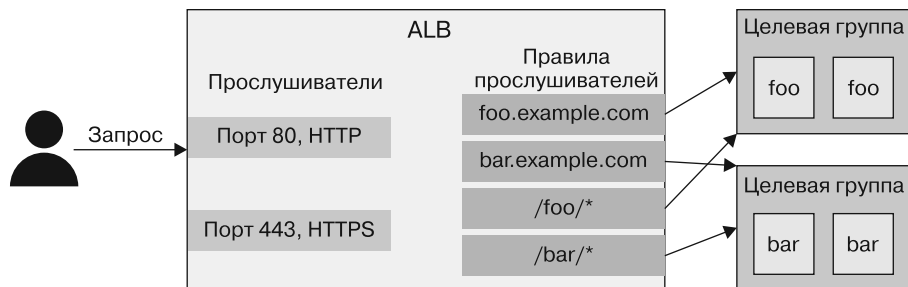


Рис. 2.11. Обзор Application Load Balancer (ALB)

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 облаке VPC по умолчанию с помощью источника данных `aws_subnet_ids`¹. Балансировщики нагрузки в AWS размещаются не на одном, а сразу на нескольких серверах, которые могут работать в разных подсетях (и, следовательно, в отдельных вычислительных центрах). AWS автоматически масштабирует количество балансировщиков в зависимости от трафика и обрабатывает отказ, если один из этих серверов выйдет из строя. Таким образом, вы получаете как масштабируемость, так и высокую доступность.

Следующий шаг — определение прослушивателя 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 прослушивателя.

¹ 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).

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 в 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 эту 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, что данная целевая группа будет проверять работоспособность ваших серверов, периодически отправляя им HTTP-запросы; сервер считается «работоспособным», только если его ответ совпадает с заданным значением `matcher` (например, вы можете указать в поле `matcher`, что ожидается ответ `200 OK`). 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 передать ей статический список серверов с помощью ресурса `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 лучше воспользоваться интеграцией между ASG и ALB. Вернитесь к ресурсу `aws_autoscaling_group` и определите аргумент `target_group_arns` так, чтобы он ссылался 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
    }
}
```

Также следует 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 определение выходной переменной `public_ip`, чтобы она возвращала не адрес сервера EC2, а доменное имя ALB:

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

Выполните `terraform apply` и почитайте полученный план. Как видите, согласно ему Terraform удалит первоначальный сервер EC2, а вместо него создаст конфигурацию запуска, ASG, ALB и группу безопасности. Если такой план вас устраивает, 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. Подождите, пока серверы загрузятся и ALB пометит их как работоспособные. А пока можете посмотреть, что вы развернули. Открыв раздел ASG консоли EC2 (<https://amzn.to/2MH3mId>), вы должны увидеть, что группа автомасштабирования уже создана (рис. 2.12).

Если перейти на вкладку **Instances** (Серверы), можно увидеть, что запущены два сервера EC2 (рис. 2.13).

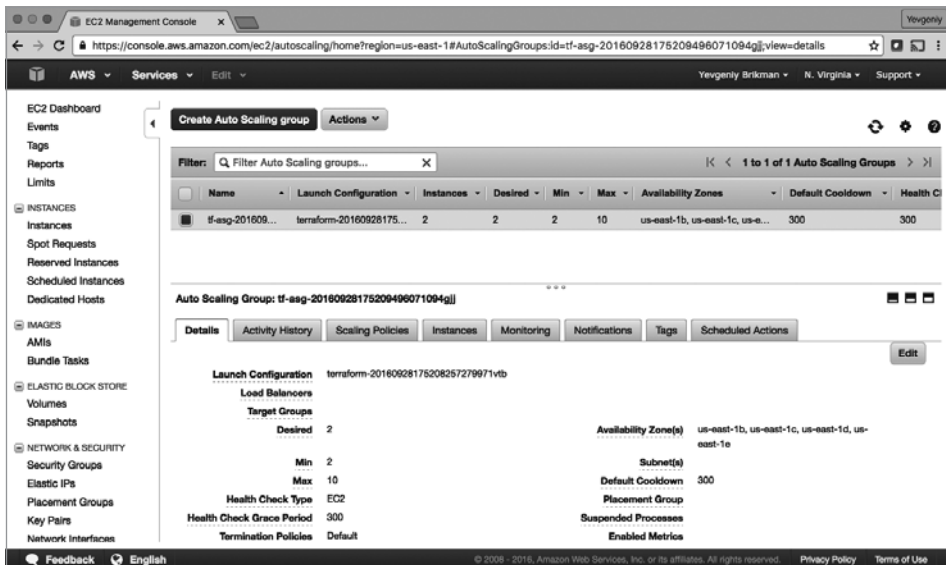


Рис. 2.12. Группа автомасштабирования

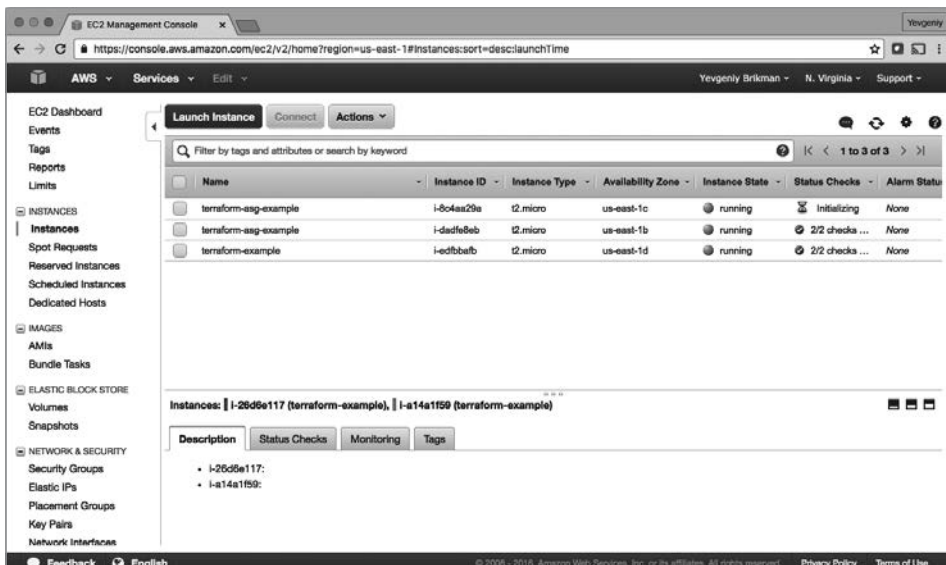


Рис. 2.13. Запуск серверов EC2 в ASG

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

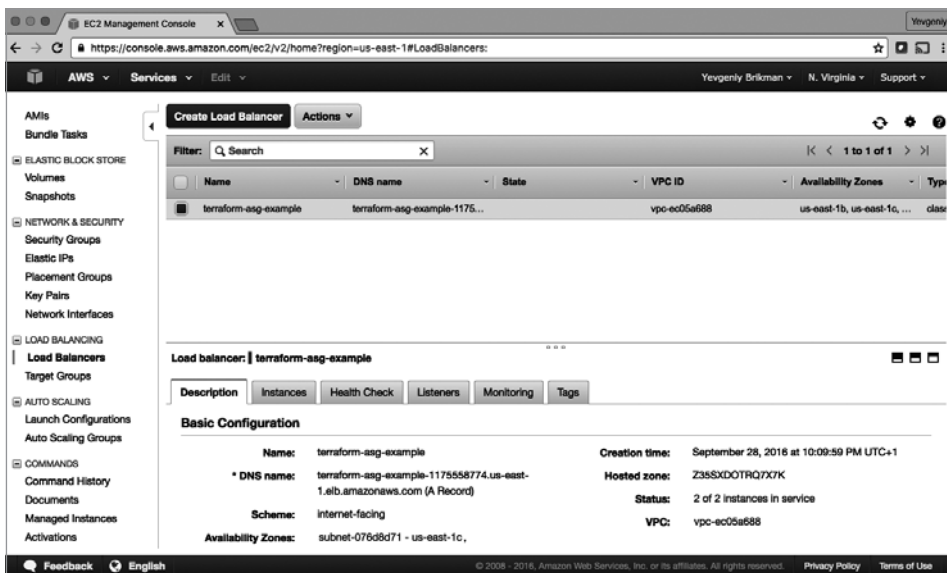


Рис. 2.14. Application Load Balancer

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



Рис. 2.15. Целевая группа

Выбрав свою целевую группу и щелкнув на вкладке **Targets** (Цели) в нижней части экрана, можно увидеть, что ваши серверы зарегистрировались в целевой группе и проходят проверки работоспособности. Подождите, пока их индикаторы состояния не начнут показывать **healthy**. Это обычно занимает 1–2 минуты. После проверьте вывод `alb_dns_name`, который вы скопировали ранее:

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

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

Теперь можно посмотреть, как кластер реагирует на создание новых и удаление старых серверов. Например, перейдите на вкладку **Instances** (Серверы) и удалите один из серверов: установите флажок, нажмите сверху кнопку **Actions** (Действия) и затем поменяйте состояние сервера на **Terminate** (Удалить). Если после этого снова попробовать обратиться к URL балансировщика ALB, то вы должны получить ответ **200 OK**, даже притом что сервер был удален. ALB автоматически обнаружит отсутствие сервера и перестанет передавать ему трафик. Что еще интересней, вскоре после удаления ASG поймет, что остался один сервер вместо двух, и автоматически запустит замену (самовосстановление!). Чтобы увидеть, как ASG меняет свой размер, добавьте в код Terraform параметр `desired_capacity` и снова выполните команды `apply`.

Удаление ненужных ресурсов

Закончив экспериментировать с Terraform в конце этой или любой из следующих глав, желательно удалить все созданные 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, стараясь распараллелить 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: **terraform apply**. На самом деле последние внесенные изменения лучше зафиксировать в Git, чтобы вы могли отслеживать историю своей инфраструктуры.

Резюме

У вас теперь есть общее представление 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. В главе 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 многократного применения.

Как управлять состоянием Terraform

В главе 2, используя Terraform для создания и обновления ресурсов, вы могли заметить, что при каждом выполнении команд `terraform plan` и `terraform apply` 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 размещение файлов, изоляцию и блокирование в проекте Terraform. Вот ключевые темы, по которым мы пройдемся.

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



Примеры кода

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

Что представляет собой состояние Terraform

При каждом 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, описанные в конфигурационных файлах, с их представлениями в реальном мире. К примеру, конфигурация Terraform определена так:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

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

```
{
  "version": 4,
  "terraform_version": "1.2.3",
  "serial": 1,
  "lineage": "86545604-7463-4aa5-e9e8-a2a221de98d2",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0fb653ca2d3203ac1",
            "availability_zone": "us-east-2b",
            "id": "i-0bc4bbe5b84387543",
            "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-0bc4bbe5b84387543` в вашей учетной записи AWS. При каждом запуске Terraform может запросить у AWS текущее состояние этого сервера и сравнить его с 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 читать или править 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, то можете 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 эти файлы должны храниться в общедоступном месте.
- *Блокирование файлов состояния.* Совместное использование одних и тех же данных сразу 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 тестовой среде или в среде обкатки 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 `terraform apply` двумя разными членами команды.
- *Наличие конфиденциальных данных*. Все данные в файлах состояния Terraform хранятся в виде обычного текста. Это чревато проблемами, поскольку в некоторых ресурсах Terraform могут храниться конфиденциальные данные. Например, если вы создаете базу данных с помощью ресурса `aws_db_instance`, Terraform сохранит имя пользователя и пароль в файле состояния в открытом виде. Открытое хранение конфиденциальных данных *где бы то ни было*, включая систему управления версиями, — плохая идея.

Вместо системы управления версиями для совместного управления файлами состояния лучше использовать удаленные хранилища, поддержка которых встроена в Terraform. *Хранилище* определяет, 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* разделяемые хранилища. Среди них можно выделить Amazon S3, Azure Storage, Google Cloud Storage и такие продукты, как Terraform Cloud 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`. Это исключает возможность человеческой ошибки.
- *Блокирование*. Большинство удаленных хранилищ имеют встроенную поддержку блокирования. При выполнении `terraform apply` Terraform автоматически устанавливает блокировку. Если в этот момент данную команду

выполнит кто-то другой, то она остановится на блокировке 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 10 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 корзиной Amazon S3), чтобы ограничить круг тех, 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 разворачивать и обслуживать дополнительную инфраструктуру.
- Рассчитан на 99,999999999 % устойчивости и 99,99 % доступности. Это означает, что вам не стоит сильно волноваться о потере данных и перебоях в работе¹.
- Поддерживает шифрование, что позволяет не беспокоиться о сохранности в тайне конфиденциальных данных, имеющихся в файлах состояния. Хотя это лишь частичное решение, так как любой член вашей команды, имеющий доступ к корзине S3, сможет просматривать эти файлы в открытом виде, но данные хотя бы будут шифроваться при сохранении (Amazon S3 поддерживает шифрование на стороне сервера с помощью AES-256) и передаче (Terraform использует SSL для чтения и записи данных в Amazon S3).
- Поддерживает блокирование с помощью DynamoDB (подробнее об этом чуть позже).
- Поддерживает *учет версий*, поэтому вы сможете хранить каждую ревизию своего состояния и в случае проблем откатываться к более старой версии.
- Недорогой, поэтому большинство сценариев применения Terraform легко вписываются в бесплатный тарифный план².

¹ Узнайте больше о гарантиях, которые дает S3, по адресу <https://amzn.to/31ihjAg>.

² Ознакомьтесь с тарифами для S3 по адресу <https://amzn.to/2yTtnw1>.

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

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

Затем создайте корзину S3, используя ресурс `aws_s3_bucket`:

```
resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-up-and-running-state"

  # Предотвращаем случайное удаление этой корзины S3
  lifecycle {
    prevent_destroy = true
  }
}
```

Этот код устанавливает следующие аргументы.

- `bucket`. Имя корзины S3. Имейте в виду, что имена корзин должны быть *глобально* уникальными среди всех клиентов AWS. Поэтому вместо `"terraform-up-and-running-state"` вы должны подобрать свое собственное имя (так как корзину с этим именем я уже создал). 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 `terraform destroy`) Terraform вернет ошибку. Это позволяет предотвратить случайное удаление важных ресурсов, таких как корзина S3 со всем вашим состоянием Terraform. Конечно, если вы действительно хотите ее удалить, просто закомментируйте этот параметр.

Теперь добавьте в эту корзину S3 несколько дополнительных уровней защиты.

Во-первых, используйте ресурс `aws_s3_bucket_versioning` для включения управления версиями в корзине S3, чтобы при каждой попытке обновить файл в корзине фактически создавалась его новая версия. Это позволит видеть более старые версии файла и возвращаться к ним в любое время, что может пригодиться, если что-то пойдет не так:

```
# Включить управление версиями, чтобы иметь возможность
# видеть всю историю изменения файлов состояния
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
}
```

```
versioning_configuration {  
  status = "Enabled"  
}  
}
```

Во-вторых, с помощью ресурса `aws_s3_bucket_server_side_encryption_configuration` включите шифрование на стороне сервера для всех данных, записываемых в эту корзину S3. Это обеспечит хранение файлов состояния и любых конфиденциальных данных в них в зашифрованном виде в корзине S3:

```
# Включить шифрование на стороне сервера  
resource "aws_s3_bucket_server_side_encryption_configuration" "default" {  
  bucket = aws_s3_bucket.terraform_state.id  
  
  rule {  
    apply_server_side_encryption_by_default {  
      sse_algorithm = "AES256"  
    }  
  }  
}
```

В-третьих, используйте ресурс `aws_s3_bucket_public_access_block`, чтобы заблокировать публичный доступ к корзине S3. Корзины S3 по умолчанию являются приватными, но, поскольку они часто используются для обслуживания статического контента (например, изображений, шрифтов, CSS, JS, HTML), их легко сделать общедоступными. Так как файлы состояния Terraform могут содержать конфиденциальные данные и секреты, стоит добавить этот дополнительный уровень защиты, чтобы гарантировать, что никто из вашей команды никогда не сможет случайно сделать эту корзину S3 общедоступной:

```
# Явно заблокировать публичный доступ к корзине S3  
resource "aws_s3_bucket_public_access_block" "public_access" {  
  bucket = aws_s3_bucket.terraform_state.id  
  block_public_acls = true  
  block_public_policy = true  
  ignore_public_acls = true  
  restrict_public_buckets = true  
}
```

Далее нужно создать таблицу DynamoDB, которая будет использоваться для блокировки. 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 план¹.

¹ Ознакомьтесь с тарифами для DynamoDB по адресу <https://amzn.to/2OJiyHr>.

Чтобы использовать DynamoDB для блокирования в связке с Terraform, нужно создать таблицу с первичным ключом с именем `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 `terraform init`, чтобы загрузить код провайдера, а затем `terraform apply`, чтобы развернуть ресурсы. После завершения развертывания вы получите корзину S3 и таблицу DynamoDB, но ваше состояние Terraform по-прежнему будет храниться локально. Чтобы хранить его в корзине S3 (с шифрованием и блокированием), нужно добавить в 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 используемого хранилища (например, `"s3"`), а `CONFIG` содержит один или несколько аргументов, предусмотренных специально для этого хранилища (скажем, имя корзины S3). Вот как выглядит конфигурация `backend` для корзины S3:

```
terraform {
  backend "s3" {
    # Укажите здесь имя своей корзины!
    bucket      = "terraform-up-and-running-state"
    key         = "global/s3/terraform.tfstate"
    region      = "us-east-2"

    # Укажите здесь имя своей таблицы DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Пройдемся по этим параметрам.

- **bucket.** Имя используемой корзины S3. Не забудьте присвоить ему имя своей корзины, созданной ранее.
- **key.** Путь к файлу внутри корзины S3, который будет использовать Terraform для сохранения файла состояния. Позже вы увидите, почему в предыдущем примере этому параметру присвоено значение `global/s3/terraform.tfstate`.
- **region.** Регион AWS, в котором находится корзина S3. Не забудьте указать регион той корзины, которую 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. Это дополнительная гарантия к настройке, включающей шифрование по умолчанию для S3.

Чтобы состояние Terraform сохранялось в этой корзине S3, нужно опять выполнить `terraform init`. 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 многократное 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 состояние Terraform будет сохранено в корзине S3. Чтобы в этом убедиться, откройте консоль управления S3 (<https://amzn.to/2Kw5qAc>) в браузере и выберите свою корзину. Вы должны увидеть нечто похожее на рис. 3.1.



Рис. 3.1. Файл состояния Terraform, хранящийся в S3

После включения этого хранилища Terraform будет автоматически загружать последнее состояние из корзины 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"
}
```

Эти переменные выведут на экран ARN (Amazon Resource Name) вашей корзины S3 и имя вашей таблицы DynamoDB. Чтобы в этом убедиться, выполните `terraform apply`:

```
$ 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, 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 корзине S3 (рис. 3.2).

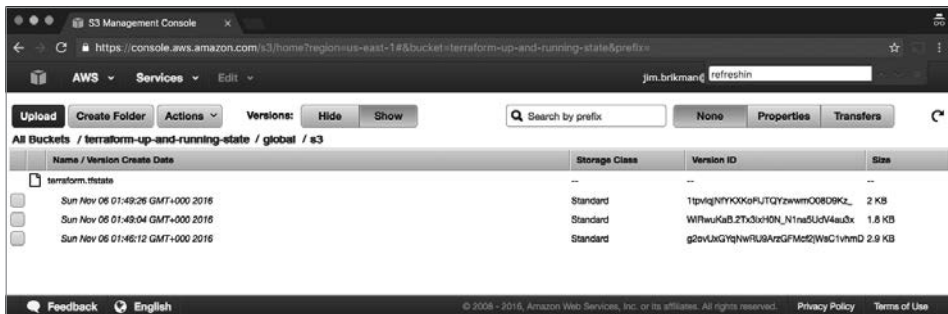


Рис. 3.2. Несколько версий файла состояния Terraform в S3

Это означает, что Terraform действительно сохраняет и загружает данные состояния в и из S3 и корзина хранит все версии файла состояния, что может пригодиться для отладки и отката к более старой версии, если что-то пойдет не так.

Ограничения хранилищ Terraform

Хранилища Terraform имеют несколько ограничений и подводных камней, о которых следует знать. Прежде всего, использование Terraform для создания корзины S3, в которой предполагается хранить состояние Terraform, похоже на ситуацию с курицей и яйцом. Чтобы добиться желаемого, вам пришлось выполнить следующее.

1. Написать код Terraform, чтобы создать корзину 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 задействовать свежесозданные корзину S3 и таблицу DynamoDB, и выполнить команду `terraform init`, чтобы скопировать локальное состояние в S3.

Если вдруг понадобится удалить корзину S3 и таблицу DynamoDB, вам придется выполнить обратные действия.

1. Перейти к коду Terraform, удалить конфигурацию `backend` и выполнить команду `terraform init`, чтобы скопировать состояние Terraform обратно на локальный диск.
2. Выполнить `terraform destroy`, чтобы удалить корзину S3 и таблицу DynamoDB.

Этот двухступенчатый процесс немного неуклюжий, зато позволяет использовать везде в вашем коде Terraform одни и те же корзину S3 и таблицу DynamoDB, к тому же его достаточно выполнить лишь раз (или по одному разу для каждой учетной записи AWS, если у вас их несколько). Если у вас уже есть корзина S3, то можете сразу указать 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
  }
}
```

Это означает, что вам придется вручную копировать и вставлять имя и регион корзины S3, а также имя таблицы DynamoDB в каждый ваш модуль Terraform. С модулями Terraform вы познакомитесь в главах 4 и 8, а пока достаточно знать, что модули — это способ организации и многократного использования кода Terraform, и что настоящий код Terraform обычно состоит из множества мелких модулей. Кроме того, вы должны 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 разворачивать и администрировать множество модулей Terraform во многих средах.

Одним из способов, помогающих решить проблему копирования/вставки кода, 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 `terraform init`. Например, вы можете вынести повторяющиеся аргументы *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 присваивать ему разные значения в разных модулях:

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

Чтобы собрать воедино все фрагменты вашей конфигурации, выполните команду `terraform init` 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. Вы можете использовать один и тот же файл `backend.hcl` со всеми своими модулями. Это решение значительно снижает дублирование кода; однако вам все равно придется в каждом модуле вручную присваивать параметру `key` уникальное значение ключа.

Еще одно решение — применение Terragrunt (<https://terragrunt.gruntwork.io/>), инструмента с открытым исходным кодом, который пытается компенсировать некоторые недостатки Terraform. Terragrunt может помочь избежать дублирования основных параметров хранилища (имя и регион корзины, имя таблицы DynamoDB) за счет определения их в едином файле и автоматического использования относительного пути к папке модуля в качестве значения `key`.

Пример с Terragrunt будет показан в главе 10.

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 развернуть новую версию своего приложения в среде обкатки вы можете нарушить его работу в промышленных условиях. Или еще хуже — 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 средах¹.

Весь смысл поддержки нескольких сред состоит в их изоляции друг от друга, поэтому, управляя всеми ими из одного набора конфигурационных файлов Terraform, вы нарушаете эту изоляцию. По аналогии с переборками на корабле, которые не дают утонуть судну из-за пробоины в одном отсеке, вы должны предусмотреть «переборки» для своей архитектуры Terraform (рис. 3.3).

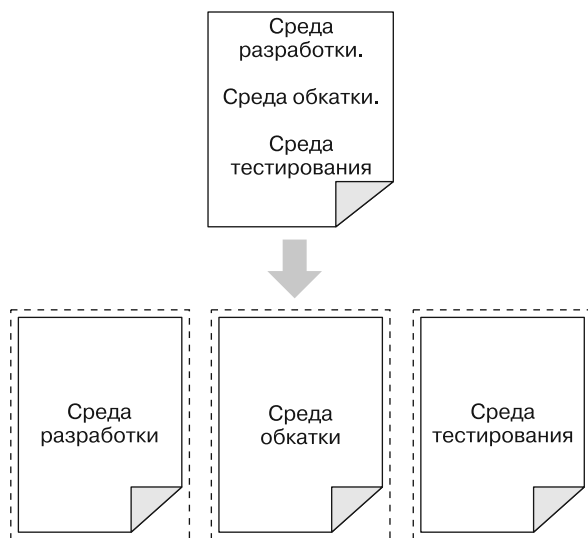


Рис. 3.3. Добавление «переборок» в архитектуру Terraform

Как показано на рис. 3.3, все среды должны описываться не в одном наборе конфигурационных файлов (*вверху*), а в разных наборах (*внизу*), поэтому появление проблемы в одной среде никак не отразится на 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 размещения файлов.* Подходит для промышленного использования, когда требуется строгая изоляция между средами.

¹ По адресу <https://bit.ly/2lTsewM> представлен яркий пример того, что может случиться, если не изолировать состояние Terraform.

Изоляция через рабочие области

Состояние Terraform можно хранить 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 `terraform workspace`. Поэкспериментируем с неким кодом Terraform, который развертывает сервер EC2:

```
resource "aws_instance" "example" {  
  ami      = "ami-0fb653ca2d3203ac1"  
  instance_type = "t2.micro"  
}
```

Сконфигурируйте для этого сервера хранилище на основе корзины 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" {  
    # Укажите здесь имя своей корзины!  
    bucket = "terraform-up-and-running-state"  
    key    = "workspaces-example/terraform.tfstate"  
    region = "us-east-2"  
  
    # Укажите здесь имя своей таблицы DynamoDB!  
    dynamodb_table = "terraform-up-and-running-locks"  
    encrypt         = true  
  }  
}
```

Выполните команды `terraform init` и `terraform apply`, 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 `terraform workspace show`, 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, заглянув в свою корзину S3, вы увидите файл `terraform.tfstate` и папку `workspaces-example`.

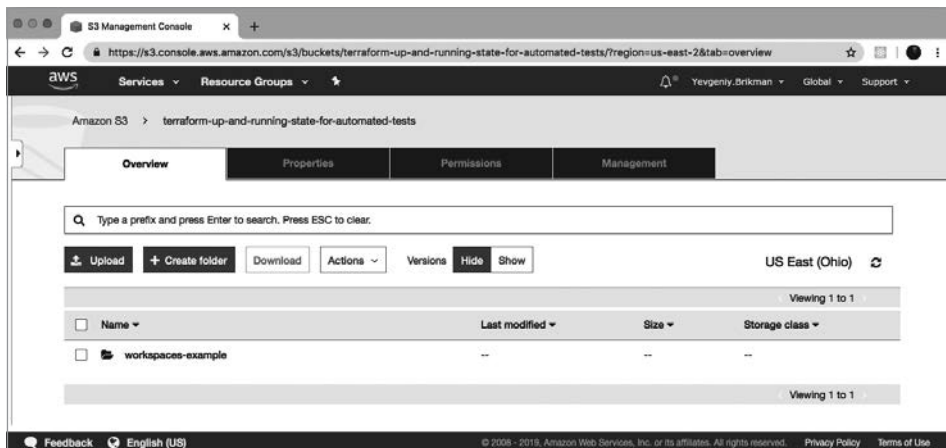


Рис. 3.4. Корзина S3 после сохранения состояния в рабочей области по умолчанию

Создадим новую рабочую область над названием `example1`, используя команду `terraform workspace new`:

```
$ 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 попытаться выполнить `terraform plan`:

```
$ terraform plan
```

Terraform will perform the following actions:

```
# aws_instance.example will be created
+ resource "aws_instance" "example" {
+ ami           = "ami-0fb653ca2d3203ac1"
+ instance_type = "t2.micro"
```

```
(...)  
}
```

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

Terraform хочет создать 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 `terraform apply`, 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 раз и создадим новую рабочую область с 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 `terraform apply`, 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 `terraform workspace list`:

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

Вы можете переключиться между ними в любой момент, используя команду `terraform workspace select`:

```
$ terraform workspace select example1  
Switched to workspace "example1".
```

Чтобы понять, как это работает внутри, еще раз загляните в свою корзину. Вы должны увидеть новую папку `env:`, как показано на рис. 3.5.

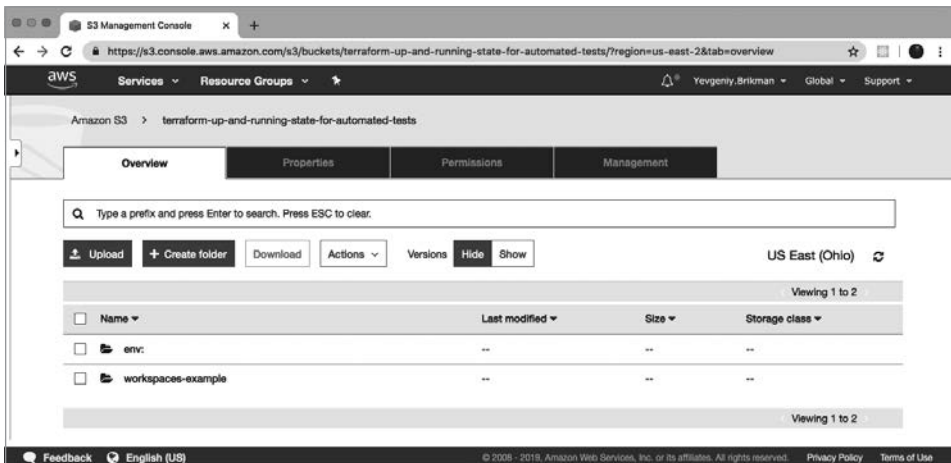


Рис. 3.5. Корзина S3 после того, как вы начали использовать собственные рабочие области

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

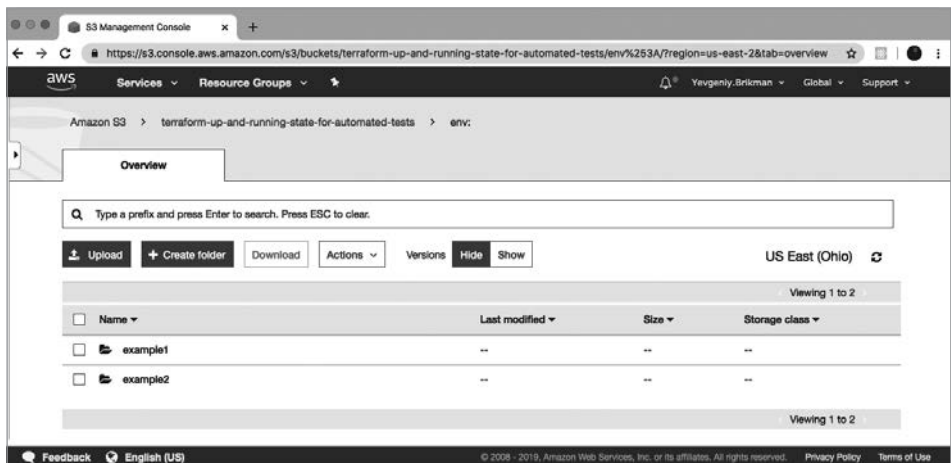


Рис. 3.6. Terraform создает по одной папке для каждой рабочей области

Внутри каждой из этих рабочих областей Terraform использует значение `key`, указанное в конфигурации хранилища, поэтому вы должны найти фай-

лы `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.

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 `terraform workspace new` 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 прочитать 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-0fb653ca2d3203ac1"
  instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"
}
```

Этот код использует *тернарный синтаксис*, чтобы присвоить параметру `instance_type`, `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 кода, но у них есть несколько недостатков.

- Файлы состояния всех рабочих областей находятся в одном и том же хранилище (например, в одной корзине S3). Это означает, что для доступа ко всем рабочим областям используются одни и те же механизмы аутентификации и управления доступом, и является одной из основных причин, почему такое решение не подходит для изоляции разных окружений, таких как среды для обкатки и промышленного применения.
- Рабочие области не видны в коде или терминале без использования команд `terraform workspace`. При чтении кода невозможно сказать, в скольких рабочих областях развернут модуль: в одной или десяти, так как выглядят они идентично. Это усложняет обслуживание из-за отсутствия полноценного представления об инфраструктуре.
- Из двух предыдущих пунктов следует, 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 `terraform`

`destroy` в рабочей области `production` вместо `staging`), а необходимость использовать один и тот же механизм аутентификации для всех рабочих областей лишает вас защиты от подобных ошибок.

Из-за этих недостатков рабочие пространства плохо подходят для изоляции одной среды от другой: например, для изоляции среды обкатки от промышленной¹. Чтобы как следует изолировать разные окружения, вместо рабочих областей лучше использовать прием с размещением файлов, о котором пойдет речь в следующем подразделе.

Но прежде, чем двигаться дальше, не забудьте удалить три сервера EC2, которые вы только что развернули. Для этого выполните `terraform workspace select <имя>` и `terraform destroy` в каждой из трех рабочих областей.

Изоляция с помощью размещения файлов

Чтобы достичь полной изоляции окружений, нужно сделать следующее.

- Поместить конфигурационные файлы Terraform для каждой среды в отдельную папку. Например, всю конфигурацию среды обкатки — в папку `stage`, а всю конфигурацию промышленной среды — в папку `prod`.
- Предусмотреть для каждой среды разные хранилища с разными механизмами аутентификации и управления доступом (например, у каждого окружения может быть отдельная учетная запись AWS со своей корзиной 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 окажет какое-либо влияние на другие.

Концепцию изоляции лучше опустить на уровень ниже, вплоть до компонентов — наборов связанных между собой ресурсов, которые обычно развертываются вместе. Например, после настройки базовой сетевой топологии для своей инфраструктуры (в терминологии AWS это виртуальное закрытое 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

¹ В документации по рабочим пространствам (<https://www.terraform.io/language/state/workspaces>) тоже говорится об этом, но эти слова спрятаны в нескольких абзацах текста, а поскольку рабочие пространства раньше назывались средами, я обнаружил, что многие пользователи до сих пор не понимают, когда не следует использовать рабочие пространства.

могут развертываться по несколько 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).

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 увидеть, как это выглядит на практике, разберем рекомендуемое размещение файлов в проектах Terraform.

На рис. 3.7 показана типичная структура каталогов для размещения файлов в моих проектах.

На самом верхнем уровне находятся отдельные папки для каждого окружения. Набор окружений зависит от проекта, но обычно включает:

- **stage** — среду для обкатки и тестирования ПО перед развертыванием в промышленном окружении;
- **prod** — среду для промышленной эксплуатации (приложения, взаимодействующие с пользователями);
- **mgmt** — среду для инструментов DevOps (например, узел-бастион, сервер непрерывной интеграции);
- **global** — папку с ресурсами, которые используются во всех средах (скажем, S3, IAM).

Внутри каждого окружения есть отдельные папки для каждого компонента. Набор компонентов зависит от проекта, но обычно включает:

- **vpc** — топологию сети для этой среды;
- **services** — приложения или микросервисы, которые запускаются в этой среде; например, пользовательский интерфейс на Ruby on Rails или серверные компоненты

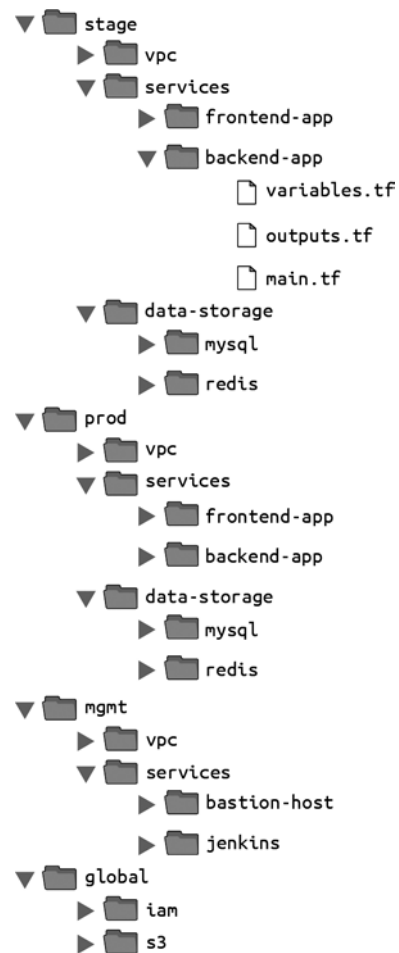


Рис. 3.7. Типичная структура каталогов для размещения файлов в проекте Terraform

на Scala. 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 со следующими именами:

- **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 давать своим файлам любые имена. Но имейте в виду, что вашим коллегам может быть не все равно, какие имена файлов вы используете. Если присваивать говорящие имена, это упростит просмотр кода. Вы всегда будете знать, где искать входные и выходные переменные или ресурсы.

Обратите внимание, что выше описано самое что ни на есть минимальное соглашение, которому вы должны следовать, потому что практически во всех случаях использования Terraform полезно иметь возможность быстро переходить к входным переменным, выходным переменным и ресурсам, но при желании вы можете выйти за рамки этого соглашения. Вот несколько примеров:

- **dependencies.tf** — обычно все источники данных помещаются в файл **dependency.tf**, чтобы облегчить выяснение внешних факторов, от которых зависит код;
- **providers.tf** — иногда полезно поместить блоки **provider** провайдеров в файл **providers.tf**, чтобы можно было сразу увидеть, с какими провайдерами взаимодействует код и какую аутентификацию нужно предоставить;
- **main-xxx.tf** — если файл **main.tf** становится слишком объемным из-за большого количества ресурсов, то его можно разбить на несколько мелких файлов, объединяющих ресурсы некоторым логическим образом: например, **main-iam.tf** может содержать все ресурсы IAM, **main-s3.tf** — все ресурсы S3 и т. д. Использование префикса **main-** упрощает просмотр списка файлов в папке, когда они организованы в алфавитном порядке, и определение, какой файл какие ресурсы содержит. Также стоит отметить, что если при попытке организовать множество ресурсов у вас получается большое количество файлов, то это может свидетельствовать о необходимости разбить код на более мелкие модули, о чем я и расскажу в главе 4.

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.

Корзину S3, созданную вами в этой главе, нужно 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 не забыли скопировать (скрытую) папку `.terraform`, иначе придется заново все инициализировать.

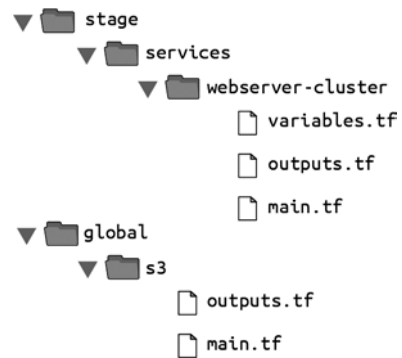


Рис. 3.8. Размещение файлов с конфигурацией кластера веб-серверов

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 `outputs.tf` соответственно.

Вам также необходимо обновить кластер веб-серверов, чтобы он использовал S3 в качестве хранилища. Для этого достаточно 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 размещение ваших файлов состояния будут полностью совпадать и связь между ними станет более очевидной. Модуль `s3` использует этот же принцип для установки параметра `key`.

Такое размещение файлов имеет ряд преимуществ.

- *Очевидность связей между кодом и окружениями* помогает понять, какие компоненты развернуты в той или иной среде.
- *Изоляция* — такая схема обеспечивает хорошую степень изоляции между окружениями и компонентами внутри одного окружения. Благодаря этому, если что-то пойдет не так, это коснется лишь небольшой части вашей инфраструктуры.

Однако эти преимущества являются также недостатками.

- *Разделение компонентов на отдельные папки* не позволит случайной оплошности нарушить работу всей инфраструктуры, но лишит вас возможности развертывать всю инфраструктуру одной командой. Если бы все компоненты

для одного окружения были описаны в одном конфигурационном файле Terraform, вы могли бы развернуть все окружение единственной командой `terraform apply`. Когда все компоненты находятся в отдельных папках, вам придется выполнить `terraform apply` в каждой из них.

Решение: с помощью Terragrunt можно запустить команды в нескольких папках одновременно, используя команду `run-all` (<https://terragrunt.gruntwork.io/docs/features/execute-terraform-commands-on-multiple-modules-at-once/>).

- *Копирование/вставка.* Размещение файлов, описанное в этом разделе, часто повторяется. Например, оба приложения, `frontend-app` и `backend-app`, будут присутствовать и в папке `stage`, и в папке `prod`.

Решение: на самом деле нет необходимости копировать и вставлять весь этот код. В главе 4 вы увидите, как использовать модули Terraform, чтобы избавиться от повторяющегося кода.

- *Зависимости ресурсов.* Разделение кода на несколько папок усложняет использование зависимостей ресурсов. Если код приложения определен в тех же файлах конфигурации Terraform, что и код базы данных, то в коде приложения вы сможете напрямую ссылаться на атрибуты базы данных (например, ссылкой на `aws_db_instance.foo.address` можно получить адрес базы данных). Но если код приложения и код базы данных находятся в разных папках, как я рекомендовал, то вы не сможете этого сделать.

Решение: один из вариантов — использовать блоки `dependency` в Terragrunt, как будет показано в главе 10. Другой вариант — использовать источник данных `terraform_remote_state`, как описано в следующем разделе.

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 в облаке 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 его доступным 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.

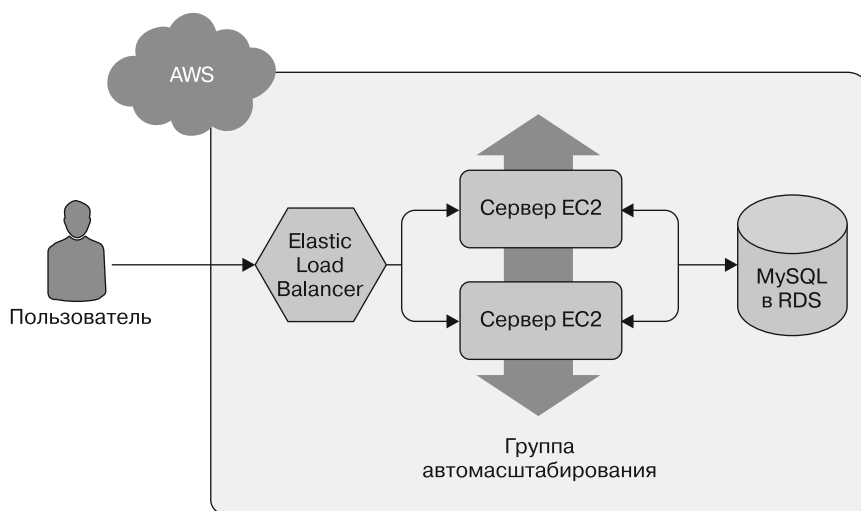


Рис. 3.9. Кластер веб-серверов взаимодействует с базой данных MySQL, развернутой поверх Amazon RDS

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 базой данных при каждом таком обновлении. Первое, что вы должны сделать, — 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.

Затем 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"
  skip_final_snapshot = true
  db_name          = "example_database"

  # Как нам задать имя пользователя и пароль?
  username = "???"
  password = "???"
}
```

Сразу после стандартного блока `provider` в верхней части файла находится новый ресурс: `aws_db_instance`. Он создает базу данных в RDS со следующими настройками:

- база данных на основе MySQL;
- объем хранилища 10 Гбайт;
- на сервере `db.t2.micro`, который имеет один виртуальный процессор, 1 Гбайт памяти и входит в бесплатный тариф AWS;
- создание окончательного снимка запрещено, так как этот код предназначен только для обучения и тестирования (если не отключить создание снимка и не указать имя для снимка в параметре `final_snapshot_identifier`, то вызов `destroy` завершится неудачей).

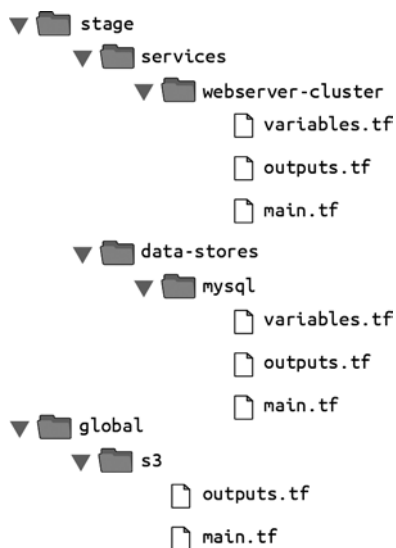


Рис. 3.10. Файлы для базы данных будут в папке `stage/data-stores`

Обратите внимание на два параметра, которые вы должны передать ресурсу `aws_db_instance` — имя главного пользователя и его пароль. Это конфиденциальная информация, поэтому ее нельзя прописывать прямо в коде в открытом виде! В главе 6 я расскажу о различных вариантах безопасной работы с конфиденциальной информацией. А пока воспользуемся вариантом, который позволяет избежать хранения каких-либо секретов в открытом виде и прост в применении: сохраните свои секреты, такие как пароли базы данных, за пределами Terraform (например, в менеджере паролей, таком как 1Password, LastPass или macOS Keychain) и передавайте их через переменные среды.

Для этого объявите переменные с именами `db_username` и `db_password` в `stage/datastores/mysql/variables.tf`:

```

variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}

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

```

Во-первых, обратите внимание, что эти переменные имеют атрибут `sensitive = true`, подсказывающий, что они содержат секреты. Это гарантирует, что Terraform не будет выводить значения переменных при выполнении команд `plan` и `apply`. Во-вторых, отметьте, что эти переменные не имеют параметра `default`. Это сделано намерено. Вы не должны хранить учетные данные для доступа к базе данных или любую конфиденциальную информацию в открытом виде. Вместо этого значение следует брать из переменных среды.

Напоминаю, что любую входную переменную, определенную в конфигурации Terraform (такую как `foo`), можно настроить так, что значение для нее будет извлекаться из переменной среды (такой как `TF_VAR_foo`). Для входных переменных `db_username` и `db_password` нужно установить переменные среды `TF_VAR_db_username` и `TF_VAR_db_password`. Вот как это делается в системах Linux/Unix/OS X:

```
$ export TF_VAR_db_username="(ИМЯ_ПОЛЬЗОВАТЕЛЯ_БД)"
$ export TF_VAR_db_password="(ПАРОЛЬ_К_БД)"
```

А так то же самое можно сделать в Windows:

```
$ set TF_VAR_db_username="(ИМЯ_ПОЛЬЗОВАТЕЛЯ_БД)"
$ set TF_VAR_db_password="(ПАРОЛЬ_К_БД)"
```

Следующий шаг после настройки учетных данных для доступа к базе данных — настройка модуля, чтобы он хранил свое состояние в корзине S3, созданной ранее в файле `stage/data-stores/mysql/terraform.tfstate`:

```
terraform {
  backend "s3" {
    # Укажите здесь имя своей корзины!
    bucket      = "terraform-up-and-running-state"
    key         = "stage/data-stores/mysql/terraform.tfstate"
    region      = "us-east-2"

    # Укажите здесь имя своей таблицы DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Наконец, добавьте две выходные переменные в `stage/data-stores/mysql/outputs.tf`, через которые будут возвращаться адрес и порт базы данных:

```
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 команды `terraform init` и `terraform apply`, чтобы создать базу данных. Учтите, что на выделение даже небольшой базы данных в Amazon RDS может уйти около 10 минут, поэтому будьте терпеливы. По завершении `apply` в терминале должны появиться такие строки:

```
$ terraform apply
```

```
(...)
```

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

```
Outputs:
```

```
address = "terraform-up-and-running.cowu6mts6srx.us-east-2.rds.amazonaws.com"
port = 3306
```

Выполните `terraform apply` еще раз, и в терминале должны появиться ваши исходящие 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
```

Эти выходные данные теперь также хранятся в состоянии Terraform для базы данных, которое находится в вашей корзине S3 в файле `stage/data-stores/mysql/terraform.tfstate`.

Чтобы код 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 будет читать файл состояния из тех же корзины S3 и папки, где хранится состояние базы данных (рис. 3.11).

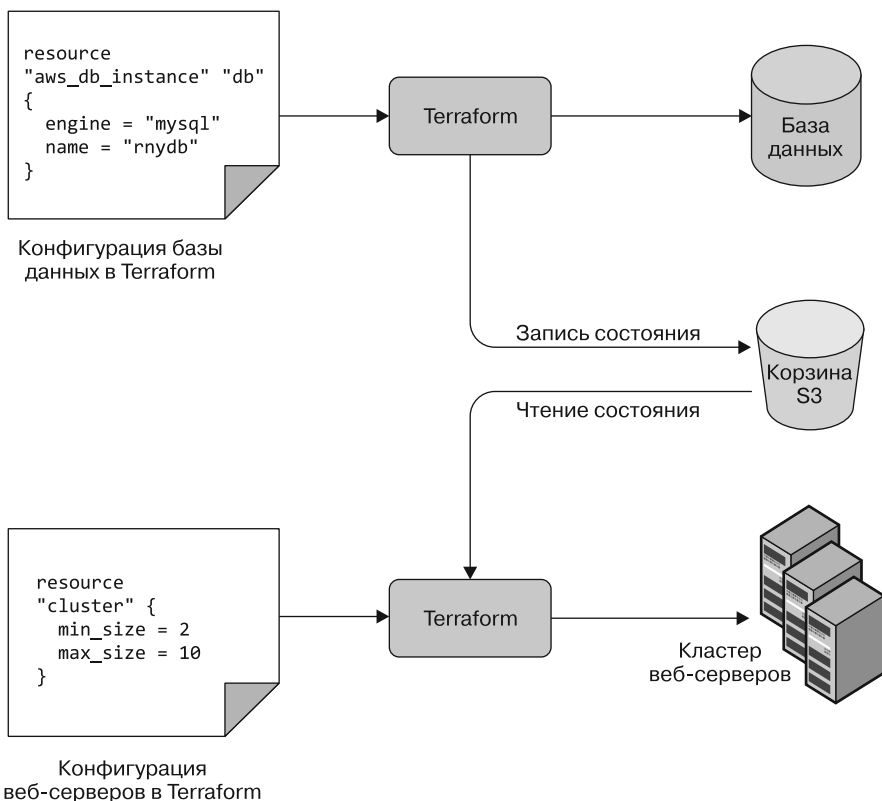


Рис. 3.11. База данных записывает свое состояние в корзину S3 (вверху), а кластер веб-серверов считывает его из той же корзины (внизу)

Важно понимать, что, как и все источники данных в Terraform, информация в `terraform_remote_state` доступна только для чтения. У вас нет никакой возможности поменять это состояние в коде кластера веб-серверов, поэтому тот факт, что вы извлекаете состояние базы данных, не представляет для нее никакого риска.

Все выходные переменные БД хранятся в файле состояния, и их значения можно прочесть 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 адрес и порт базы данных из источника `terraform_remote_state` 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 нерышливым выглядит 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 `templatefile`.

Terraform имеет множество *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 `FMT`¹. Чтобы поэкспериментировать со встроенными функциями, воспользуйтесь командой `terraform console`. Она создает интерактивную консоль, в которой можно попробовать вызывать функции Terraform, запрашивать состояние 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 массивами². К их числу относится функция `templatefile`:

```
templatefile(<PATH>, <VARS>)
```

¹ Описание синтаксиса `sprintf` можно найти на странице <https://golang.org/pkg/fmt/>.

² Полный список встроенных функций представлен на странице <https://www.terraform.io/language/functions>.

Эта функция читает файл в пути `PATH`, интерпретирует его содержимое и возвращает результат в виде строки. Под словами «интерпретирует его содержимое» я имею в виду, что файл, доступный в пути `PATH`, может использовать синтаксис интерполяции строк в Terraform (`${...}`), и Terraform будет отображать содержимое этого файла, подставляя значения переменных из `VARS` вместо ссылок на них.

Чтобы увидеть, как все это работает, сохраните следующий код в файле `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 получает значения переменных с помощью стандартного синтаксиса интерполяции Terraform. Единственными переменными в данном случае являются те, что передаются функции `templatefile` во втором параметре (как это сделать, я покажу), поэтому для доступа к ним не нужен никакой префикс: например, вместо `${var.server_port}` следует писать `${server_port}`.
- Теперь в скрипте можно заметить теги HTML (такие как `<h1>`), что делает вывод в браузере более удобочитаемым.

Заключительный шаг — обновление параметра `user_data` в ресурсе `aws_launch_configuration`. Добавьте в него вызов функции `templatefile` и передайте ей переменные в виде ассоциативного массива:

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  # Использовать интерпретируемый скрипт
  user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  })
}
```

```
# Требуется при использовании конфигурации запуска
# вместе с группой автомасштабирования
# 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 командой `terraform apply`, подождать, пока серверы зарегистрируются в ALB и открыть URL ALB в браузере, можно увидеть нечто похожее на рис. 3.12.

Ура! Теперь ваш кластер веб-серверов может программно получать адрес и порт 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 (такая как ActiveRecord).



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

Резюме

Причина, по которой 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 кода¹.

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

¹ Подробнее о защитных механизмах в ПО можно почитать по адресу <https://bit.ly/2YJuqJb>.

ГЛАВА 4

Многократное использование инфраструктуры с помощью модулей Terraform

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

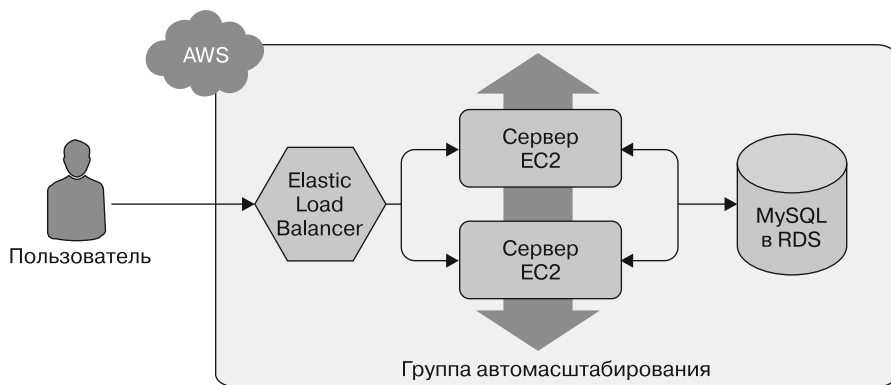


Рис. 4.1. Балансировщик нагрузки, кластер веб-серверов и база данных

Она хорошо подходит на роль первой среды, но их обычно нужно как минимум две: одна, тестовая, для обкатки внутри команды, а другая, промышленная, для обслуживания реальных пользователей (рис. 4.2). В идеале обе среды должны быть почти идентичными, хотя, чтобы сэкономить, для тестовой среды можно выделить чуть меньше серверов (или серверы меньшего размера).

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

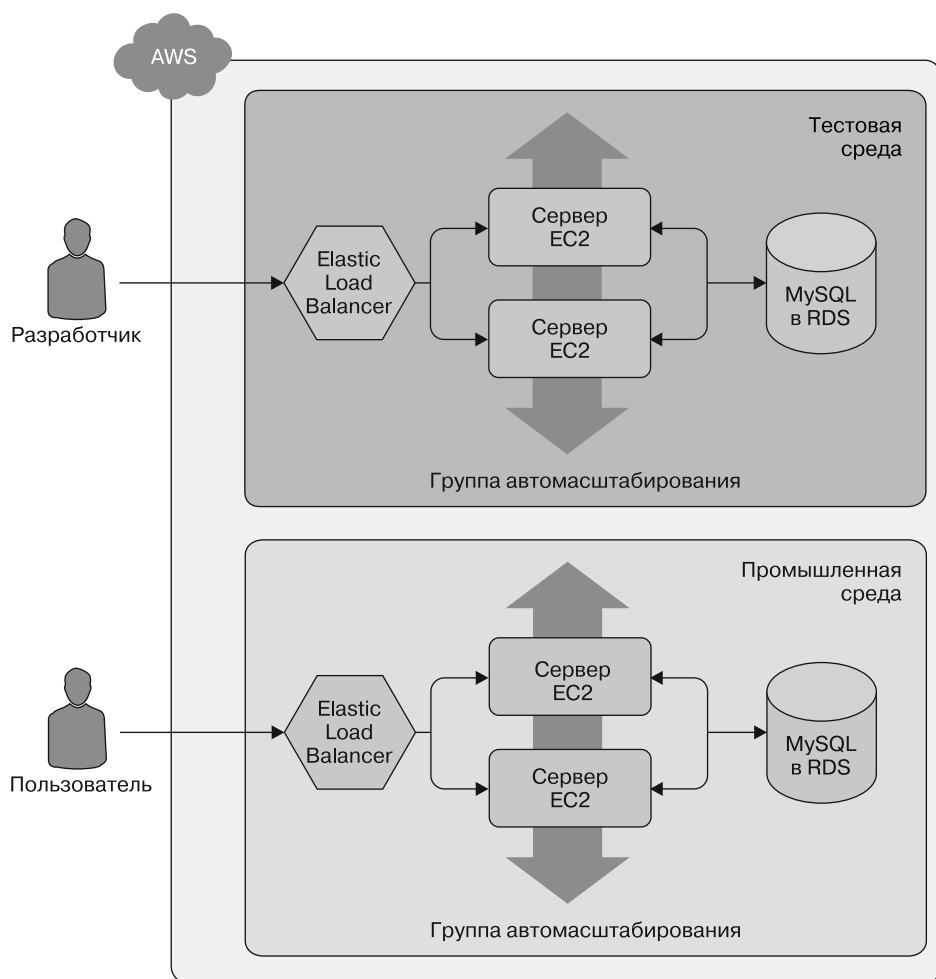


Рис. 4.2. Две среды, каждая со своим балансировщиком нагрузки, кластером веб-серверов и базой данных

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

```
# Определение функции в одном месте
def example_function()
  puts "Hello, World"
end

# Использование функции в других местах кода
example_function()
```

Terraform позволяет поместить код внутри *модуля*, который можно многократно применять в разных фрагментах конфигурации. Вместо копирования кода 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.

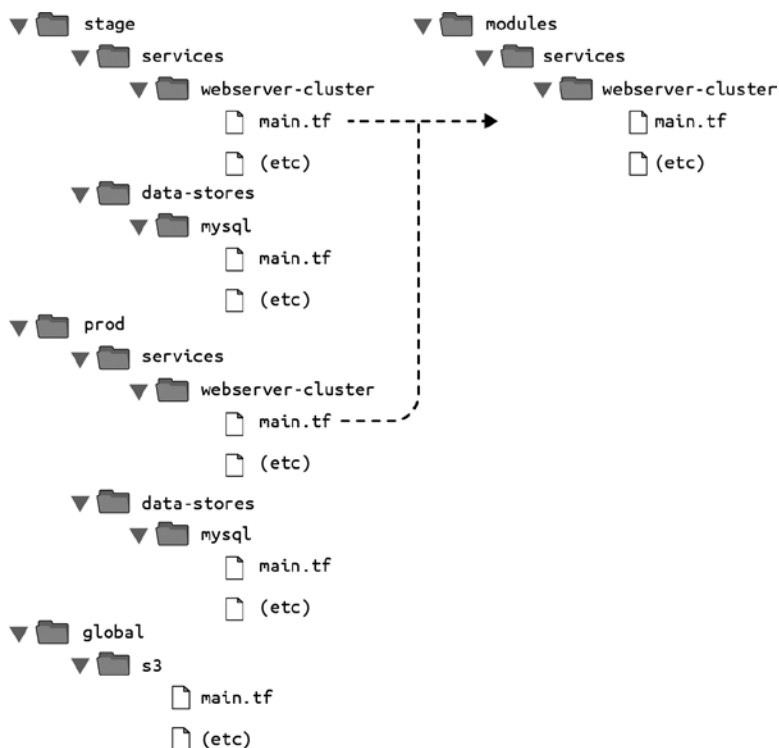


Рис. 4.3. Поместив код в модуль, вы сможете применить его повторно в разных окружениях

Это очень важно. Модули являются ключом к созданию 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;
- версионирование.



Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу <https://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 раз-вертывали 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 `terraform destroy` в `stage/services/webserver-cluster`, чтобы удалить все ресурсы, созданные вами ранее. Затем создайте новую папку верхнего уровня с именем `modules` и переместите в нее все файлы из `stage/services/webserver-cluster` в `modules/services/webserver-cluster`. В итоге ваша структура папок должна выглядеть как на рис. 4.4.

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

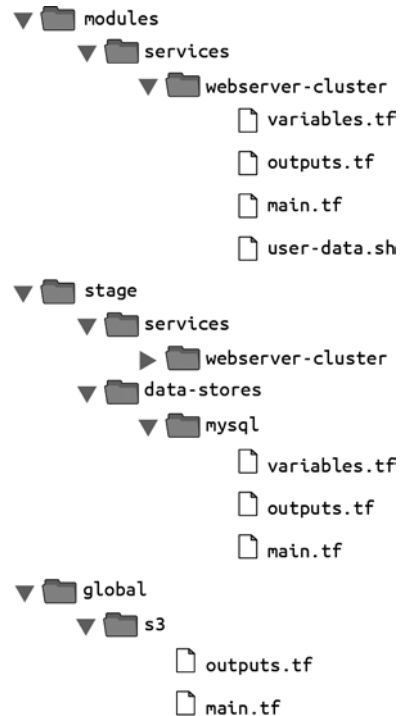


Рис. 4.4. Структура папок с модулем и тестовой средой

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 содержит аргументы для этого модуля. Например, вы можете создать новый файл `stage/services/webserver-cluster/main.tf` и использовать в нем модуль `webserver-cluster`:

```
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` со следующим содержимым:

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

module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"
}
```

Вот и все: многократное 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. Поэтому при попытке повторно использовать этот модуль 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 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, предусмотрев передачу ей входных параметров:

```
# Функция с двумя входными параметрами
def example_function(param1, param2)
  puts "Hello, #{param1} #{param2}"
end

# Передача двух параметров в вызов функции
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 новые входные переменные:

```
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` вместо прописанных вручную имен (скажем, `"terraform-asg-example"`). 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, как параметру `name` присваивается значение `"${var.cluster_name}-alb"`. Аналогичные изменения нужно внести в ресурс `aws_security_group` (например, присвоить ему имя `"${var.cluster_name}-instance"`), а также в `aws_alb` и в раздел `tag` ресурса `aws_autoscaling_group`.

Обновите также источник данных `terraform_remote_state`, используя переменные `db_remote_state_bucket` и `db_remote_state_key` для настройки параметров `bucket` и `key` соответственно, чтобы гарантировать чтение файла 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      = "webserver-stage"
  db_remote_state_bucket = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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      = "webserver-prod"
  db_remote_state_bucket = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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. Входные переменные — это API модуля и определяют его поведение 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, а в промышленных условиях — большой кластер, способный справиться с тяжелыми 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-0fb653ca2d3203ac1"
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]
```

```

user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address = data.terraform_remote_state.db.outputs.address
    db_port = data.terraform_remote_state.db.outputs.port
})

# Требуется при использовании конфигурации запуска
# вместе с группой автомасштабирования.
lifecycle {
    create_before_destroy = true
}
}

```

Обновите аналогично 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 = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
    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 мощные серверы (например, `m4.large`) с большим количеством процессоров и объемом

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 = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

Локальные переменные модуля

Определение параметров модуля с помощью входных переменных — отличное решение. А можно ли определить переменную внутри модуля для каких-то промежуточных вычислений или просто чтобы не дублировать код, но при этом не делать ее доступной для настройки на входе? 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 в нескольких местах в модуле. Явное и многократное задание этих «магических» значений усложняет чтение и поддержку кода. Их можно вынести во входные переменные, но в таком случае пользователи модуля смогут (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 их нельзя переопределить 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 настроить их для увеличения и уменьшения количества действующих серверов в зависимости от нагрузки. Для этого можно воспользоваться *запланированным действием*, которое будет менять размер кластера в заданное время суток. Например, если кластер испытывает повышенную нагрузку в рабочее время, то можно запланировать увеличение и уменьшение количества серверов в 9 утра и в 5 вечера соответственно.

Запланированное действие, определенное в модуле `webserver-cluster`, относится как к тестовой, так и к промышленной среде. Поскольку во время

тестирования такое масштабирование не нужно, можно 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 следующие 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 `"0 9 * * *"` означает «в 9 утра каждый день»), а второй уменьшает этот показатель на ночь (`"0 17 * * *"` 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:

```
# Функция, возвращающая значение
def example_function(param1, param2)
  return "Hello, #{param1} #{param2}"
end

# Вызов функции и получение возвращаемого значения
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 исходящим 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 нужно проверить после развертывания кластера. Для этого в файле `/modules/services/webserver-cluster/outputs.tf` добавьте еще одну выходную переменную:

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

После этого выходные переменные можно «передать» в 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 обращайтесь внимание на:

- пути к файлам;
- вложенные блоки.

Пути к файлам

В главе 3 вы поместили скрипт пользовательских данных для кластера веб-серверов во внешний файл, `user-data.sh`, и применили встроенную функцию `templatefile`, чтобы прочитать его с диска. Неочевидный момент функции `templatefile` — путь к файлу, 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 создает проблем, если функция `templatefile` вызывается в конфигурационном файле Terraform, который находится в той же папке, откуда запускается команда `terraform apply` (то есть если функция `templatefile` применяется в корневом модуле). Но если вызвать `templatefile` 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 вызове `templatefile` в файле `modules/services/webservercluster/main.tf` следует использовать `path.module`:

```
user_data = templatefile("${path.module}/user-data.sh", {
  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. *Вложенный блок* — это аргумент ресурса, определяемый в формате:

```
resource "xxx" "yyy" {  
  <NAME> {  
    [CONFIG...]  
  }  
}
```

NAME — это имя вложенного блока (например, `ingress`), а CONFIG состоит из одного или нескольких аргументов, принимаемых этим блоком (например, `from_port` и `to_port`). Например, правила для входящего и исходящего трафика в ресурсе `aws_security_group_resource` можно определить с помощью вложенных блоков (таких как `ingress {...}`) или отдельных ресурсов `aws_security_group_rule`.

При попытке использовать *оба* приема — вложенные блоки и отдельные ресурсы — вы получите ошибки при наличии конфликтов между конфигурациями. Поэтому использоваться должен только какой-то один подход. Мой совет: при создании модулей отдавайте предпочтение отдельным ресурсам.

Преимущество использования отдельных ресурсов в том, что их можно добавлять где угодно, тогда как вложенные блоки разрешено определять только внутри модуля, создающего ресурс. Поэтому использование отдельных ресурсов делает модули более гибкими и конфигурируемыми.

Например, в модуле `webserver-cluster` (`modules/services/webserver-cluster/main.tf`) вы использовали вложенные блоки для определения правил для входящего и исходящего трафика:

```
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  
  }  
}
```

Из-за использования вложенных блоков пользователь модуля не имеет возможности добавлять дополнительные правила извне. Чтобы сделать модуль более гибким, его следует изменить и определить правила для входящего и исходящего трафика 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
}
```

Вы должны также экспортировать идентификатор `aws_security_group` через выходную переменную в файле `groups/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"
}
```

Если теперь вам понадобится открыть дополнительный порт в тестовой среде для нужд тестирования, вы сможете это сделать, добавив ресурс `aws_security_group_rule` в файл `stage/services/webserver-cluster/main.tf`:

```
module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  # (параметры скрыты для удобства)
}
```

```
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 входящие или исходящие правила в виде вложенного блока, то 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 `terraform apply` 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 закрытое 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.

Версионирование модулей

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.

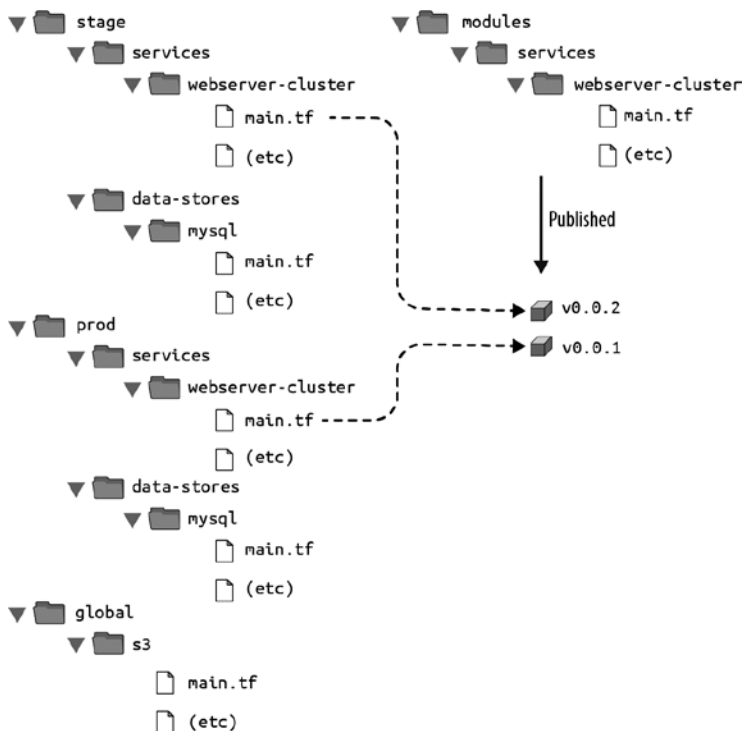


Рис. 4.5. Применение разных версий модуля в разных окружениях

Во всех примерах модулей, которые вы видели до сих пор, параметр `source` содержал локальный путь. Но, помимо путей к файлам, модули Terraform поддерживают и другие виды источников, такие как URL-адреса Git/Mercurial и произвольные HTTP URL¹. Самый простой способ версионирования моду-

¹ Все подробности о URL-источниках можно найти на странице <https://www.terraform.io/language/modules/sources>.

ля — размещение его кода в отдельном Git-репозитории, URL 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 содержит текущую инфраструктуру, развернутую в каждом окружении (`stage`, `prod`, `mgmt` и т. д.). Их можно представить как «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.

Чтобы организовать код таким образом, сначала нужно переместить папки `stage`, `prod` и `global` в папку `live`. Затем 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 (<https://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

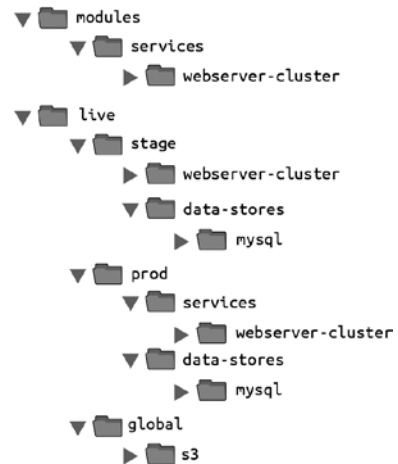


Рис. 4.6. Компоновка файлов с несколькими репозиториями

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 = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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 мало-понятными. 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*. Эта схема предполагает назначение версий в формате `MAJOR.MINOR.PATCH` (например, `1.0.4`) и определяет правила увеличения каждого элемента в номере версии. Вы должны увеличивать:

- элемент `MAJOR` при внесении несовместимых изменений в API;
- элемент `MINOR` при добавлении новых возможностей при сохранении обратной совместимости;
- элемент `PATCH` при исправлении ошибок при сохранении обратной совместимости.

Семантическое версионирование позволяет сообщить пользователям модуля, какого рода изменения вы внесли и как это сказывается на процессе обновления.

Поскольку вы обновили свою конфигурацию с использованием разных URL для разных версий вашего модуля, нужно заново выполнить команду `terraform init`, чтобы загрузить его код:

```
$ terraform init
Initializing modules...
Downloading git@github.com:brikis98/terraform-up-and-running-code.git?ref=v0.3.0
for webserver_cluster...
```

(...)

На этот раз вы можете видеть, что Terraform загружает код модуля из Git, а не из 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, то, чтобы использовать этот репозиторий в качестве источника, 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 SSH¹.

URL 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 выполнить клонирование 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

¹ Хорошее руководство по работе с SSH-ключами можно найти по адресу <https://bit.ly/2ZFLJwe>.

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 main
```

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 можно обновить URL, используемый в тестовой среде (`live/stage/services/webserver-cluster/main.tf`):

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/webserver-cluster?ref=v0.0.2"

  cluster_name      = "webservers-stage"
  db_remote_state_bucket = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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 = "github.com/foo/modules//services/webserver-cluster?ref=v0.0.1"

  cluster_name      = "webservers-prod"
  db_remote_state_bucket = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

Тщательно проверив версию `v0.0.2` в тестовой среде и убедившись в ее корректности, можно 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

Версионирование модулей отлично подходит, когда развертывание происходит в общем окружении (тестовом или промышленном), но если вы просто занимаетесь тестированием на собственном компьютере, лучше использовать локальные пути к файлам. 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 модулей будут использоваться локальные 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 откатываться 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 строк 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 понадобится 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 микросервисах без простоя? Этим углубленным аспектам синтаксиса Terraform посвящена глава 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 некоторые виды задач проще решаются в процедурном 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 позволяют 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 <https://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 и модулей.
- Выражение `for_each` для циклического перебора ресурсов, блоков, вложенных в ресурсы, и модулей.
- Выражение `for` для циклического перебора списков и ассоциативных массивов.
- Строковую директиву `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 метаметр `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 дать каждому пользователю уникальное 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" {
  + name      = "neo.0"
  (...)
}

# aws_iam_user.example[1] will be created
+ resource "aws_iam_user" "example" {
  + name      = "neo.1"
  (...)
}
```



```
# aws_iam_user.example[2] will be created
+ resource "aws_iam_user" "example" {
+   name      = "neo.2"
+   (...)
}
```

Plan: 3 to add, 0 to change, 0 to destroy.

Конечно, такое имя, как "neo.0", не очень полезно. Но если совместить `count.index` с некоторыми встроенными в Terraform функциями, каждую итерацию этого цикла можно изменить еще сильнее.

Например, все нужные 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 = var.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:

```
МАССИВ[<ИНДЕКС>]
```

Например, вот как взять из массива `var.user_names` элемент с индексом 1:

```
var.user_names[1]
```

- с *функцией length*. В Terraform есть встроенная функция `length`, которая имеет следующий синтаксис:

```
length(<МАССИВ>)
```

Как вы уже догадались, функция `length` возвращает количество элементов в заданном массиве. Она также работает со строками и ассоциативными массивами.

Если объединить все это вместе, 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" {
  + name          = "neo"
  (...)
}

# aws_iam_user.example[1] will be created
+ resource "aws_iam_user" "example" {
  + name          = "trinity"
  (...)
}

# aws_iam_user.example[2] will be created
+ resource "aws_iam_user" "example" {
  + name          = "morpheus"
  (...)
}
```

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 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, чтобы вернуть в выходной переменной ARN одного из пользователей IAM, нужно сделать следующее:

```
output "neo_arn" {
  value          = aws_iam_user.example[0].arn
  description = "The ARN for first user"
}
```

Чтобы получить ARN *всех* пользователей IAM, нужно указать символ `*` вместо индекса:

```
output "all_arns" {
  value      = aws_iam_user.example[*].arn
  description = "The ARNs for all users"
}
```

Если выполнить команду `apply`, выходная переменная `first_arn` будет содержать только ARN пользователя Neo, а `all_arns` — список всех ARN:

```
$ terraform apply
```

```
(...)
```

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

```
Outputs:
```

```
first_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",
]
```

Начиная с версии Terraform 0.13, параметр `count` также можно использовать с модулями. Например, представьте, что у вас есть модуль в `modules/landing-zone/iam-user`, который создает одного пользователя IAM:

```
resource "aws_iam_user" "example" {
  name = var.user_name
}
```

Имя пользователя передается в этот модуль через входную переменную:

```
variable "user_name" {
  description = "The user name to use"
  type        = string
}
```

И модуль возвращает ARN созданного пользователя IAM в выходной переменной:

```
output "user_arn" {
  value      = aws_iam_user.example.arn
  description = "The ARN of the created IAM user"
}
```

Такой модуль можно было бы использовать с параметром `count`, чтобы создать трех пользователей IAM:

```
module "users" {
  source = "../../modules/landing-zone/iam-user"
  count   = length(var.user_names)
  user_name = var.user_names[count.index]
}
```

В примере выше параметр `count` используется для перебора элементов списка имен пользователей:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

И вывести ARN созданных пользователей IAM:

```
output "user_arns" {
  value     = module.users[*].user_arn
  description = "The ARNs of the created IAM users"
}
```

Точно так же, как применение `count` к ресурсу превращает его в массив ресурсов, применение `count` к модулю превращает его в массив модулей.

Выполнив `apply` для этого кода, вы получите следующий результат:

```
$ terraform apply
```

```
(...)
```

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

```
Outputs:
```

```
all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
```

Как видите, `count` работает более или менее одинаково и с ресурсами, и с модулями.

К сожалению, у параметра `count` есть два ограничения, которые делают его куда менее полезным. Во-первых, с помощью `count` можно пройти по всему ресурсу, но при этом нельзя перебирать его вложенные блоки.

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 `terraform plan`?

```
$ 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. Но почему?

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 из середины 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 удаляется ресурс, находящийся внутри списка, Terraform удаляет все ресурсы, следующие за ним, и воссоздает их заново, с нуля. Ох. Конечно, итоговым результатом будет именно то, что вы просили (то есть два пользователя IAM с именами **"morpheus"** и **"neo"**), но вряд ли вам бы хотелось достичь этого за счет удаления и изменения ресурсов, так как при этом можно потерять доступ (вы не сможете использовать IAM пользователя во время выполнения **apply**) и, что еще хуже, потерять данные (если удаляемый ресурс является базой данных, вы рискуете потерять все данные в ней!)

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, либо модуля. Сначала рассмотрим первый вариант. Синтаксис выглядит так:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  for_each = <COLLECTION>

  [CONFIG ...]
}
```

`COLLECTION` — это 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 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 `first_arn` новой, `all_users`:

```
output "all_users" {
  value = aws_iam_user.example
}
```

Вот что произойдет, если выполнить `terraform apply`:

```
$ 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. Чтобы вернуть выходную переменную `all_arns`, нужно приложить дополнительные усилия и извлечь соответствующие ARN, применив встроенную функцию `values` (которая возвращает только значения ассоциативного массива) и добавив символ `*`:

```
output "all_arns" {
  value = values(aws_iam_user.example)[*].arn
}
```

Мы получим нужный нам результат:

```
$ 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",
]
```

Теперь, получив ассоциативный массив ресурсов с помощью `for_each` вместо обычного массива ресурсов, который возвращает `count`, вы можете безопасно удалять его элементы. Например, если опять удалить `"trinity"` из списка `var.user_names` и выполнить `terraform plan`, 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 нужный ресурс, не смещая остальных. Вот почему для создания множественных копий ресурса почти всегда следует выбирать `for_each` вместо `count`.

С модулями `for_each` работает почти так же. Используя ранее использованный модуль `iam-user`, с его помощью можно создать трех пользователей IAM, применив `for_each`:

```
module "users" {
  source = "../../modules/landing-zone/iam-user"

  for_each = toset(var.user_names)
  user_name = each.value
}
```

И вывести ARN этих пользователей:

```
output "user_arns" {
  value      = values(module.users)[*].user_arn
  description = "The ARNs of the created IAM users"
}
```

Запустив `apply` для этого кода, вы получите ожидаемый результат:

```
$ terraform apply
```

```
(...)
```

Apply complete! Resources: 3 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",
]
```

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 = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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"
    ManagedBy = "terraform"
  }
}
```

В этом листинге устанавливается несколько полезных тегов: `Owner` определяет, какой команде принадлежит ASG, а `ManagedBy` сигнализирует, что данная инфраструктура управляется с помощью Terraform (это говорит о том, что ее не следует редактировать вручную).

Итак, вы указали свои теги. Но как назначить их ресурсу `aws_autoscaling_group`? Для этого нужно циклически перебрать `var.custom_tags`, как показано 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 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 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 «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 `terraform apply`, 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          = "ManagedBy"
```

```

    + propagate_at_launch = true
    + value                 = "terraform"
  }
}

```

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



Соблюдение стандартов назначения тегов

Обычно хорошей идеей считается разработка стандарта выбора тегов для вашей команды и создание модулей Terraform, реализующих этот стандарт в виде кода. Конечно, можно вручную присвоить правильные теги всем ресурсам во всех модулях, но, когда ресурсов много, сделать это будет очень утомительно и легко допустить ошибки. Если есть теги, которые требуется присвоить всем ресурсам AWS, то надежнее добавить блок `default_tags` к провайдеру `aws` в каждом модуле:

```

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

  # Теги, добавляемые по умолчанию ко всем ресурсам AWS
  default_tags {
    tags = {
      Owner       = "team-foo"
      ManagedBy   = "Terraform"
    }
  }
}

```

Этот код гарантирует, что каждый ресурс AWS, созданный в этом модуле, будет включать теги `Owner` и `ManagedBy` (единственные исключения — ресурсы, не поддерживающие теги, и ресурс `aws_autoscaling_group`, который поддерживает теги, но не работает с `default_tags`, из-за чего, собственно, вам пришлось проделать всю работу, описанную в предыдущем разделе, чтобы присвоить теги в модуле `webserver-cluster`). `default_tags` гарантирует присваивание общего набора тегов всем ресурсам и в то же время позволяет перепределять эти теги для каждого отдельного ресурса. В главе 9 вы увидите, как определять и применять политики в виде кода, например «все ресурсы должны иметь тег `ManagedBy`», с помощью таких инструментов, как OPA.

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?

Представьте, что вы написали код Terraform, который принимает на входе 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

# Выведет: ['NEO', 'TRINITY', 'MORPHEUS']

```

Python позволяет выразить точно такой же код одной строкой, используя синтаксис под названием «генератор списков» (*list comprehension*):

```

names = ["neo", "trinity", "morpheus"]
upper_case_names = [name.upper() for name in names]
print upper_case_names

# Выведет: ['NEO', 'TRINITY', 'MORPHEUS']

```

Python также позволяет отфильтровать итоговый список по заданному условию:

```

names = ["neo", "trinity", "morpheus"]
short_upper_case_names = [name.upper() for name in names if len(name) < 5]
print short_upper_case_names

# Выведет: ['NEO']

```

Terraform предлагает похожие возможности в виде выражения `for` (не путать с выражением `for_each` из предыдущего раздела), имеющего 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:

```

output "upper_names" {
  value = [for name in var.names : upper(name)]
}

```

Если выполнить для этого кода команду `terraform apply`, она выведет:

```
$ terraform apply
```

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

```
Outputs:
```

```
upper_names = [
  "NEO",
  "TRINITY",
  "MORPHEUS",
]
```

По аналогии с генераторами списков в Python можно задать выражение для фильтрации полученного результата:

```
output "short_upper_names" {
  value = [for name in var.names : upper(name) if length(name) < 5]
}
```

Выполнив `terraform apply` 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 команду `terraform apply` для этого кода, она выведет:

```

bios = [
  "morpheus is the mentor",
  "neo is the hero",
  "trinity is the love interest",
]

```

Выражение `for` может вернуть ассоциативный массив вместо списка, если использовать такой 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) в каждой итерации выводится не только значение, но и ключ, отделенный от него стрелкой. Например, так можно перевести в верхний регистр все ключи и значения ассоциативного массива:

```

output "upper_roles" {
  value = {for name, role in var.hero_thousand_faces : upper(name) => upper(role)}
}

```

При выполнении этот код выведет:

```

upper_roles = {
  "MORPHEUS" = "MENTOR"
  "NEO" = "HERO"
  "TRINITY" = "LOVE INTEREST"
}

```

Циклы с использованием строковой директивы `for`

Ранее 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 (`%{...}`).

Terraform поддерживает два типа строковых директив: циклы `for` и условные выражения. В этом разделе мы рассмотрим циклы `for`; а условные выражения

рассмотрим далее в этой главе. 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 в 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 = "%{ for name in var.names }${name}, %{ endfor }"
}
```

Выполнив `terraform apply`, вы получите следующий вывод:

```
$ terraform apply
```

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

```
Outputs:
```

```
for_directive = "neo, trinity, morpheus, "
```

Существует также версия синтаксиса строковой директивы `for`, которая выводит индекс в цикле `for`:

```
%{ for <INDEX>, <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

Вот пример использования индекса:

```
output "for_directive_index" {
  value = "%{ for i, name in var.names }(${i}) ${name}, %{ endfor }"
}
```

Выполнив `terraform apply`, вы получите следующий вывод:

```
$ terraform apply
```

```
(...)
```

```
Outputs:
```

```
for_directive_index = "(0) neo, (1) trinity, (2) morpheus, "
```

Обратите внимание, что в обоих случаях выводятся дополнительные символы запятой и пробела. Такое поведение можно исправить, используя условное выражение, в частности строковую директиву `if`, как описывается в следующем разделе.

Условные выражения

Как и циклы, условные выражения в Terraform бывают нескольких видов, каждый из которых рассчитан на немного 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. Для начала в следующем разделе рассмотрим выражения `if`, 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 создает группу автомасштабирования (ASG), балансировщик нагрузки (ALB), группы безопасности и ряд других ресурсов. Чего он *не* создает, 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 их 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 проверить 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 выражений `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 присвоить `0`, этот ресурс вообще не будет создан.
- Terraform поддерживает *условные выражения* в формате `<CONDITION> ? <TRUE_VAL> : <FALSE_VAL>`. Это *тернарный синтаксис*, который может быть знаком вам по другим языкам программирования. Он проверяет булеву логику в `CONDITION` и, если результат равен `true`, возвращает `TRUE_VAL`; в противном случае возвращает `FALSE_VAL`.

Объединив эти две идеи, можно 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 вызов 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 = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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 вызов 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 с главы 4):

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"

  cluster_name      = "webservers-prod"
  db_remote_state_bucket = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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"
  ManagedBy = "terraform"
}
}
```

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 режим доступа — только для чтения или для чтения и записи — должен выбирать тот, кто применяет конфигурацию Terraform. Этот пример немного надуманный, но он позволяет продемонстрировать простую разновидность выражения if-else.

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 `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 записать так:

```
# 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 ветвь `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 ветвь `else`). Теперь вы знаете, как реализовать условную инструкцию `if-else`!

Теперь, получив возможность создавать те или иные ресурсы по условию `if/else`, хотелось бы узнать, как обратиться к какому-нибудь атрибуту созданного

ресурса? Например, представьте, что вы хотите добавить выходную переменную `neo_cloudwatch_policy_arn`, содержащую ARN фактически подключенной политики.

Самый простой вариант — использовать тернарный синтаксис:

```
output "neo_cloudwatch_policy_arn" {
  value = (
    var.give_neo_cloudwatch_full_access
    ? aws_iam_user_policy_attachment.neo_cloudwatch_full_access[0].policy_arn
    : aws_iam_user_policy_attachment.neo_cloudwatch_read_only[0].policy_arn
  )
}
```

В данный момент этот код будет работать нормально, но его легко сломать по неосторожности: если условие в параметре `count` ресурса `aws_iam_user_policy_attachment` изменится в будущем, например будет зависеть от нескольких переменных, а не только от `var.give_neo_cloudwatch_full_access`, то есть риск, что вы забудете обновить условие в этой выходной переменной и в результате получите трудноуловимую ошибку при попытке доступа к несуществующему элементу массива.

Более безопасное решение — воспользоваться функциями `concat` и `one`. Функция `concat` принимает два и более списка и объединяет их в один список. Функция `one` принимает список, и если в списке нет элементов, то возвращает `null`; если в списке один элемент, то возвращает этот элемент; а если в списке несколько элементов, то выдает ошибку. Объединив эти два значения с помощью функций `one` и `concat` и добавив знак звездочки, вы получите:

```
output "neo_cloudwatch_policy_arn" {
  value = one(concat(
    aws_iam_user_policy_attachment.neo_cloudwatch_full_access[*].policy_arn,
    aws_iam_user_policy_attachment.neo_cloudwatch_read_only[*].policy_arn
  ))
}
```

В зависимости от результата условия `if/else`, один из атрибутов, `neo_cloudwatch_full_access` или `neo_cloudwatch_read_only`, будет иметь пустое значение, а другой — содержать один элемент, поэтому, объединив их, вы получите список с одним элементом и функция `one` вернет этот элемент. Этот код сохранит работоспособность, как бы ни изменилось условие `if/else`.

Применение параметра `count` и встроенных функций для имитации выражения `if-else` сродни грязному трюку, но этот подход работает достаточно хорошо. Как можно 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, наверное, догадываетесь, что похожая стратегия возможно и при использовании выражения `for_each`.

Если передать `for_each` пустую коллекцию, получится ноль ресурсов, вложенных блоков или модулей. Но если коллекция непустая, будет создан один или несколько ресурсов, вложенных блоков или модулей. 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 и модулей, тогда как `for_each` лучше подходит для любых других видов циклов и условных выражений.

Условные выражения с использованием строковой директивы `if`

Давайте теперь рассмотрим строковую директиву `if`, которая имеет следующий синтаксис:

```
%{ if <CONDITION> }<TRUEVAL>%{ endif }
```

`CONDITION` — это любое выражение, возвращающее булево значение, а `TRUEVAL` — выражение, которое нужно вывести, если `CONDITION` равно `true`.

Выше в этой главе вы использовали строковую директиву `for` для циклического обхода содержимого строки и вывода нескольких имен, разделенных запятыми. Проблема заключалась в лишних символах запятой и пробела в конце строки. Решить эту проблему можно с помощью строковой директивы `if`:

```
output "for_directive_index_if" {
  value = <<EOF
%{ for i, name in var.names }
  ${name}%{ if i < length(var.names) - 1 }, %{ endif }
%{ endfor }
EOF
}
```

Здесь можно заметить несколько изменений по сравнению с исходной версией:

- я поместил код во *встроенный документ*, позволяющий определять многострочные строки, чтобы разбить код на несколько строк и сделать его более удобочитаемым;
- я использовал строковую директиву `if`, чтобы не выводить запятую и пробел после последнего элемента в списке.

Если выполнить команду `terraform apply`, она выведет следующее:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
for_directive_index_if = <<EOF
```

```
    neo,
```

```
    trinity,
```

```
    morpheus
```

```
EOF
```

Ой. Завершающая запятая исчезла, но добавилось несколько дополнительных пробельных символов (пробелов и перевода строки). Каждый пробельный символ, имеющийся во встроенном документе, попадает в финальную строку. Это можно исправить, добавив маркеры удаления пробельных символов (~) — они уберут лишние пробелы, предшествующие маркеру или следующие за ним:

```
output "for_directive_index_if_strip" {
  value = <<EOF
  %{~ for i, name in var.names ~}
  ${name}%{ if i < length(var.names) - 1 }, %{ endif }
  %{~ endfor ~}
  EOF
}
```

Попробуем применить эту версию:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
for_directive_index_if_strip = "neo, trinity, morpheus"
```

Уже лучше: никаких лишних пробелов и запятых. Однако вывод можно сделать еще красивее, добавив в строковую директиву ветку `else`, имеющую следующий синтаксис:

```
%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif }
```

`FALSEVAL` — это выражение, которое выводится, если `CONDITION` равно `false`. Вот пример, как добавить точку в конец выводимой строки:

```
output "for_directive_index_if_else_strip" {
  value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ else }.%{ endif }
%{~ endfor ~}
EOF
}
```

Теперь команда `terraform apply` выведет следующее:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
for_directive_index_if_else_strip = "neo, trinity, morpheus."
```

Развертывание с нулевым временем простоя

Итак, ваши модули имеют простой и понятный API для развертывания кластера веб-серверов. Теперь встает важный вопрос: как обновить этот кластер? То есть как развернуть в нем новый образ AMI (Amazon Machine Image) после внесения изменений в код? И как это сделать так, чтобы пользователи не ощутили перебоев в работе?

Для начала образ AMI нужно сделать доступным в виде входной переменной в файле `modules/services/webserver-cluster/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 в 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 в пользовательских данных возвращает из своего однострочного HTTP-сервера:

```
variable "ami" {
  description = "The AMI to run in the cluster"
  type        = string
  default     = "ami-0fb653ca2d3203ac1"
}

variable "server_text" {
  description = "The text the web server should return"
  type        = string
  default     = "Hello, World"
}
```

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 `image_id` значение переменной `var.ami` и обновите вызов `templatefile` в параметре `user_data`, передав ей `var.server_text`:

```
resource "aws_launch_configuration" "example" {
  image_id      = var.ami
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = templatefile("${path.module}/user-data.sh", {
    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
  })

  # Требуется при использовании конфигурации запуска вместе с группой.
  # 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`:

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"

  ami          = "ami-0fb653ca2d3203ac1"
  server_text  = "New server text"

  cluster_name      = "webservers-stage"
  db_remote_state_bucket = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  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 `server_text` теперь имеет 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              = "webserver-cluster-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-0fb653ca2d3203ac1"
    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 предполагает 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, можно уничтожить группу ASG (например, с помощью команды `terraform destroy`) и затем создать ее заново (скажем, выполнив `terraform apply`). 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 цикла¹.

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 уничтожить 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
  }
}
```

¹ Автором этой методики является Пол Хинзе (<https://bit.ly/2lksQgv>).

```

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-0fb653ca2d3203ac1"
  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 текст `forces replacement` напротив параметра `name` ресурса `aws_autoscaling_group`. Он 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. 5.1).

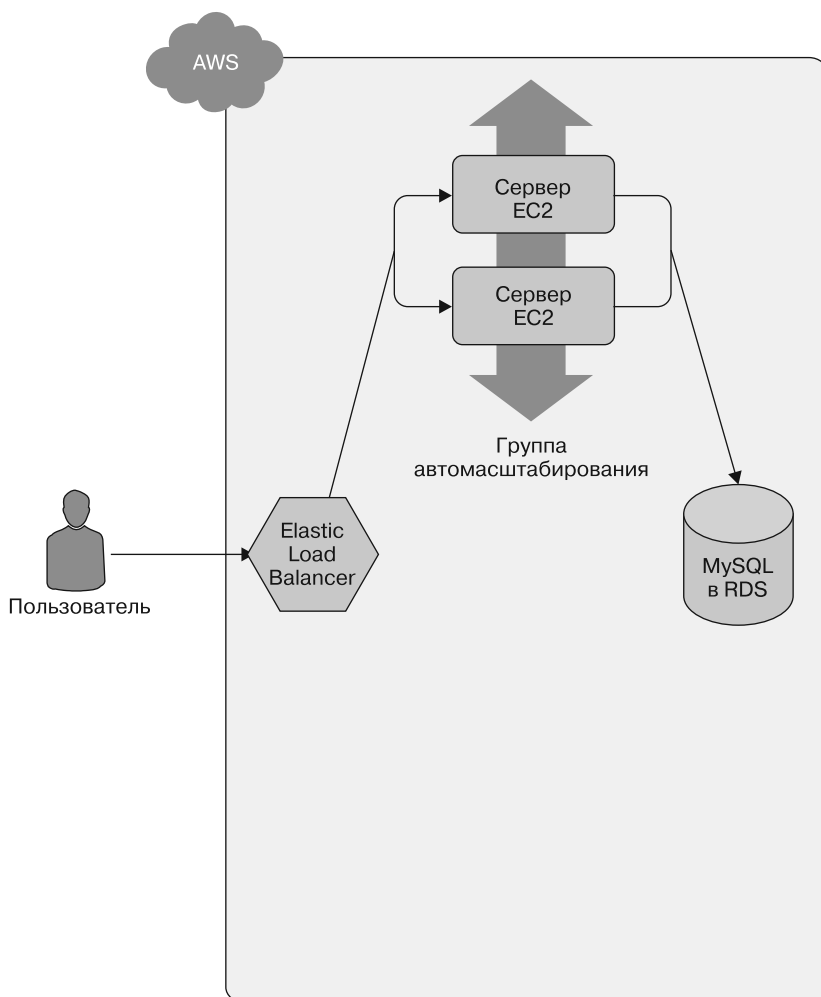


Рис. 5.1. Сначала имеется исходная группа ASG, выполняющая код версии v1

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. 5.2).

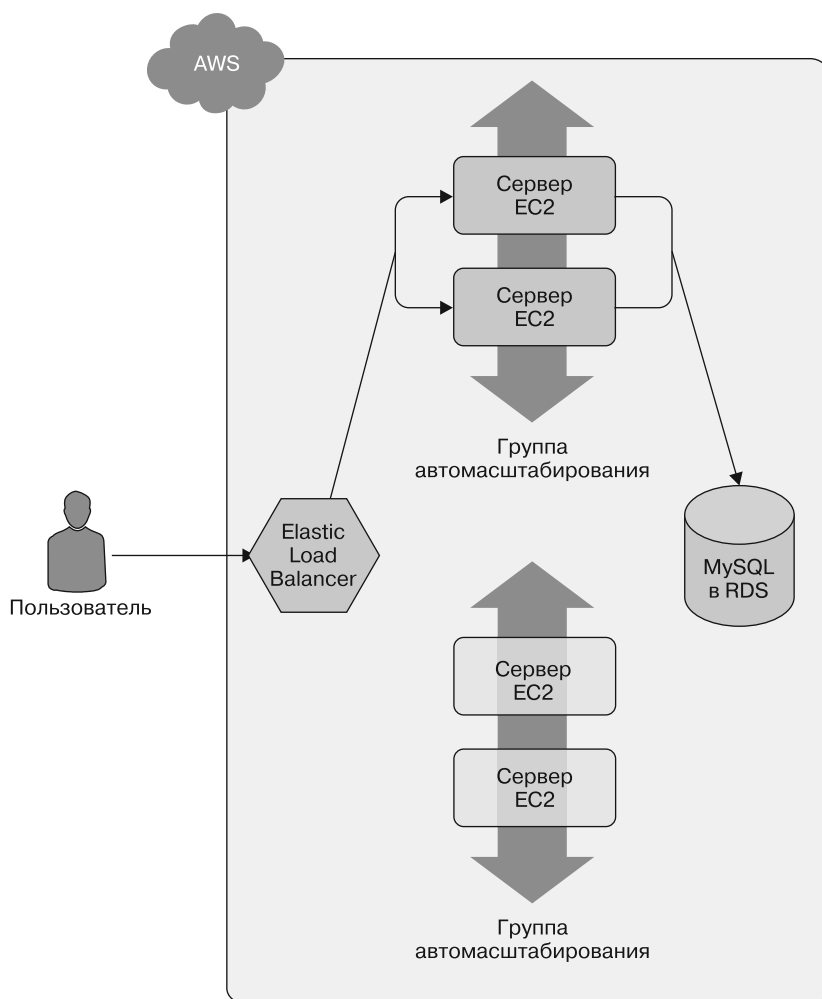


Рис. 5.2. Terraform начинает развертывание нового экземпляра ASG с кодом версии v2

После одной-двух 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. 5.3).

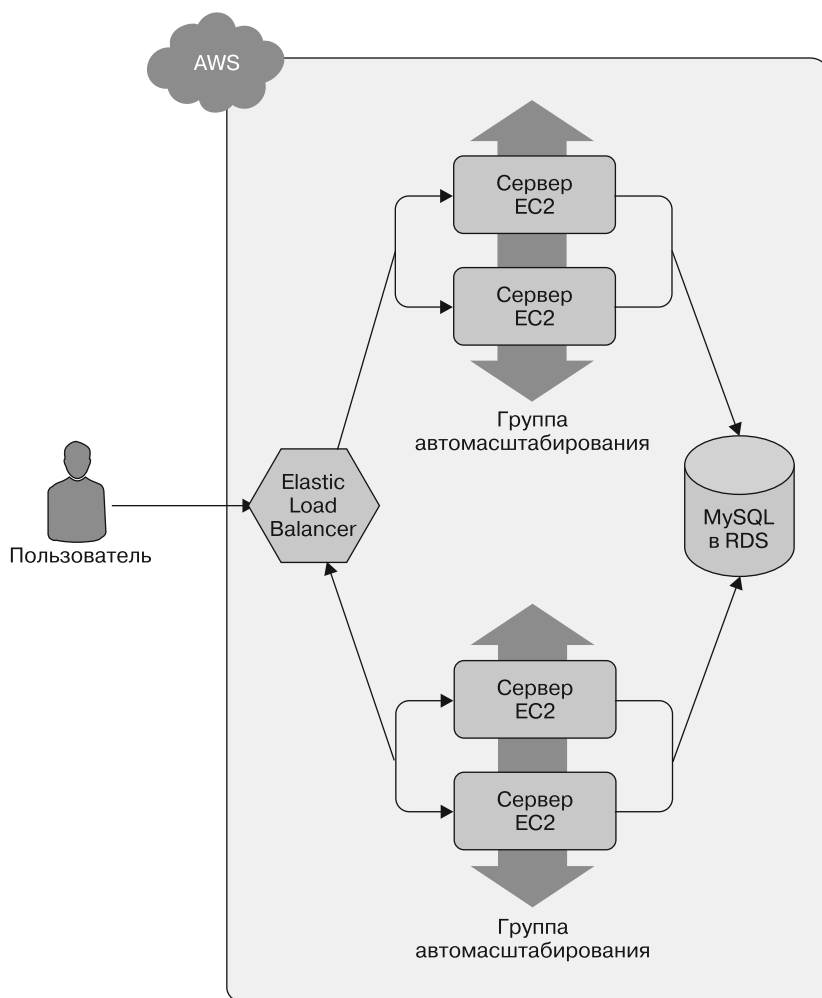


Рис. 5.3. Серверы в новой группе ASG загрузились, подключились к БД, зарегистрировались в ALB и начали обслуживать запросы

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 останавливаются (рис. 5.4).

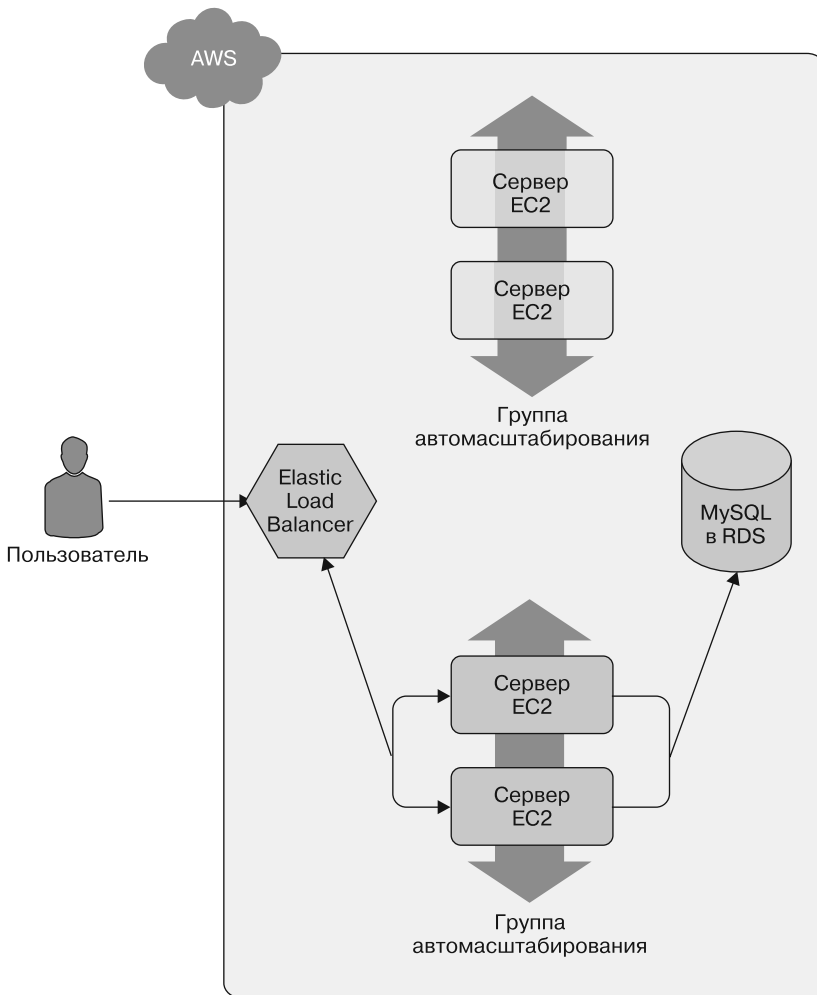


Рис. 5.4. Серверы из старой группы ASG начинают останавливаться

Через одну-две минуты старая группа ASG исчезнет, и у вас останется только версия v2 вашего приложения в новом экземпляре ASG (рис. 5.5).

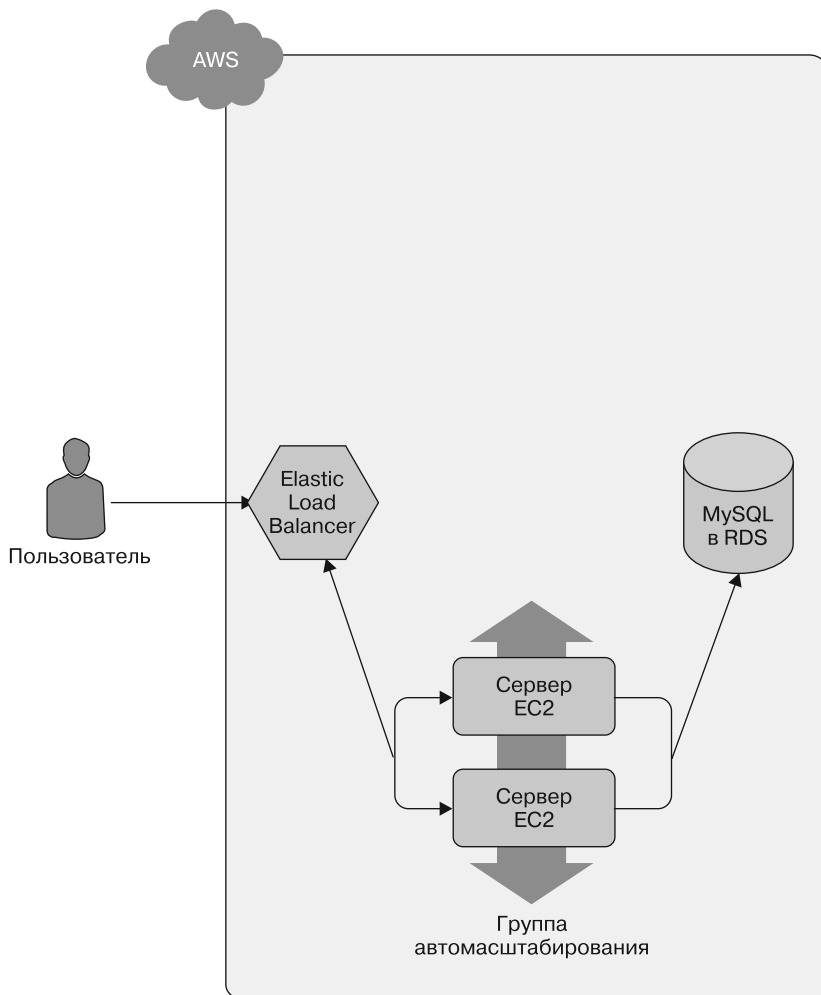


Рис. 5.5. Теперь остается только новая группа ASG, которая выполняет код версии v2

В течение всего процесса у вас будут оставаться действующие серверы, обслуживающие запросы от ALB, поэтому простоя не наблюдается. Открыв URL-адрес ALB в своем браузере, вы должны увидеть что-то похожее на рис. 5.6.

Получилось! Сервер с новым сообщением развернут. В качестве увлекательного эксперимента внесите еще одно изменение в параметр `server_text` (например,

поменяйте текст на `foo bar`) 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://<url_балансировщика_нагрузки>; sleep 1; done
```



Рис. 5.6. Новый код уже развернут

Где-то на протяжении первой минуты ответ должен оставаться прежним: `New server text`. Затем вы заметите чередование `New server text` и `foo bar`. Значит, новые серверы зарегистрировались в ALB и прошли проверку работоспособности. Еще через минуту сообщение `New server text` исчезнет, и вы будете видеть только `foo bar`. 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 приложения обнаружится ошибка, из-за которой оно не сможет 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 10 минут). После этого посчитает развертывание неудавшимся, удалит серверы v2 ASG и завершит работу с ошибкой (тем временем версия v1 приложения продолжит 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.

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 показали, но у них есть важное ограничение: 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-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

Поскольку параметру `count` присвоено статическое значение, этот код заработает без проблем и, когда вы выполните команду `apply`, он создаст три сервера EC2. Но что, если вам понадобится развернуть по одному серверу

в каждой зоне доступности (Availability Zone, AZ) в текущем регионе AWS? Вы можете обновить свой код, чтобы он загрузил список зон из источника данных `aws_availability_zones`, 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-0fb653ca2d3203ac1"
  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 присвоить значение выходной переменной `result` этого ресурса параметру `count` ресурса `aws_instance`:

```
resource "aws_instance" "example_3" {
  count          = random_integer.num_instances.result
  ami            = "ami-0fb653ca2d3203ac1"
  instance_type  = "t2.micro"
}
```

Если выполнить для этого кода `terraform plan`, 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.

Ограничения развертываний с нулевым временем простоя

Использование `create_before_destroy` с ASG для развертывания с нулевым временем простоя имеет несколько ограничений.

Первая проблема заключается в том, что этот прием не работает с политиками автоматического масштабирования. Или, если быть более точным, он сбрасывает размер ASG обратно в `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, такими как настройка параметра `recurrence` в `aws_autoscaling_schedule` или установка параметра `desired_capacity` группы ASG, чтобы получить его значение из пользовательского сценария, который использует AWS API, чтобы выяснить, сколько экземпляров было запущено перед развертыванием.

Вторая и более серьезная проблема заключается в том, что для важных и сложных задач, таких как развертывание с нулевым временем простоя, желательно использовать готовые надежные решения, а не обходные пути, требующие бессистемного склеивания `create_before_destroy`, `min_elb_capacity`, пользовательских скриптов и т. д. Как оказывается, в настоящее время AWS предлагает готовое решение для групп автомасштабирования, называемое *обновлением сервера* (*instance refresh*).

Вернитесь к ресурсу `aws_autoscaling_group` и отмените изменения, которые вы внесли, реализуя развертывание с нулевым временем простоя:

- присвойте параметру `name` прежнее значение `var.cluster_name`, чтобы он зависел от имени `aws_launch_configuration`;
- удалите настройки `create_before_destroy` и `min_elb_capacity`.

А теперь обновите ресурс `aws_autoscaling_group`, задействовав в нем блок `instance_refresh`, как показано ниже:

```
resource "aws_autoscaling_group" "example" {
  name                  = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier   = data.aws_subnets.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

  # Использовать instance_refresh для накатывания изменений на ASG
  instance_refresh {
    strategy = "Rolling"
    preferences {
      min_healthy_percentage = 50
    }
  }
}
```

Если вы развернете эту группу ASG, а затем измените какой-либо параметр (например, `server_text`) и запустите `plan`, то Terraform предложит простое обновление `aws_launch_configuration`:

Terraform will perform the following actions:

```
# module.webserver_cluster.aws_autoscaling_group.ex will be updated in-place
~ resource "aws_autoscaling_group" "example" {
  id                  = "webserver-cluster-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-0fb653ca2d3203ac1"
    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.

Если теперь запустить команду `apply`, она завершится очень быстро и в первый момент не будет развернуто ничего нового. Однако, поскольку вы изменили конфигурацию запуска, AWS запустит в фоновом режиме процесс обновления сервера, как показано на рис. 5.7.

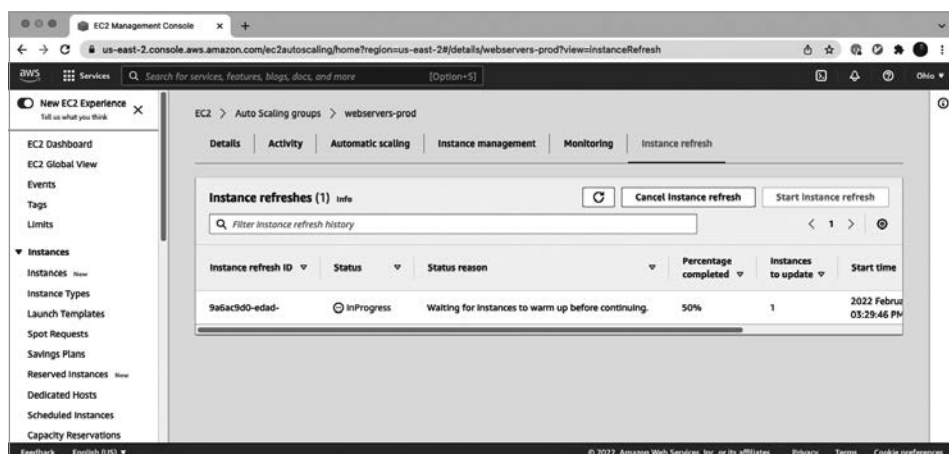


Рис. 5.7. Процесс обновления сервера продолжается

AWS сначала запустит один новый сервер, подождет, пока он пройдет проверку работоспособности, остановит один из старых серверов, а затем повторит процесс со вторым сервером и завершит процесс обновления, как показано на рис. 5.8.

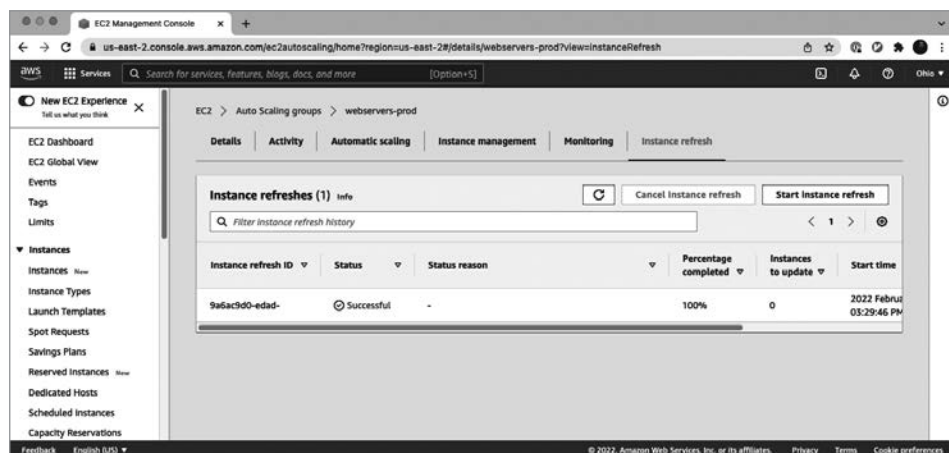


Рис. 5.8. Процесс обновления сервера завершился

Этот процесс полностью управляется AWS, он может настраиваться в широких пределах, довольно хорошо обрабатывает ошибки и не требует никаких обходных путей. Единственный недостаток: иногда процесс может протекать медленно (замена двух серверов, к примеру, занимает до 20 минут), но в остальном это

гораздо более надежное решение для большинства развертываний с нулевым временем простоя.

В общем случае предпочтение желательно отдавать готовым решениям развертывания, таким как обновление сервера, если это возможно. Такие решения не всегда были доступны на заре Terraform, но в наши дни они поддерживаются многими ресурсами. Например, если вы используете Amazon EC2 Container Service (ECS) для развертывания контейнеров Docker, то вам будет приятно узнать, что ресурс `aws_ecs_service` имеет встроенную поддержку развертывания с нулевым временем простоя посредством параметров `deployment_maximum_percent` и `deployment_minimum_healthy_percent`; если вы используете Kubernetes для развертывания контейнеров Docker, то вам пригодится ресурс `kubernetes_deployment` со встроенной поддержкой развертывания с нулевым временем простоя, которая активизируется присваиванием параметру `strategy` значения `RollingUpdate` и передачей конфигурации через блок `rolling_update`. Загляните в документацию по ресурсам, которые вы используете, и по возможности используйте встроенные решения!

Даже хороший план может оказаться неудачным

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 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.

Дело все в том, что команда `terraform plan` учитывает только ресурсы, указанные в файле состояния Terraform. Если ресурсы созданы *каким-то другим способом* (например, вручную, щелкая кнопкой мыши в 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 `terraform import`. 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 AWS API, найдет вашего пользователя IAM и создаст в файле состояния связь между ним и ресурсом `aws_iam_user.existing_user` в 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 такие инструменты, как `terraformer` (<https://github.com/GoogleCloudPlatform/terraformer>) и `terracognita` (<https://github.com/cycloidio/terracognita>), способные автоматически импортировать код и состояние из облачных окружений.

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, иначе возникнут непредвиденные проблемы.

Например, распространенный вид рефакторинга — замена имен 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, в модуле `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 изменить параметр `name` в каком-то ресурсе, Terraform удалит старую версию этого ресурса и создаст 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 один вид рефакторинга, который вас может заинтересовать, — изменение идентификатора Terraform. Возьмем в качестве примера ресурс `aws_security_group` в модуле `webserver-cluster`:

```
resource "aws_security_group" "instance" {
  # (...)
}
```

Этот ресурс имеет идентификатор `instance`. Представьте, что во время рефакторинга вы решили заменить его более понятным (по вашему мнению) именем `cluster_instance`:

```
resource "aws_security_group" "cluster_instance" {
  # (...)
}
```

Что в итоге получится? 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 новую, а в промежутке между этими двумя дей-

ствиями ваши серверы начнут отклонять любой сетевой трафик. С аналогичными проблемами можно столкнуться, если изменить идентификатор, связанный с модулем, разделите один модуль на несколько модулей или добавьте `count` или `for_each` к ресурсу или модулю, у которого раньше их не было.

Вот четыре основных урока, которые вы должны извлечь из этого обсуждения.

- *Всегда используйте команду `plan`.* Она может помочь выявить все эти загвоздки. Тщательно просматривайте ее вывод и обращайтесь 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 выполнить `apply` еще раз.
- *Рефакторинг может повлечь изменение состояния.* Если вы решили реорганизовать свой код, не вызвав случайных перебоев, то вам необходимо соответствующим образом обновить состояние Terraform. Однако никогда не следует обновлять файлы состояния Terraform вручную! В таких случаях у вас есть на выбор два варианта: выполнить команду `terraform state mv` или обеспечить автоматическое обновление состояния, добавив в код блок `moved`.

Сначала рассмотрим прием с командой `terraform state mv`, 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` следует 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 `terraform plan` не покажет никаких изменений, значит, вы все сделали правильно.

Необходимость помнить о запуске команд вручную может повлечь ошибки, особенно если вы выполнили рефакторинг модуля, используемого десятками команд в вашей компании, и каждая из этих команд должна не забыть выполнить `terraform state mv`, чтобы избежать простоев. К счастью, в Terraform 1.1 появилось автоматическое решение этой проблемы: блоки `moved`. Каждый раз, выполняя рефакторинг кода, добавляйте блок `moved`, чтобы указать, как должно обновляться состояние. Например, чтобы указать, что ресурс `aws_security_group` меняет имя с `instance` на `cluster_instance`, добавьте следующий блок `moved`:

```
moved {
  from = aws_security_group.instance
  to   = aws_security_group.cluster_instance
}
```

Теперь, когда кто-то запустит `apply` для этого кода, Terraform автоматически определит, что он должен обновить файл состояния:

Terraform will perform the following actions:

```
# aws_security_group.instance has moved to
# aws_security_group.cluster_instance
resource "aws_security_group" "cluster_instance" {
  name      = "moved-example-security-group"
  tags      = {}
  # (8 unchanged attributes hidden)
}
```

Plan: 0 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:

Если ввести `yes`, то Terraform обновит состояние автоматически, а поскольку в плане не указано никаких ресурсов для добавления, изменения или уничтожения, Terraform не будет вносить никаких других изменений — а это именно то, что вам нужно!

- *Некоторые параметры нельзя изменять.* Параметры многих ресурсов неизменяемые. Если попытаться их изменить, Terraform удалит старый ресурс и создаст 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`.

Резюме

Несмотря на свою декларативную сущность, Terraform включает целый ряд конструкций, делающих 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 имитации условных выражений, поэтому потратьте некоторое время на чтение документации по функциям на сайте <https://www.terraform.io/docs/configuration/functions.html> и позвольте разыгаться своему «внутреннему хакеру». Главное, не переусердствуйте, так как кто-то все равно должен поддерживать ваш код; просто постарайтесь научиться создавать опрятные и изящные API для своих модулей.

Теперь перейдем к главе 6, где я расскажу, как создавать не только чистые и красивые модули, но и модули, способные надежно и безопасно обрабатывать конфиденциальные данные.

ГЛАВА 6

Управление конфиденциальными данными

В какой-то момент вам и вашему программному обеспечению будут переданы различные секреты, такие как пароли баз данных, ключи API, сертификаты TLS, ключи SSH, ключи GPG и т. д. Это все конфиденциальные данные, попадание которых в чужие руки может нанести большой ущерб вашей компании и ее клиентам. Создавая программное обеспечение, вы обязаны хранить такие данные в секрете.

Например, рассмотрим следующий код Terraform, развертывающий базу данных:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # Как сохранить в секрете значения этих параметров?
  username = "???"
  password = "???"
}
```

Этот код требует присвоить конфиденциальные значения двум параметрам: `username` и `password`, которые являются учетными данными главного пользователя базы данных. Если они попадут в руки злоумышленника, это может иметь катастрофические последствия, потому что эти учетные данные дают неограниченный доступ к базе данных. Итак, как же сохранить эти данные в секрете?

Это часть более широкой темы *управления секретами*, которой посвящена эта глава. Вот темы, которые будут здесь затронуты:

- основы управления секретами;
- инструменты управления секретами;
- использование инструментов управления секретными данными в комплексе с Terraform.

Основы управления секретами

Первое правило управления секретами:

Не храните секреты в открытом месте.

Второе правило управления секретами:

НЕ ХРАНИТЕ СЕКРЕТЫ В ОТКРЫТОМ МЕСТЕ.

Серьезно, не делайте этого. Например, *не стоит* жестко зашивать учетные данные вашей базы данных непосредственно в код Terraform и сохранять его в системе управления версиями:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # НЕ ДЕЛАЙТЕ ЭТОГО!!!
  username = "admin"
  password = "password"
  # НЕ ДЕЛАЙТЕ ЭТОГО!!!
}
```

Хранить секреты в открытом виде в системе управления версиями — плохая идея, и тому есть несколько причин.

- Любой, кто имеет доступ к системе управления версиями, имеет доступ к этим секретам. В примере выше любой разработчик вашей компании, имеющий доступ к этому коду Terraform, сможет узнать учетные данные суперпользователя вашей базы данных.
- Каждый компьютер, имеющий доступ к системе управления версиями, хранит копию этих конфиденциальных данных. Каждый компьютер, когда-либо извлекавший код из этого репозитория, может хранить копию этих

конфиденциальных данных на своем локальном жестком диске. Сюда относятся компьютеры разработчиков в вашей команде, компьютеры, участвующие в процессе непрерывной интеграции (например, Jenkins, CircleCI, GitLab и т. д.), компьютеры, участвующие в управлении версиями (например, GitHub, GitLab, BitBucket), компьютеры, участвующие в развертывании (например, все ваши тестовые и промышленные среды), компьютеры, хранящие резервные копии (например, CrashPlan, Time Machine и т. д.), и др.

- *Любое программное обеспечение, которое вы запускаете, имеет доступ к этим конфиденциальным данным.* Поскольку секреты хранятся в открытом виде на многих жестких дисках, любое программное обеспечение, выполняющееся на любом из этих компьютеров, потенциально может прочитать эти конфиденциальные данные.
- *Нет никакой возможности проверить или отозвать доступ к этой конфиденциальной информации.* Когда секреты хранятся на сотнях жестких дисков в открытом виде, нет возможности узнать, кто имел к ним доступ (нет журнала аудита), и нет простого способа ограничить доступ.

Проще говоря, храня секреты в открытом виде, вы даете злоумышленникам (хакерам, конкурентам, недовольным бывшим сотрудникам и т. д.) возможность завладеть особо важными данными вашей компании — например, путем взлома системы управления версиями или используемых вами компьютеров или скомпрометировав какое-либо программное обеспечение на любом из этих компьютеров, и вы не будете знать, произошла ли утечка секретных данных, и не сможете своевременно предпринять какие-либо меры противодействия.

Поэтому очень важно использовать правильный инструмент управления для сохранения конфиденциальных данных в секрете.

Инструменты управления секретами

Всесторонний обзор всех аспектов управления секретами выходит за рамки этой книги, но, чтобы вы могли использовать инструменты управления секретами в комплексе с Terraform, стоит кратко затронуть следующие темы:

- типы хранимых секретов;
- как хранить секреты;
- интерфейс, используемый для доступа к секретам;
- сравнение инструментов управления секретами.

Типы хранимых секретов

Существует три основных типа секретов: личные секреты, секреты клиентов и секреты инфраструктуры.

Личные секреты — это секреты, принадлежащие отдельному лицу, например имена пользователей и пароли для веб-сайтов, SSH-ключи, ключи PGP.

Секреты клиентов — это секреты, принадлежащие вашим клиентам. Обратите внимание: если вы управляете программным обеспечением, которым пользуются другие сотрудники вашей компании (например, внутренним сервером Active Directory вашей компании), то эти сотрудники выступают вашими клиентами. Примеры секретов: имена пользователей и пароли, которые ваши клиенты используют для входа в ваш продукт; личная информация ваших клиентов; их личная медицинская информация.

Секреты инфраструктуры — это секреты, принадлежащие вашей инфраструктуре. Примеры: пароли баз данных; ключи API; сертификаты TLS.

Большинство инструментов управления секретами предназначены для хранения секретов только одного из этих типов. Конечно, можно попытаться хранить в них секреты других типов, но это редко бывает хорошей идеей с точки зрения безопасности или удобства использования. Например, способ хранения паролей, являющихся секретами инфраструктуры, полностью отличается от способа хранения паролей, являющихся секретами клиентов. В первом случае обычно используется алгоритм шифрования, такой как AES (может быть одноразовый), чтобы иметь возможность расшифровать секреты и вернуть исходный пароль, тогда как во втором чаще всего применяется алгоритм хеширования (например, bcrypt) с солью, чтобы не было возможности вернуть исходный пароль. Выбор неправильного подхода может привести к катастрофе, поэтому используйте правильные инструменты!

Как хранить секреты

Две наиболее распространенные стратегии хранения секретов — использование файлового или централизованного хранилища секретов.

Файловые хранилища секретов хранят секреты в зашифрованных файлах, которые обычно передаются в систему управления версиями. Чтобы зашифровать файлы, нужен ключ шифрования. Этот ключ сам по себе является секретом! В результате возникает парадоксальная ситуация: как сохранить этот ключ в безопасности? Ключ нельзя сохранить в системе управления версиями в виде

обычного текста, так как исчезает смысл шифровать что-либо с его помощью. Вы можете зашифровать его с помощью другого ключа, но тогда вам придется придумать, как надежно хранить этот второй ключ.

Наиболее распространенное решение этой проблемы — хранение ключа в службе управления ключами (Key Management Service, KMS), предоставляемой вашим облачным провайдером, такой как AWS KMS, GCP KMS или Azure Key Vault. При таком подходе вы решаете проблему, позволяя облачному провайдеру надежно хранить секреты и управлять доступом к ним. Другой вариант — использовать ключи PGP. Каждый разработчик может иметь свой ключ PGP, состоящий из *открытого* и *закрытого* ключей. Если зашифровать секрет с помощью одного или нескольких открытых ключей, то только разработчики с соответствующими закрытыми ключами смогут расшифровать их. Закрытые ключи, в свою очередь, защищены паролем, который разработчик либо запоминает, либо хранит в личном средстве управления секретами.

Централизованные хранилища секретов обычно имеют вид веб-сервисов, с которыми вы общаетесь по сети. Они шифруют ваши секреты и сохраняют их в своем хранилище данных, таком как MySQL, PostgreSQL, DynamoDB и т. д. Для шифрования секретов этим централизованным хранилищам требуется ключ шифрования. Обычно управление ключом шифрования осуществляет сам сервис, но он также может использовать для этого KMS облачного провайдера.

Интерфейс доступа к секретам

Доступ к большинству инструментов управления секретами можно получить через API, CLI и/или пользовательский интерфейс (UI).

Практически все централизованные хранилища секретов предоставляют API, которому можно посылать сетевые запросы: например, REST API, доступный через HTTP. API удобно использовать, когда ваш код должен программно читать секреты. Например, в процессе загрузки приложение может выполнить вызов API к централизованному хранилищу секретов, чтобы получить пароль для доступа к базе данных. Кроме того, как вы увидите далее в этой главе, можно написать код Terraform, использующий API централизованного хранилища для извлечения секретов.

Все файловые хранилища секретов работают через *интерфейс командной строки* (CLI). Многие из централизованных хранилищ тоже предоставляют инструменты CLI, которые «за кулисами» посылают сервису API-вызовы. Инструменты CLI — это удобный вариант доступа к секретам для разработчиков (например, они могут зашифровать файл, выполнив несколько команд CLI) и для скриптов (например, для шифрования секретов).

Некоторые из централизованных хранилищ секретов предоставляют также *пользовательский интерфейс* (UI), который можно открыть на настольном компьютере или мобильном устройстве. Потенциально это еще более удобный способ доступа к секретам, которым могут пользоваться все члены вашей команды.

Сравнение инструментов управления секретами

В табл. 6.1 приводится сравнение популярных инструментов управления секретами с разбивкой по трем характеристикам, описанным в предыдущих разделах.

Таблица 6.1. Сравнение инструментов управления секретами

	Типы секретов	Хранение секретов	Интерфейс
HashiCorp Vault	Инфраструктурные	Централизованный сервис	UI, API, CLI
AWS Secrets Manager			
Google Secrets Manager			
Azure Key Vault			
Confidant			
Keywhiz			
SOPS		Файлы	CLI
git-secret			
1Password	Личные	Централизованный сервис	UI, API, CLI
LastPass			
BitWarden			
KeePass			
Keychain (macOS)		Файлы	UI, CLI
Credential Manager (Windows)			
pass			
			CLI
Active Directory	Клиентские	Централизованный сервис	UI, API, CLI
Auth0			
Okta			
OneLogin			
Ping			
AWS Cognito			

Поскольку эта книга о Terraform, далее мы сосредоточимся в основном на инструментах управления инфраструктурными секретами, доступ к которым осуществляется через API или CLI (однако время от времени я буду упоминать инструменты управления личными секретами), потому что они часто используются для хранения секретов, необходимых для аутентификации в инструментах инфраструктуры).

Использование инструментов управления секретами в комплексе с Terraform

Теперь перейдем к вопросу использования инструментов управления секретами в комплексе с Terraform и заглянем во все три места, где код Terraform может столкнуться с секретами.

- Провайдеры.
- Ресурсы и источники данных.
- Файлы состояния и файлы планов.

Провайдеры

Обычно первая встреча с секретами при работе с Terraform происходит, когда возникает необходимость пройти аутентификацию у провайдера. Например, если вы хотите запустить `terraform apply` для кода, использующего провайдер AWS, вам необходимо сначала пройти аутентификацию в AWS, а это обычно означает использование ключей доступа, которые являются секретными. Как хранить эти секреты? И как сделать их доступными для Terraform?

Есть много способов ответить на эти вопросы. Один из способов, которым *не следует* пользоваться, хотя он иногда описывается в документации Terraform, — это поместить секреты непосредственно в код в виде обычного текста:

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

  # НЕ ДЕЛАЙТЕ ЭТОГО!!!
  access_key = "(ACCESS_KEY)"
  secret_key = "(SECRET_KEY)"
  # НЕ ДЕЛАЙТЕ ЭТОГО!!!
}
```

Хранение учетных данных в открытом виде небезопасно, как обсуждалось выше, и к тому же непрактично, так как это вынуждает использовать один набор учетных данных для всех пользователей данного модуля, тогда как в большинстве

случаев предпочтительнее использовать разные учетные данные на разных компьютерах (например, когда `apply` выполняется на компьютерах разных разработчиков или на сервере непрерывной интеграции) и в разных средах (`dev`, `stage`, `prod`).

Существуют гораздо более безопасные методы хранения учетных данных и предоставления доступа к ним разным провайдерам Terraform. Давайте рассмотрим эти методы, сгруппировав их по видам пользователей, запускающих Terraform:

- *люди* — разработчики, запускающие Terraform на своих компьютерах;
- *компьютеры* — автоматизированные системы (например, сервер CI), работающие под управлением Terraform без присутствия людей.

Пользователи-люди

Практически все провайдеры Terraform позволяют указывать учетные данные некоторым иным образом, кроме помещения их непосредственно в код. Самый распространенный вариант — переменные окружения. Например, вот как можно использовать переменные окружения для аутентификации в AWS:

```
$ export AWS_ACCESS_KEY_ID=(ID_КЛЮЧА_ДОСТУПА)
$ export AWS_SECRET_ACCESS_KEY=(ЗАКРЫТЫЙ_КЛЮЧ_ДОСТУПА)
```

Использование переменных окружения для хранения учетных данных предотвращает просачивание секретов в открытом виде в ваш код, гарантирует, что каждый, кто запускает Terraform, предоставит свои учетные данные и что эти учетные данные всегда будут храниться только в памяти, а не на диске¹.

Вы можете задать один важный вопрос: где лучше хранить ID ключа доступа и закрытый ключ? Они выглядят как длинные последовательности случайных букв и цифр, слишком длинные и слишком случайные, чтобы их можно было запомнить, но, если хранить их на компьютере в открытом виде, это все равно рискованно. Этот раздел ориентирован на пользователей-людей, поэтому решение состоит в хранении ключей доступа (и других секретов) в менеджере, предназначенном для личных секретов. Например, ключи доступа можно сохранить в 1Password или LastPass и копировать/вставлять их в команды экспорта в терминале.

¹ Обратите внимание, что большинство командных оболочек Linux/Unix/macOS сохраняет историю вводившихся вами команд на диске, в некотором файле (например, `~/.bash_history`). Вот почему показанные здесь команды `export` начинаются с пробела: если команда начинается с пробела, то большинство командных оболочек не будет записывать ее в файл истории. Отметьте также, что вам может потребоваться присвоить переменной окружения `HISTCONTROL` значение `"ignoreboth"`, чтобы включить эту функцию, если у вас она отключена по умолчанию.

Если вам часто требуется указывать эти учетные данные в CLI, то еще удобнее будет использовать менеджер секретов, поддерживающий интерфейс CLI. Например, 1Password предлагает инструмент CLI под названием `op`. Его можно использовать в Mac и Linux для аутентификации в 1Password, как показано ниже:

```
$ eval $(op signin my)
```

После аутентификации, исходя из предположения, что вы сохранили свои ключи доступа в 1Password под именем `aws-dev` с полями `id` и `secret`, можно обратиться к `op`, чтобы сохранить эти ключи в переменных окружения:

```
$ export AWS_ACCESS_KEY_ID=$(op get item 'aws-dev' --fields 'id')
$ export AWS_SECRET_ACCESS_KEY=$(op get item 'aws-dev' --fields 'secret')
```

Такие инструменты, как 1Password и `op`, отлично подходят для управления секретами общего назначения, но для некоторых провайдеров существуют специальные инструменты CLI, еще больше упрощающие эту задачу. Например, для аутентификации в AWS можно использовать инструмент с открытым исходным кодом `aws-vault`. Вот, например, как можно с помощью команды `aws-vault add` сохранить ключи доступа под именем `dev`:

```
$ aws-vault add dev
Enter Access Key Id: (YOUR_ACCESS_KEY_ID)
Enter Secret Key: (YOUR_SECRET_ACCESS_KEY)
```

Утилита `aws-vault` надежно сохранит эти учетные данные в менеджере паролей вашей операционной системы (например, Keychain в macOS или Credential Manager в Windows). После сохранения учетных данных их можно использовать для аутентификации в AWS в любой команде CLI, как показано ниже:

```
$ aws-vault exec <PROFILE> -- <COMMAND>
```

`PROFILE` — это имя профиля, созданного вами ранее командой `add` (например, `dev`), а `COMMAND` — команда, которую нужно выполнить. Например, вот как можно запустить `terraform apply`, используя сохраненные ранее учетные данные `dev`:

```
$ aws-vault exec dev -- terraform apply
```

Команда `exec` автоматически использует AWS STS для получения временных учетных данных и передает их в виде переменных окружения выполняемой вами команде (в данном случае `terraform apply`). Благодаря этому ваши хранимые учетные данные не только остаются надежно защищенными (в менеджере паролей операционной системы), но и предоставляются любому запускаемому вами процессу лишь на время, поэтому риск их утечки сводится к минимуму. `aws-vault` также имеет встроенную поддержку ролей IAM, многоэтапной аутентификации (Multi-Factor Authentication, MFA), входа в учетные записи в веб-консоли и многого другого.

Пользователи машины

Пользователь-человек может вспомнить пароль, но как быть в случаях, когда человека нет рядом? Например, представьте, что вы настраиваете конвейер непрерывной интеграции/непрерывной доставки (CI/CD), который автоматически запускает код Terraform. Как бы вы безопасно аутентифицировали этот конвейер? В данном случае речь идет об аутентификации пользователя-машины. Вопрос в том, как заставить одну машину (например, сервер CI) аутентифицировать себя на другой машине (например, на серверах AWS API) без передачи каких-либо секретов в открытом виде?

Решение этой задачи во многом зависит от типов задействованных машин, то есть от машины, выполняющей аутентификацию, и машины, на которой выполняется аутентификация. Рассмотрим три примера:

- использование CircleCI как сервера CI с хранимыми секретами;
- использование сервера EC2 с ролями IAM и Jenkins в качестве сервера CI;
- использование GitHub Actions как сервера CI с OIDC.



Упрощенные примеры

Этот раздел содержит примеры, поясняющие, как организовать аутентификацию провайдера в контексте CI/CD, но все остальные аспекты рабочего процесса CI/CD сильно упрощены. Более полные примеры рабочих процессов CI/CD, готовые к использованию в промышленной среде, приводятся в главе 9.

- *Использование CircleCI как сервера CI с хранимыми секретами.* Представьте, что для запуска кода Terraform вам нужно использовать CircleCI, популярную управляемую платформу CI/CD. С помощью CircleCI вы настраиваете этапы сборки в файле `.circleci/config.yml`, где определяете задание для запуска `terraform apply`:

```
version: '2.1'
orbs:
  # Установка Terraform с помощью CircleCi Orb
  terraform: circleci/terraform@1.1.0
jobs:
  # Определение задание для запуска 'terraform apply'
  terraform_apply:
    executor: terraform/default
    steps:
      - checkout          # Клонировать код из git
      - terraform/init    # Запустить 'terraform init'
      - terraform/apply   # Запустить 'terraform apply'
```

```
workflows:
  # Создать рабочий процесс для запуска задания с 'terraform apply',
  # что определено выше
  deploy:
    jobs:
      - terraform_apply
    # Запускать этот процесс только после фиксации в главную ветвь
  filters:
    branches:
      only:
        - main
```

Для аутентификации у провайдера с помощью такого инструмента, как CircleCI, нужно лишь создать учетную запись для компьютера в этом провайдере (то есть учетную запись, используемую исключительно для автоматизации, а не для аутентификации человека) и сохранить эти учетные данные в CircleCI, в так называемом *контексте CircleCI*. После этого, в момент запуска сборки, CircleCI обеспечит доступность учетных данных в таком контексте для ваших рабочих процессов в форме переменных окружения. Например, если ваш код Terraform должен аутентифицироваться в AWS, создайте нового пользователя IAM в AWS, определите разрешения, необходимые для развертывания изменений Terraform, и вручную скопируйте ключи доступа этого пользователя в контекст CircleCI, как показано на рис. 6.1.

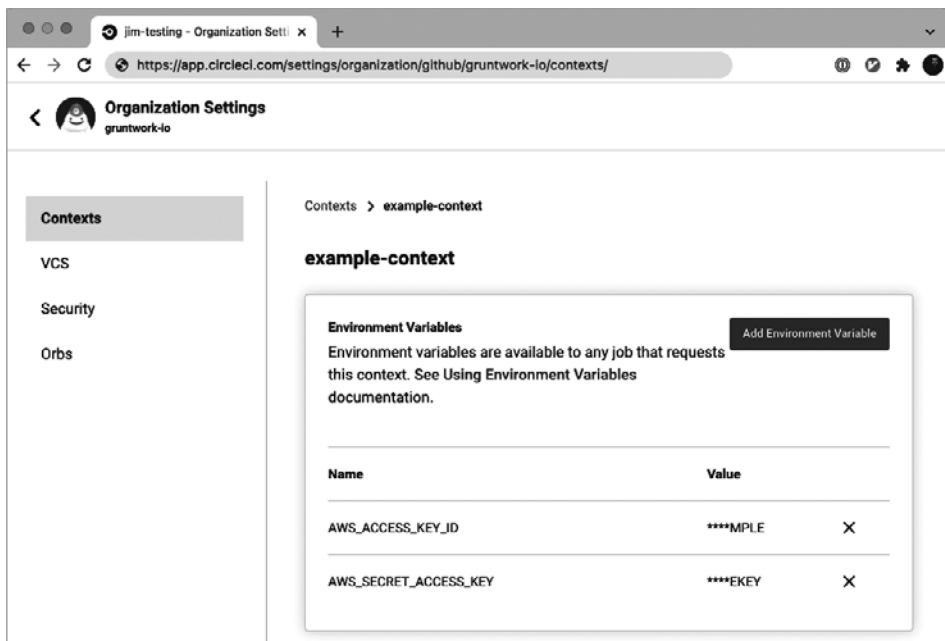


Рис. 6.1. Контекст CircleCI с учетными данными AWS

После этого измените настройки рабочих процессов в файле `.circleci/config.yml`, задействовав контекст CircleCI через параметр `context`:

```
workflows:
  # Создать рабочий процесс для запуска задания с 'terraform apply',
  # что определено выше
  deploy:
    jobs:
      - terraform_apply
    # Запускать этот процесс только после фиксации в главную ветвь
    filters:
      branches:
        only:
          - main
    # Экспортировать секреты из контекста CircleCI
    # в форме переменных окружения
    context:
      - example-context
```

Когда запускается сборка, CircleCI автоматически экспортирует секреты в этом контексте в форме переменных окружения — в данном случае `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY` — и `terraform apply` будет автоматически использовать их для аутентификации у вашего провайдера.

Основные недостатки подхода: 1) вам придется вручную управлять учетными данными и 2) из-за этого нужно использовать постоянные учетные данные, которые после сохранения в CircleCI редко (если вообще когда-либо) меняются. Следующие два примера демонстрируют альтернативные подходы.

- *Использование сервера EC2 с ролями IAM и Jenkins в качестве сервера CI.* Если для запуска кода Terraform использовать сервер EC2 — например, с Jenkins в роли сервера CI, то для аутентификации я рекомендую создать для этого сервера EC2 роль IAM. Роль IAM подобна учетной записи пользователя IAM в том смысле, что это объект в AWS, для которого можно определить разрешения IAM. Однако, в отличие от пользователей IAM, роли IAM не связаны с каким-либо одним человеком и не имеют постоянных учетных данных (пароля или ключей доступа). Вместо этого роль может быть *присвоена* другим объектам IAM: например, пользователю IAM можно присвоить роль, чтобы временно получить доступ к другим разрешениям, которых он обычно не имеет; многим сервисам AWS, таким как серверы EC2, можно присваивать роли IAM для предоставления разрешений, которыми обладает ваша учетная запись AWS.

Например, вот код развертывания сервера EC2, который вы видели много раз:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

Чтобы создать роль IAM, необходимо сначала определить *политику присваивания роли*, то есть политику IAM, устанавливающую, кому разрешено присваивать роль IAM. Политику IAM можно записать в простом формате JSON, но в Terraform есть удобный источник данных `aws_iam_policy_document`, который может создать определение JSON за вас. Вот как можно использовать `aws_iam_policy_document` для определения политики присваивания ролей, которая позволит службе EC2 присвоить себе роль IAM:

```
data "aws_iam_policy_document" "assume_role" {
  statement {
    effect = "Allow"
    actions = ["sts:AssumeRole"]

    principals {
      type       = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}
```

Теперь можно воспользоваться ресурсом `aws_iam_role` для создания роли IAM и использования ее описания в формате JSON, полученного из `aws_iam_policy_document`, в качестве политики присваивания роли:

```
resource "aws_iam_role" "instance" {
  name_prefix      = var.name
  assume_role_policy = data.aws_iam_policy_document.assume_role.json
}
```

Итак, у вас есть роль IAM, но по умолчанию роли IAM не дают никаких разрешений. Поэтому нужно сделать следующий шаг — прикрепить к роли IAM одну или несколько политик IAM, которые фактически определяют, что можно делать, обладая этой ролью. Давайте представим, что вы используете Jenkins для запуска кода Terraform, который развертывает серверы EC2. Вы можете с помощью источника данных `aws_iam_policy_document` определить политику IAM, предоставляющую права администратора серверам EC2:

```
data "aws_iam_policy_document" "ec2_admin_permissions" {
  statement {
    effect = "Allow"
    actions = ["ec2:*"]
    resources = ["*"]
  }
}
```

И прикрепить эту политику к своей роли IAM, используя ресурс `aws_iam_role_policy`:

```
resource "aws_iam_role_policy" "example" {
  role   = aws_iam_role.instance.id
  policy = data.aws_iam_policy_document.ec2_admin_permissions.json
}
```

Последний шаг — разрешить серверу EC2 автоматически присваивать себе роль IAM, создав *профиль экземпляра*:

```
resource "aws_iam_instance_profile" "instance" {
  role = aws_iam_role.instance.name
}
```

И передать этот профиль серверу EC2 через параметр `iam_instance_profile`:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

  # Подключить профиль экземпляра
  iam_instance_profile = aws_iam_instance_profile.instance.name
}
```

За кулисами AWS запустит для каждого сервера EC2 *конечную точку метаданных экземпляра*, доступную по адресу <http://169.254.169.254>. К этой конечной точке могут обращаться только процессы, работающие на самом сервере EC2, чтобы использовать ее для получения метаданных о сервере. Например, подключившись по SSH к серверу EC2, вы можете запросить эту конечную точку с помощью `curl`:

```
$ ssh ubuntu@<IP_OF_INSTANCE>
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-1022-aws x86_64)
(...)
```

```
$ curl http://169.254.169.254/latest/meta-data/
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
events/
hibernation/
hostname
identity-credentials/
(...)
```

Если к серверу EC2 прикреплена роль IAM (через профиль экземпляра), то эти метаданные будут включать учетные данные AWS, которые можно использовать для аутентификации в AWS и присваивания этой роли IAM. Любой инструмент, использующий AWS SDK, такой как Terraform, знает, как автоматически использовать эти учетные данные, поэтому, запустив `terraform apply` на сервере EC2 с этой ролью IAM, вы аутентифицируете

свой код Terraform с разрешениями этой роли IAM, что даст ему права администратора EC2, необходимые для успешной работы¹.

Для любого автоматизированного процесса, запущенного в AWS, например, сервера CI, роли IAM обеспечивают возможность аутентификации: 1) без необходимости управлять учетными данными вручную и 2) с использованием временных учетных данных, которые AWS предоставляет через конечную точку метаданных экземпляра и которые меняются автоматически. Это два больших преимущества перед постоянными учетными данными, управляемыми вручную с помощью такого инструмента, как CircleCI, который работает за границами вашей учетной записи AWS. Однако, как показано в следующем примере, иногда те же преимущества можно получить и для внешних инструментов.

- *Использование GitHub Actions как сервера CI с OIDC.* GitHub Actions — еще одна популярная управляемая платформа CI/CD, которую можно использовать для запуска Terraform. Раньше сервис GitHub Actions требовал вручную копировать учетные данные, как в примере с CircleCI. Однако с 2021 года GitHub Actions предлагает лучшую альтернативу: поддержку *Open ID Connect (OIDC)*. С помощью OIDC можно установить доверенное соединение между системой CI и облачным провайдером (GitHub Actions поддерживает AWS, Azure и Google Cloud), чтобы ваша система CI могла аутентифицироваться без необходимости вручную управлять какими-либо учетными данными.

Представим, что вы определили рабочие процессы GitHub Actions в файлах YAML в папке `.github/workflows`, например в файле `terraform.yml`:

```
name: Terraform Apply
# Запускать этот процесс только после фиксации в главной ветви
on:
  push:
    branches:
      - 'main'
jobs:
```

¹ По умолчанию конечная точка метаданных экземпляра открыта для всех пользователей ОС, работающих на ваших серверах EC2. Я рекомендую заблокировать ее, чтобы только определенные пользователи ОС могли получить к ней доступ: например, если вы запускаете приложение на сервере EC2 как пользователь `app`, то можете использовать `iptables` или `nftables`, чтобы разрешить доступ к конечной точке метаданных экземпляра только пользователю `app`. В таком случае, если злоумышленник обнаружит какую-либо уязвимость и сможет выполнить код на вашем сервере, он получит доступ к разрешениям роли IAM, только если сумеет аутентифицироваться как пользователь `app` (а не как любой другой). Еще лучше, если разрешения роли IAM нужны только во время загрузки (например, для чтения пароля базы данных). В таком случае можно полностью отключить конечную точку метаданных экземпляра после загрузки, чтобы злоумышленник, получивший доступ позже, вообще не мог использовать ее.


```

TerraformApply:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2

    # Запустить Terraform, используя действие
    # setup-terraform от HashiCorp
    - uses: hashicorp/setup-terraform@v1
      with:
        terraform_version: 1.1.0
        terraform_wrapper: false
  run: |
    terraform init
    terraform apply -auto-approve

```

Если код Terraform должен взаимодействовать с таким провайдером, как AWS, то вы должны предоставить данному рабочему процессу возможность аутентификации у этого провайдера. Чтобы сделать это с помощью OIDC¹, сначала нужно создать *провайдер удостоверений IAM OIDC* в вашей учетной записи AWS, используя ресурс `aws_iam_openid_connect_provider`, и настроить его так, чтобы он доверял ключу GitHub Actions, полученному через источник данных `tls_certificate`:

```

# Создать провайдера удостоверений IAM OIDC
# и настроить доверие к GitHub
resource "aws_iam_openid_connect_provider" "github_actions" {
  url          = "https://token.actions.githubusercontent.com"
  client_id_list = ["sts.amazonaws.com"]
  thumbprint_list = [
    data.tls_certificate.github.certificates[0].sha1_fingerprint
  ]
}

# Получить ключ GitHub OIDC
data "tls_certificate" "github" {
  url = "https://token.actions.githubusercontent.com"
}

```

Теперь можно создать роли IAM, действуя точно так же, как в предыдущем разделе. Вот, например, роль IAM с правами администратора EC2 (правда, политика присваивания для этих ролей IAM будет выглядеть иначе):

```

data "aws_iam_policy_document" "assume_role_policy" {
  statement {

```

¹ На момент написания книги поддержка OIDC между GitHub Actions и AWS продолжала активно трансформироваться и что-то в ней могло измениться. Обязательно загляните в документацию GitHub OIDC (<https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services>), чтобы познакомиться с последними обновлениями.

```

actions = ["sts:AssumeRoleWithWebIdentity"]
effect   = "Allow"

principals {
  identifiers = [aws_iam_openid_connect_provider.github_actions.arn]
  type       = "Federated"
}

condition {
  test      = "StringEquals"
  variable  = "token.actions.githubusercontent.com:sub"
  # Эту роль IAM смогут присвоить себе репозитории и ветви,
  # определяемые в var.allowed_repos_branches
  values = [
    for a in var.allowed_repos_branches :
    "repo:${a["org"]}/${a["repo"]}:ref:refs/heads/${a["branch"]}"
  ]
}
}
}

```

Эта политика позволяет провайдеру удостоверений IAM OIDC присвоить себе роль IAM посредством федеративной аутентификации. Обратите внимание на блок `condition`, который гарантирует, что только репозитории и ветви в GitHub, указанные во входной переменной `allowed_repos_branches`, смогут присвоить себе эту роль IAM:

```

variable "allowed_repos_branches" {
  description = "GitHub repos/branches allowed to assume the IAM role."
  type = list(object({
    org    = string
    repo   = string
    branch = string
  }))
  # Пример:
  # allowed_repos_branches = [
  #   {
  #     org    = "brikis98"
  #     repo   = "terraform-up-and-running-code"
  #     branch = "main"
  #   }
  # ]
}

```

Это важно, чтобы вы случайно не разрешили всем репозиториям GitHub аутентифицироваться в вашей учетной записи AWS! Теперь можно настроить сборку в GitHub Actions, чтобы она присваивала себе эту роль IAM. Во-первых, в начале определения вашего рабочего процесса дайте сборке разрешение `id-token: write`:

```

permissions:
  id-token: write

```

Затем непосредственно перед запуском Terraform добавьте шаг аутентификации в AWS с помощью действия `configure-aws-credentials`:

```
# Аутентификация в AWS с помощью OIDC
- uses: aws-actions/configure-aws-credentials@v1
  with:
    # Здесь определяется присваиваемая роль IAM
    role-to-assume: arn:aws:iam::123456789012:role/example-role
    aws-region: us-east-2

# Запустить Terraform, используя действие
# setup-terraform от HashiCorp
- uses: hashicorp/setup-terraform@v1
  with:
    terraform_version: 1.1.0
    terraform_wrapper: false
  run: |
    terraform init
    terraform apply -auto-approve
```

Теперь, когда вы запустите эту сборку в одном из репозиториях и ветвей, перечисленных в переменной `allowed_repos_branches`, GitHub сможет автоматически присвоить себе вашу роль IAM, используя временные учетные данные, а Terraform аутентифицируется в AWS, используя эту роль IAM, и для этого не потребуется управлять никакими учетными данными вручную.

Ресурсы и источники данных

Следующее место, где код Terraform может столкнуться с секретами, — это ресурсы и источники данных. Например, выше в главе вы видели пример передачи учетных данных для доступа к базе данных в ресурсе `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # НЕ ДЕЛАЙТЕ ЭТОГО!!!
  username = "admin"
  password = "password"
  # НЕ ДЕЛАЙТЕ ЭТОГО!!!
}
```

В этой главе я уже говорил об этом моменте, но он настолько важный, что стоит повторить еще раз: хранить учетные данные в коде в открытом виде — плохая идея. Но как правильно поступать в таких ситуациях?

Есть три основных варианта:

- переменные окружения;
- зашифрованные файлы;
- хранилища секретов.

Переменные окружения

Вариант с переменными окружения вы уже видели в главе 3, а также выше в этой главе, когда мы говорили о провайдерах и убирали секреты из кода, используя преимущества встроенной в Terraform возможности чтения переменных окружения.

Чтобы использовать этот метод, объявите переменные, через которые вы будете передавать секреты:

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}

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

Как и в главе 3, эти переменные включают параметр `sensitive = true`, чтобы предотвратить вывод их содержимого при выполнении команд `plan` или `apply`. Также у этих переменных не должно быть значений по умолчанию (чтобы не хранить секреты в открытом виде).

Теперь передайте переменные ресурсам Terraform, которым нужны эти секреты:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine            = "mysql"
  allocated_storage = 10
  instance_class    = "db.t2.micro"
  skip_final_snapshot = true
  db_name            = var.db_name

  # Передача секретов в ресурс
  username = var.db_username
  password = var.db_password
}
```

Сейчас вы можете присвоить значение каждой переменной `foo`, определив переменную окружения `TF_VAR_foo`:

```
$ export TF_VAR_db_username=(DB_USERNAME)
$ export TF_VAR_db_password=(DB_PASSWORD)
```

Передача секретов через переменные окружения избавляет от необходимости хранить секреты в коде, но не дает ответа на важный вопрос: как безопасно их хранить? Одна из приятных особенностей способа хранения на основе переменных окружения — они подходят практически для любого решения управления секретами. Например, один из вариантов — сохранять секреты в личном менеджере секретов (таком как 1Password) и вручную присваивать их переменным окружения в терминале. Другой вариант — сохранить секреты в централизованном хранилище (например, HashiCorp Vault) и написать сценарий, использующий API или CLI этого хранилища для чтения секретов и присваивания их переменным окружения.

Использование переменных окружения имеет следующие преимущества.

- Не допускается попадание секретных данных в открытом виде в код и систему управления версиями.
- Простота хранения секретов, поскольку можно использовать практически любое другое решение для управления секретами. То есть, если в вашей компании уже практикуется некий вариант управления секретами, часто можно найти способ заставить его работать с переменными окружения.
- Простота получения секретов, поскольку чтение переменных окружения можно реализовать на любом языке.
- Простота интеграции с автоматическими тестами, поскольку переменные окружения легко использовать для имитации значений.
- Отсутствие затрат на использование переменных окружения в отличие от других решений по управлению секретами, которые обсуждаются ниже.

Однако использование переменных окружения имеет свои недостатки.

- Отсутствие определений в самом коде Terraform усложняет его понимание и сопровождение. Каждый, кто использует ваш код, должен знать, что нужно предпринять дополнительные шаги, чтобы вручную создать эти переменные окружения или запустить сценарий-обертку.
- Более высокая сложность стандартизации методов управления секретами. Поскольку все управление секретами происходит за пределами Terraform, код не обеспечивает никаких свойств безопасности и, возможно, кто-то продолжит управлять секретами небезопасным способом (например, хранить их в открытом виде).

- Поскольку секреты не версионированы, не упаковываются и не тестируются вместе с кодом, увеличивается вероятность ошибок конфигурации, скажем, можно добавить новый секрет в одной среде (например, в тестовой), но забыть сделать это в другой (например, в промышленной).

Зашифрованные файлы

Второй метод основан на шифровании секретов, сохранении зашифрованных данных в файле и передаче этого файла в систему управления версиями.

Чтобы зашифровать некоторые данные, потребуется ключ шифрования. Как упоминалось выше в этой главе, он сам является секретом, поэтому вам понадобится безопасный способ его хранения. Типичное решение — использовать KMS вашего облачного провайдера (например, AWS KMS, Google KMS, Azure Key Vault) или ключи PGP одного или нескольких разработчиков из вашей команды.

Рассмотрим пример использования AWS KMS. Прежде всего необходимо создать *ключ, управляемый клиентом (Customer Managed Key, CMK)*, — ключ шифрования, которым будет управлять AWS. Чтобы создать CMK, сначала необходимо определить *политику ключей*, то есть политику IAM, определяющую, кто может использовать этот CMK. Для простоты создадим политику ключей, которая предоставит текущему пользователю права администратора через CMK. Получить информацию о текущем пользователе — его имя пользователя, ARN и т. д. — можно с помощью источника данных `aws_caller_identity`:

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

data "aws_caller_identity" "self" {}
```

И теперь можно использовать выходные данные источника `aws_caller_identity` в источнике данных `aws_iam_policy_document` для создания политики ключей, предоставляющих текущему пользователю права администратора через CMK:

```
data "aws_iam_policy_document" "cmk_admin_policy" {
  statement {
    effect = "Allow"
    resources = ["*"]
    actions = ["kms:*"]
    principals {
      type = "AWS"
      identifiers = [data.aws_caller_identity.self.arn]
    }
  }
}
```

Далее можно создать СМК, используя ресурс `aws_kms_key`:

```
resource "aws_kms_key" "cmk" {  
  policy = data.aws_iam_policy_document.cmk_admin_policy.json  
}
```

Обратите внимание, что по умолчанию KMS идентифицирует СМК только по длинному числовому идентификатору (например, `b7670b0e-ed67-28e4-9b15-0d61e1485be3`), поэтому рекомендуется также создать удобный *псевдоним* для СМК, используя ресурс `aws_kms_alias`:

```
resource "aws_kms_alias" "cmk" {  
  name          = "alias/kms-cmk-example"  
  target_key_id = aws_kms_key.cmk.id  
}
```

Псевдоним выше позволит вам ссылаться на свой СМК по псевдониму `alias/kms-cmk-example`, а не по длинному идентификатору, такому как `b7670b0e-ed67-28e4-9b15-0d61e1485be3`. Создав СМК, можете начать использовать его для шифрования и расшифровывания данных. Учтите, что по замыслу вы никогда не сможете увидеть основной ключ шифрования (и, следовательно, допустить его утечку). Только AWS имеет доступ к этому ключу шифрования, но вы можете использовать его с помощью AWS API и интерфейса командной строки, как описано ниже.

Сначала создайте файл `db-creds.yml` с некоторыми секретами, например с учетными данными для доступа к базе данных:

```
username: admin  
password: password
```

Обратите внимание: этот файл *не нужно* сохранять в системе управления версиями, потому что он еще не зашифрован! Зашифровать эти данные можно с помощью команды `aws kms encrypt` и сохранить полученный зашифрованный текст в новом файле. Вот небольшой Bash-скрипт (для Linux/Unix/macOS) с именем `encrypt.sh`, выполняющий эти шаги с помощью AWS CLI:

```
CMK_ID="$1"  
AWS_REGION="$2"  
INPUT_FILE="$3"  
OUTPUT_FILE="$4"  
  
echo "Encrypting contents of $INPUT_FILE using CMK $CMK_ID..."  
ciphertext=$(aws kms encrypt \  
  --key-id "$CMK_ID" \  
  --region "$AWS_REGION" \  
  --plaintext "fileb://$INPUT_FILE" \  
  --output text \  
  --query CiphertextBlob)
```

```
echo "Writing result to $OUTPUT_FILE..."
echo "$ciphertext" > "$OUTPUT_FILE"

echo "Done!"
```

Вот как можно зашифровать файл `db-creds.yml` с помощью скрипта `encrypt.sh` и созданного ранее KMS CMK, а затем сохранить зашифрованный текст в новом файле с именем `db-creds.yml.encrypted`:

```
$ ./encrypt.sh \
  alias/kms-cmk-example \
  us-east-2 \
  db-creds.yml \
  db-creds.yml.encrypted
```

```
Encrypting contents of db-creds.yml using CMK alias/kms-cmk-example...
Writing result to db-creds.yml.encrypted...
Done!
```

Теперь можно удалить `db-creds.yml` (обычный текстовый файл) и без опаски сохранить `db-creds.yml.encrypted` (зашифрованный файл) в системе управления версиями. Итак, теперь у вас есть зашифрованный файл с некоторыми секретами, но как использовать этот файл в коде Terraform?

Сначала нужно расшифровать секреты в этом файле с помощью источника данных `aws_kms_secrets`:

```
data "aws_kms_secrets" "creds" {
  secret {
    name      = "db"
    payload   = file("${path.module}/db-creds.yml.encrypted")
  }
}
```

Код выше читает `db-creds.yml.encrypted` с диска с помощью вспомогательной функции `file` и при наличии разрешения на доступ к соответствующему ключу в KMS расшифровывает содержимое этого файла. Код вернет вам содержимое исходного файла `db-creds.yml`, поэтому следующим шагом будет анализ разметки YAML, как показано ниже:

```
locals {
  db_creds = yamldecode(data.aws_kms_secrets.creds.plaintext["db"])
}
```

Этот код извлекает данные для доступа к базе данных из источника `aws_kms_secrets`, анализирует разметку YAML и сохраняет результаты в локальной переменной `db_creds`. После этого можно прочитать имя пользователя и пароль из `db_creds` и передать их ресурсу `aws_db_instance`:


```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # Передача секретов в ресурс
  username = local.db_creds.username
  password = local.db_creds.password
}
```

Итак, теперь вы знаете, как сохранить секреты в зашифрованном файле и отправить его в систему управления версиями, а также как автоматически прочитать эти секреты из файла в коде Terraform. Однако следует отметить, что работать с зашифрованными файлами неудобно. Чтобы внести изменения в такой файл, вам придется расшифровать его на локальном компьютере с помощью длинной команды `aws kms decrypt`, внести некоторые изменения, повторно зашифровать файл с помощью другой длинной команды `aws kms encrypt` и при этом проявить максимальную осторожность, чтобы случайно не отправить открытый текст в систему управления версиями или не оставить их на своем компьютере. Это утомительный и чреватый ошибками процесс.

Одно из возможных решений — использовать инструмент с открытым исходным кодом `sops` (<https://github.com/mozilla/sops>). Команда `sops <FILE>` автоматически расшифрует файл `FILE` и откроет текстовый редактор по умолчанию с содержимым в виде обычного текста. Когда вы закончите редактирование и закроете текстовый редактор, `sops` автоматически зашифрует содержимое. В результате этапы шифрования и дешифрования становятся практически прозрачными, отпадает необходимость запускать длинные команды `aws kms` и уменьшается вероятность случайно отправить секретные данные в открытом виде в систему управления версиями. Начиная с 2022 года, `sops` может работать с файлами, зашифрованными с помощью ключей AWS KMS, GCP KMS, Azure Key Vault или PGP. Обратите внимание, что Terraform пока не имеет встроенной поддержки расшифровки файлов, зашифрованных с помощью `sops`, поэтому вам придется использовать сторонний провайдер, например `carlpett/sops` (<https://registry.terraform.io/providers/carlpett/sops/latest/docs>), или, если вы пользуетесь Terragrunt, встроенную функцию `sops_decrypt_file` (https://terragrunt.gruntwork.io/docs/reference/built-in-functions/#sops_decrypt_file).

Прием с использованием зашифрованных файлов имеет следующие преимущества.

- Не допускает попадания секретных данных в открытом виде в код и в систему управления версиями.

- Секреты хранятся в системе управления версиями в зашифрованном виде, поэтому они извлекаются, упаковываются и тестируются вместе с остальным кодом. Это помогает уменьшить количество ошибок конфигурации, когда новый секрет добавляется в одной среде (например, в тестовой) и не добавляется по забывчивости в другой (например, в промышленной).
- Легко получить секреты, если используемый вами формат шифрования изначально поддерживается Terraform или сторонним плагином.
- Работает с множеством различных вариантов шифрования: AWS KMS, GCP KMS, PGP и т. д.
- Все определено в коде. Не требуется никаких дополнительных действий вручную или сценариев-оберток (хотя для интеграции с `sops` нужен сторонний плагин).

И недостатки.

- Хранить секреты сложнее. Вам придется выполнить множество команд (например, `aws kms encrypt`) или использовать внешний инструмент, такой как `sops`. Но предварительно необходимо научиться работать с этими инструментами!
- Интеграция с автоматическими тестами сложнее, поскольку придется проделывать дополнительную работу, чтобы сделать ключи шифрования и зашифрованные данные доступными во всех ваших тестовых средах.
- Секреты теперь зашифрованы, но хранение в системе управления версиями затрудняет их изменение и отзыв. Если кто-нибудь когда-нибудь скомпрометирует ключ шифрования, он сможет вернуться и расшифровать все секреты, которые когда-либо были зашифрованы с его помощью.
- Возможность выявить, кто получил доступ к секретам, практически отсутствует. Если вы используете облачную систему управления ключами (например, AWS KMS), то с помощью журнала аудита сможете узнать, кто использовал ключ шифрования, но не сможете определить, для чего на самом деле (то есть к каким секретам был получен доступ).
- Сервисы управления ключами обычно стоят недорого. Например, хранение каждого ключа в AWS KMS вам обойдется в 1 доллар США в месяц плюс 0,03 доллара США за 10 000 вызовов API, при этом для каждой операции дешифрования и шифрования требуется один вызов API. Типичная схема использования, когда у вас есть небольшое количество ключей в KMS и ваши приложения применяют эти ключи для расшифровывания секретов во время загрузки, обычно обходится в 1–10 долларов в месяц. Для более крупных развертываний с десятками приложений и сотнями секретов цена обычно находится в диапазоне от 10 до 50 долларов в месяц.

- Высокая сложность стандартизации методов управления секретами. Разные разработчики или команды могут предпочитать разные способы хранения ключей шифрования или управления зашифрованными файлами, и высока вероятность ошибок, таких как неправильное использование шифрования или случайное сохранение простого текстового файла в системе управления версиями.

Хранилища секретов

Третий метод основан на хранении ваших секретов в централизованном хранилище.

К наиболее популярным хранилищам секретов можно отнести AWS Secrets Manager, Google Secrets Manager, Azure Key Vault и HashiCorp Vault.

Рассмотрим пример с использованием AWS Secrets Manager. Первый шаг — сохранение учетных данных для доступа к базе данных в AWS Secrets Manager. Это можно сделать с помощью веб-консоли AWS (рис. 6.2).

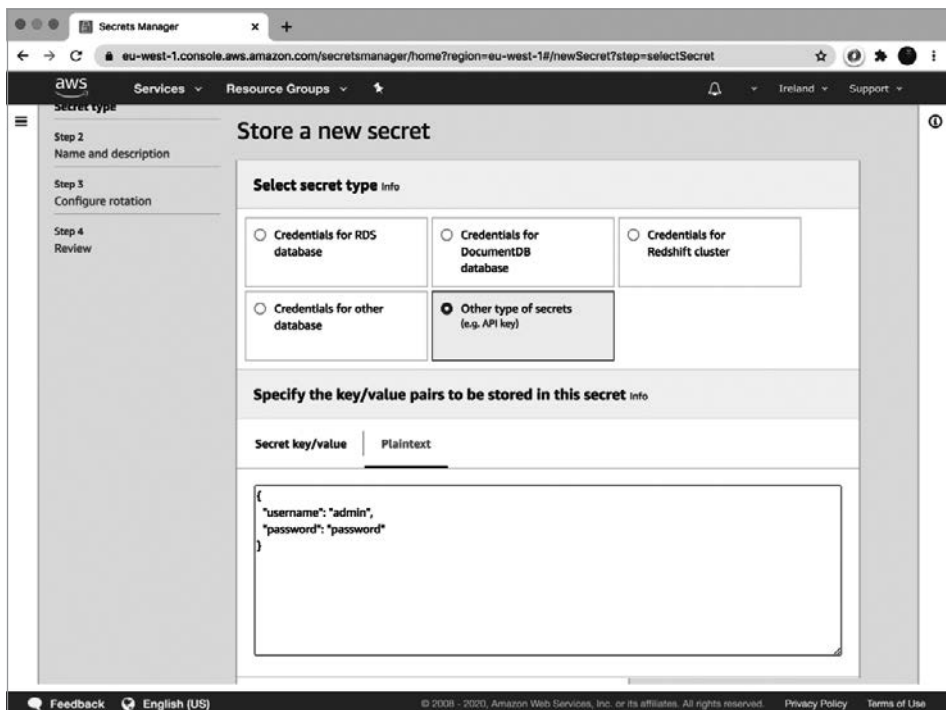


Рис. 6.2. Сохранение секретов в формате JSON в AWS Secrets Manager

Обратите внимание, что секреты на рис. 6.2 представлены в формате JSON, рекомендованном для хранения данных в AWS Secrets Manager.

Перейдите к следующему шагу и обязательно присвойте секрету уникальное имя, например db-creds (рис. 6.3).

The screenshot shows the AWS Secrets Manager console in the 'eu-west-1' region. The 'Store a new secret' wizard is in progress, specifically at the 'Secret name and description' step. The 'Secret name' field is filled with 'db-creds', and the 'Description' field contains 'Database credentials'. Below these fields, there is a section for 'Tags - optional' which is currently empty. At the bottom of the wizard, there are buttons for 'Cancel', 'Previous', and 'Next'. The 'Next' button is highlighted, indicating the user should proceed to the next step.

Рис. 6.3. Присвоение секрету уникального имени в AWS Secrets Manager

Нажмите Next (Далее) и Store (Сохранить), чтобы сохранить секрет. Теперь вы сможете в своем коде Terraform использовать источник данных `aws_secretsmanager_secret_version` для чтения секрета db-creds:

```
data "aws_secretsmanager_secret_version" "creds" {
  secret_id = "db-creds"
}
```

Поскольку секрет хранится в формате JSON, для его анализа можно использовать функцию `jsondecode` и сохранить результат в локальной переменной `db_creds`:

```
locals {
  db_creds = jsondecode(
    data.aws_secretsmanager_secret_version.creds.secret_string
  )
}
```

А затем прочитать учетные данные для доступа к базе данных из `db_creds` и передать их в ресурс `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # Передача секретов в ресурс
  username = local.db_creds.username
  password = local.db_creds.password
}
```

Использование хранилищ секретов имеет следующие преимущества.

- Не допускает попадания секретных данных в открытом виде в код и в систему управления версиями.
- Все определено в коде. Не требуется никаких дополнительных действий вручную или сценариев-оберток.
- Просто хранить секреты за счет возможности использовать веб-интерфейс.
- Многие хранилища секретов поддерживают ротацию и отзыв секретов, что может пригодиться в случае раскрытия секрета. В качестве профилактической меры можно даже включить ротацию по расписанию (например, каждые 30 дней).
- Хранилища секретов обычно поддерживают подробные журналы аудита, точно показывающие, кто к каким данным обращался.
- Хранилища секретов упрощают стандартизацию практик управления секретами за счет использования определенных видов шифрования, хранения, шаблонов доступа и т. д.

И недостатки.

- Поскольку секреты не версионизируются, не упаковываются и не тестируются вместе с кодом, увеличивается вероятность ошибок конфигурации, например, можно добавить новый секрет в одной среде (например, в тестовой), но забыть сделать это в другой (например, в промышленной).
- Услуги большинства управляемых хранилищ секретов стоят денег. В частности, AWS Secrets Manager взимает 0,40 доллара США в месяц за каждый хранимый секрет плюс 0,05 доллара США за каждые 10 000 вызовов API, которые совершаются для сохранения или получения данных. Типичная схема использования, когда у вас есть несколько десятков секретов, хранящихся в нескольких средах, и несколько приложений, читающих эти секреты

во время загрузки, обычно обходится в 10–25 долларов в месяц. Для более крупных развертываний с десятками приложений и сотнями секретов цена может достигать сотен долларов в месяц.

- Если вы используете самоуправляемое хранилище секретов, такое как HashiCorp Vault, то одновременно тратите деньги на его работу (например, платите AWS за 3–5 серверов EC2 для запуска Vault в режиме высокой доступности), а также время и деньги — на развертывание, настройку, обновление и мониторинг хранилища. Время разработчиков стоит очень дорого, поэтому в зависимости от количества времени, которое им придется потратить на управление хранилищем секретов, это может стоить вам несколько тысяч долларов в месяц.
- Получить секреты сложнее, особенно в автоматизированных средах (например, когда приложение загружается и пытается прочесть пароль для доступа к базе данных), поскольку вам придется решить, как организовать безопасную аутентификацию между несколькими компьютерами.
- Интеграция с автоматическими тестами сложнее, поскольку большая часть тестируемого кода теперь зависит от внешней системы, которую необходимо имитировать и в которой нужно хранить тестовые данные.

Файлы состояний и файлы планов

Еще два места, где вы встретите секреты при использовании Terraform:

- файлы состояний;
- файлы планов.

Файлы состояний

Надеюсь, в этой главе я убедил вас не хранить секреты в открытом виде и смог показать более удачные альтернативы. Однако многих пользователей Terraform застает врасплох тот факт, что независимо от выбранного метода *любые секреты, которые передаются в ресурсы и источники данных Terraform, в конечном итоге оказываются в состоянии Terraform в открытом виде!*

Например, откуда бы вы ни читали учетные данные для доступа к базе данных — из переменных окружения, зашифрованных файлов, централизованного хранилища секретов, — после передачи этих учетных данных такому ресурсу, как `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
```

```
instance_class      = "db.t2.micro"
skip_final_snapshot = true
db_name             = var.db_name

# Передача секретов в ресурс
username = local.db_creds.username
password = local.db_creds.password
}
```

Terraform сохранит их в файле `terraform.tfstate` в открытом виде. Этот вопрос остается открытым с 2014 года (<https://bit.ly/33gqaVe>), и четких планов по его решению пока нет. Есть несколько обходных путей, помогающих удалить секреты из файлов состояния, но они довольно хрупкие и могут перестать работать с ближайшим новым выпуском Terraform, поэтому я не рекомендую их.

На данный момент, независимо от выбора конкретного метода управления секретами, вы должны делать следующее.

- *Хранить состояние Terraform в серверном хранилище, поддерживающем шифрование.* Для хранения состояния вместо локального файла `terraform.tfstate`, помещаемого в систему управления версиями, следует использовать один из серверов, поддерживаемых Terraform, который изначально использует шифрование, например S3, GCS и Azure Blob Storage. Эти серверы будут шифровать файлы состояния как при передаче (например, через TLS), так и перед сохранением на диск (например, с помощью алгоритма AES-256).
- *Строго контролируйте, кто имеет доступ к вашему серверу Terraform.* Поскольку файлы состояния Terraform могут содержать секреты, нужно контролировать, кто имеет доступ к серверу, так же тщательно, как доступ к самим секретам. Например, если в качестве серверного хранилища у вас S3, настройте политику IAM, которая предоставляет доступ к корзине S3, используемой промышленной средой, очень узкому кругу доверенных разработчиков или, может быть, только серверу CI, который применяется для развертывания промышленной версии.

Файлы планов

Вы много раз встречали команду `terraform plan`. Одна особенность, о которой вы, вероятно, еще не знаете, заключается в возможности сохранить вывод команды `plan` (diff) в файле:

```
$ terraform plan -out=example.plan
```

Команда выше сохраняет план в файле `example.plan`. После этого можно запустить команду `apply` с этим файлом плана, чтобы гарантировать, что Terraform применит именно те изменения, которые вы видели изначально:

```
$ terraform apply example.plan
```

Это удобная особенность Terraform, но важно помнить, что, как и в случае с состоянием Terraform, *любые секреты, которые передаются в ресурсы и источники данных, в конечном итоге окажутся в вашем плане Terraform в открытом виде*. Например, если вы запустили `plan` для кода `aws_db_instance` и сохранили файл плана, то этот файл будет содержать имя пользователя и пароль базы данных в открытом виде.

Поэтому, если вы собираетесь использовать файлы планов, сделайте следующее.

- *Зашифруйте файлы планов Terraform.* Если планируете сохранять файлы планов, найдите способ зашифровать их как при передаче (например, через TLS), так и перед сохранением на диске (например, с помощью алгоритма AES-256). В частности, файлы планов можно хранить в корзине S3, поддерживающей оба типа шифрования.
- *Строго контролируйте список тех, кто может получить доступ к вашему плану.* Поскольку файлы планов Terraform могут содержать секреты, нужно контролировать перечень тех, кто имеет к ним доступ, по крайней мере так же тщательно, как контролируется доступ к самим секретам. Например, если для хранения файлов планов вы используете S3, то настройте политику IAM, которая предоставляет доступ к корзине S3, используемой промышленной средой, очень узкому кругу доверенных разработчиков или, может быть, только серверу CI, который применяется для развертывания промышленной версии.

Резюме

Вот основные выводы из этой главы.

- Во-первых, вы можете не запоминать ничего из этой главы, кроме одного: **не храните секреты в открытом виде**.
- Во-вторых, для передачи секретов провайдерам пользователи-люди могут использовать менеджеры личных секретов и создавать переменные окружения, а пользователи-компьютеры — использовать хранимые учетные данные, роли IAM или OIDC. Краткий перечень компромиссных вариантов для пользователей-компьютеров приводится в табл. 6.2.
- В-третьих, для передачи секретов ресурсам и источникам данных используйте переменные окружения, зашифрованные файлы или централизованные хранилища секретов. В табл. 6.3 приводится краткое сравнение этих трех вариантов.

Таблица 6.2. Сравнение методов передачи секретов пользователями-компьютерами (например, серверами CI) провайдерам Terraform

	Хранимые учетные данные	Роли IAM	OIDC
Пример	CircleCI	Jenkins на сервере EC2	GitHub Actions
Не требует управлять учетными данными вручную	✗	✓	✓
Не требует использовать постоянные учетные данные	✗	✓	✓
Действует внутри экосистемы облачного провайдера	✗	✓	✓
Действует за пределами экосистемы облачного провайдера	✓	✗	✓
Широко поддерживается по состоянию на 2022 год	✓	✓	✗

Таблица 6.3. Сравнение методов передачи секретов ресурсам и источникам данных Terraform

	Переменные окружения	Зашифрованные файлы	Централизованные хранилища секретов
Не допускает попадания в код секретов в открытом виде	✓	✓	✓
Секреты управляются как код	✗	✓	✓
Имеется журнал аудита доступа к ключам шифрования	✗	✓	✓
Имеется журнал аудита доступа к индивидуальным секретам	✗	✗	✓
Простота ротации и отзыва секретов	✗	✗	✓
Простота стандартизации управления секретами	✗	✗	✓
Версионирование секретов вместе с кодом	✗	✓	✗
Простота хранения секретов	✓	✗	✓
Простота извлечения секретов	✓	✓	✗
Простота интеграции с автоматизированным тестированием	✓	✗	✗
Затраты	0	\$	\$\$\$

- И наконец, в-четвертых: запомните, что независимо от способа передачи секретов ресурсам и источникам данных Terraform будет выводить эти секреты в файлы состояния и файлы планов в открытом виде, поэтому всегда шифруйте эти файлы (при передаче и перед записью на диск) и строго контролируйте доступ к ним.

Теперь, когда вы понимаете, как управлять секретами при работе с Terraform и как безопасно передавать секреты провайдерам Terraform, перейдем к главе 7, где вы узнаете, как использовать Terraform в случаях, когда имеется несколько провайдеров (например, несколько регионов, учетных записей, облаков).

Работа с несколькими провайдерами

До сих пор почти каждый пример в книге включал только один блок `provider`:

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

Этот блок настраивает код, осуществляющий развертывание в единственном регионе AWS и в единственной учетной записи AWS. В связи с этим возникает несколько вопросов.

1. Как поступить, если потребуется выполнить развертывание в нескольких регионах AWS?
2. Как поступить, если придется выполнить развертывание в нескольких учетных записях AWS?
3. Как поступить, если потребуется выполнить развертывание в других облаках, например Azure или GCP?

Чтобы получить ответы на эти вопросы, в этой главе мы подробнее рассмотрим следующие темы:

- работу с одним провайдером в Terraform;
- работу с несколькими копиями одного провайдера;
- работу с несколькими разными провайдерами.

Работа с одним провайдером

До сих пор вы использовали провайдеров как нечто волшебное. Такой подход оправдывает себя, когда рассматриваются простые примеры с одним базовым провайдером, но, если у вас появится необходимость работать с несколькими регионами, учетными записями, облаками и т. д., вам придется пойти дальше. Начнем с детального рассмотрения одного провайдера, чтобы лучше понять, как он работает.

- Что такое провайдер?
- Как устанавливаются провайдеры?
- Как используются провайдеры?

Что такое провайдер

Представляя провайдеров в главе 2, я описал их как *платформы*, с которыми работает Terraform: например, AWS, Azure, Google Cloud, Digital Ocean и т. д. Как же Terraform взаимодействует с этими платформами?

Внутри Terraform делится на две части.

- *Ядро*. Двоичный файл `terraform`, предоставляющий все основные функции Terraform, используемые всеми платформами, такие как интерфейс командной строки (то есть `plan`, `apply` и т. д.), анализатор и интерпретатор кода Terraform (HCL), средство построения графа зависимостей из ресурсов и источников данных, логика чтения и записи файлов состояния и т. д. Исходный код написан на языке Go и хранится в открытом виде в репозитории GitHub (<https://github.com/hashicorp/terraform>), принадлежащем и поддерживаемом компанией HashiCorp.
- *Провайдеры*. Это *плагины* для ядра Terraform. Каждый плагин написан на Go и реализует определенный интерфейс. Ядро Terraform знает, как установить и запустить плагин. Каждый из этих плагинов предназначен для работы с какой-либо платформой во внешнем мире, например AWS, Azure или Google Cloud. Ядро Terraform взаимодействует с плагинами через *вызовы удаленных процедур* (*Remote Procedure Calls, RPC*), а плагины, в свою очередь, — с соответствующими платформами через сеть (например, используя HTTP-вызовы), как показано на рис. 7.1.

Код каждого плагина обычно хранится в собственном репозитории. Например, все функции AWS, которые до сих пор использовались в книге, находятся в плагине под названием Terraform AWS Provider (или просто AWS Provider), который хранится в собственном репозитории (<https://github.com/hashicorp/terraform->

provider-aws). Первоначально большинство провайдеров было реализовано компанией HashiCorp, и до сих пор она помогает поддерживать многие из них. Но в настоящее время основную работу по развитию каждого провайдера взяли на себя компании, владеющие базовыми платформами: например, сотрудники AWS работают над провайдером AWS, сотрудники Microsoft работают над провайдером Azure, сотрудники Google — над провайдером Google Cloud и т. д.

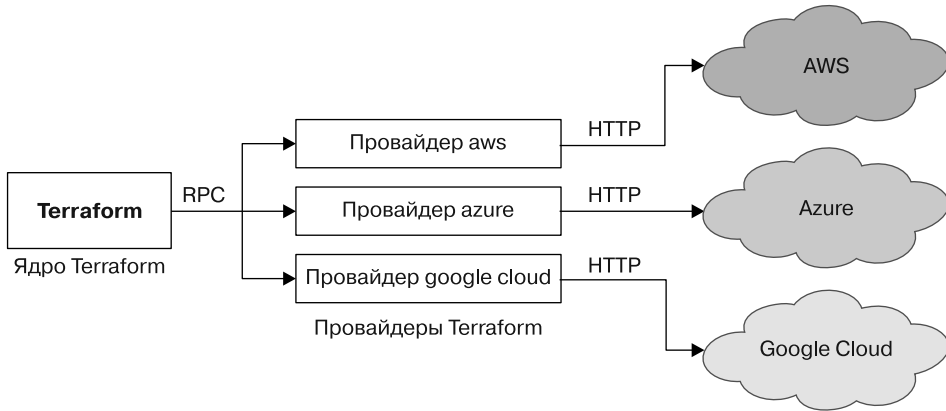


Рис. 7.1. Взаимодействия между ядром Terraform, провайдерами и внешним миром

Каждый провайдер использует свой префикс и предоставляет один или несколько ресурсов и источников данных, имена которых начинаются с этого префикса. Например, все ресурсы и источники данных от провайдера AWS имеют имена, начинающиеся с префикса `aws_` (`aws_instance`, `aws_autoscaling_group`, `aws_ami`), ресурсы и источники данных от провайдера Azure имеют имена, начинающиеся с префикса `azurerm_` (`azurerm_virtual_machine`, `azurerm_virtual_machine_scale_set`, `azurerm_image`) и т. д.

Как устанавливаются провайдеры

Для провайдеров, официально поддерживаемых в Terraform, таких как AWS, Azure и Google, Cloud, достаточно просто добавить в код блок `provider`¹:

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

¹ На самом деле можно вообще опустить блок `provider` и просто добавить любой ресурс или источник данных официально поддерживаемого провайдера, и Terraform по префиксу определит, какой провайдер использовать. Например, если добавить ресурс `aws_instance`, то по префиксу `aws_` Terraform определит, что нужен провайдер AWS.

После запуска `terraform init` Terraform автоматически загрузит код провайдера:

```
$ terraform init
```

```
Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.19.0"...
- Installing hashicorp/aws v4.19.0...
- Installed hashicorp/aws v4.19.0 (signed by HashiCorp)
```

Немного похоже на волшебство, верно? Как Terraform узнает, какой провайдер вам нужен? Или какую версию вы хотите получить? Или откуда его скачать? При обучении и исследовании можно положиться на подобное волшебство, но при разработке и сопровождении промышленного кода вам наверняка потребуются больший контроль над установкой провайдеров в Terraform. Организовать такой контроль можно с помощью блока `require_providers`:

```
terraform {
  required_providers {
    <LOCAL_NAME> = {
      source      = "<URL>"
      version     = "<VERSION>"
    }
  }
}
```

Рассмотрим элементы этого синтаксиса.

- *LOCAL_NAME* — локальное имя провайдера, которое будет использоваться в этом модуле. Каждому провайдеру нужно присвоить уникальное имя и указать его в блоке `provider` с настройками провайдера. Почти всегда *в качестве локального имени предпочтительнее использовать фактическое имя провайдера*: например, для поставщика AWS лучше указывать локальное имя `aws`, поэтому блок `provider` записывается как `provider "aws" { ... }`. Однако в редких случаях у вас может оказаться два провайдера с одинаковыми предпочтительными локальными именами (например, два провайдера, обрабатывающие HTTP-запросы и имеющие предпочтительное локальное имя `http`). В таких случаях можно использовать локальные имена для устранения неоднозначности между ними.
- *URL* — адрес, откуда Terraform должен загрузить провайдер, в формате `[<HOSTNAME>/]<NAMESPACE>/<TYPE>`, где `HOSTNAME` — имя хоста реестра Terraform, через который распространяется провайдер, `NAMESPACE` — пространство имен организации (обычно название компании), а `TYPE` — имя платформы, которой управляет этот провайдер (обычно `TYPE` — это предпочтительное ло-

кальное имя). Например, полный URL провайдера AWS, размещенного в общедоступном реестре Terraform (<https://registry.terraform.io/>), имеет вид `registry.terraform.io/hashicorp/aws`. Однако обратите внимание, что часть `HOSTNAME` необязательна, и если ее опустить, то Terraform по умолчанию загрузит провайдер из общедоступного реестра Terraform, поэтому предыдущий URL провайдера AWS можно записать в более компактной форме: `hashicorp/aws`. Обычно часть `HOSTNAME` включается в URL только для нестандартных провайдеров, загружаемых из частных реестров Terraform (например, из реестра, который вы используете в Terraform Cloud или Terraform Enterprise).

- **VERSION** — ограничение версии. При желании можно потребовать установить конкретную версию, указав точный номер, например `4.19.0`, а можно разрешить установку версии из некоторого диапазона, например `> 4.0, < 4.3`. Подробнее об управлении версиями мы поговорим в главе 8.

Скажем, вот как можно установить версию 4.x провайдера AWS:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Теперь вам должно быть понятно «волшебство» установки провайдера, которое вы видели ранее. Если вы добавите в свой код новый блок `provider` с именем провайдера `foo` и не укажете блок `require_providers`, то при запуске команды `terraform init` Terraform автоматически выполнит следующие шаги:

- попытается загрузить провайдера `foo`, предполагая, что `HOSTNAME` — это общедоступный реестр Terraform, а `NAMESPACE` — это `hashicorp`, соответственно, URL для загрузки будет иметь вид: `registry.terraform.io/hashicorp/foo`;
- если это действительный URL, то Terraform установит последнюю доступную версию провайдера `foo`.

Чтобы установить провайдер, не входящий в пространство имен `hashicorp` (например, провайдер `DataDog`, `CloudFlare`, `Confluent` или специализированный провайдер, созданный вами), или потребовать установить конкретную версию, вам придется добавить блок `require_providers`.



Всегда включайте блок `required_providers`

Как вы узнаете в главе 8, часто важно контролировать версию используемого провайдера, поэтому я рекомендую всегда включать в свой код блок `require_providers`.

Как используются провайдеры

Получив новые знания о провайдерах, вернемся к их использованию. Первый шаг — добавить в код блок `require_providers`, чтобы указать, какой провайдер должен использоваться:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Затем нужно добавить блок `provider` с настройками этого провайдера:

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

До настоящего момента вы настраивали в провайдере AWS только регион, но, помимо региона, есть множество других параметров. Всегда сверяйтесь с документацией для вашего провайдера, чтобы получить подробную информацию: обычно документация находится в том же реестре, который используется для загрузки провайдера (указан в URL в параметре `source`). Например, документация провайдера AWS находится в общедоступном реестре Terraform (<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>). В документации обычно объясняется, как настроить провайдер для работы с разными пользователями, ролями, регионами, учетными записями и т. д.

После настройки провайдера все его ресурсы и источники данных (то есть все с одинаковым префиксом), которые вы помещаете в свой код, будут автоматически использовать эту конфигурацию. Например, после настройки региона в провайдере `aws` на `us-east-2` все ресурсы `aws_` в вашем коде будут автоматически развернуты в `us-east-2`.

Но как быть, если понадобится развернуть одни ресурсы в регионе `us-east-2`, а другие — в другом регионе, например `us-west-1`? Или некоторые ресурсы должны быть развернуты в совершенно другой учетной записи AWS? Для этого вам придется научиться настраивать несколько копий одного и того же провайдера, как описано в следующем разделе.

Работа с несколькими копиями одного провайдера

Чтобы понять, как работать с несколькими копиями одного и того же провайдера, рассмотрим несколько распространенных случаев, когда возникает такая необходимость:

- работу с несколькими регионами AWS;
- работу с несколькими учетными записями AWS;
- создание модулей, способных работать с несколькими провайдерами.

Работа с несколькими регионами AWS

Большинство поставщиков облачных услуг позволяют развертывать ПО в центрах обработки данных (регионах) по всему миру, но, настраивая провайдер Terraform, вы обычно настраиваете его для развертывания только в одном из возможных регионов. Например, до сих пор вы выполняли развертывание только в одном регионе AWS — `us-east-2`:

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

Но как быть, если необходимо выполнить развертывание в нескольких регионах? Например, как развернуть одни ресурсы в `us-east-2`, а другие — в `us-west-1`? У вас может возникнуть соблазн решить эту проблему, определив два блока `provider`, по одному для каждого региона:

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

```
provider "aws" {  
  region = "us-west-1"  
}
```

Но тогда возникает новая проблема: как указать, какую из этих конфигураций должен использовать каждый ресурс, источник данных и модуль? Для начала рассмотрим источники данных. Представьте, что у вас есть две копии источника данных `aws_region`, который возвращает текущий регион AWS:

```
data "aws_region" "region_1" {  
}
```

```
data "aws_region" "region_2" {  
}
```

Как заставить источник данных `region_1` использовать провайдер `us-east-2`, а источник данных `region_2` — провайдер `us-west-1`? Задача решается просто: нужно добавить псевдоним каждому провайдеру:

```
provider "aws" {  
  region = "us-east-2"  
  alias   = "region_1"  
}
```

```
provider "aws" {  
  region = "us-west-1"  
  alias   = "region_2"  
}
```

Псевдоним — это собственное имя блока `provider`, которое можно явно передавать ресурсам, источникам данных и модулям, чтобы они использовали конфигурацию конкретного провайдера. Чтобы потребовать от источников данных `aws_region` использовать определенный провайдер, в них нужно задать параметр `provider`:

```
data "aws_region" "region_1" {  
  provider = aws.region_1  
}
```

```
data "aws_region" "region_2" {  
  provider = aws.region_2  
}
```

Добавьте выходные переменные для проверки:

```
output "region_1" {  
  value       = data.aws_region.region_1.name  
  description = "The name of the first region"  
}
```

```
output "region_2" {  
  value       = data.aws_region.region_2.name  
  description = "The name of the second region"  
}
```

И запустите команду `apply`:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
region_1 = "us-east-2"  
region_2 = "us-west-1"
```

Теперь каждый источник данных `aws_region` использует свой провайдер и, следовательно, работает в своем регионе AWS. Тот же метод с установкой параметра `provider` работает и с ресурсами. Например, вот как можно развернуть два сервера EC2 в разных регионах:

```
resource "aws_instance" "region_1" {
  provider = aws.region_1

  # Обратите внимание на отличающийся AMI ID!!
  ami      = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

resource "aws_instance" "region_2" {
  provider = aws.region_2

  # Обратите внимание на отличающийся AMI ID!!
  ami      = "ami-01f87c43e618bf8f0"
  instance_type = "t2.micro"
}
```

Обратите внимание, что в каждом ресурсе `aws_instance` присутствует параметр `provider`, гарантирующий его развертывание в нужном регионе. Также обратите внимание, что параметры `ami` должны быть разными для двух ресурсов `aws_instance`: это связано с тем, что идентификаторы AMI уникальны для всех регионов AWS, поэтому идентификатор образа с Ubuntu 20.04 в `us-east-2` отличается от идентификатора с Ubuntu 20.04 в `us-west-1`. Поиск этих идентификаторов AMI и управление ими вручную утомительны и подвержены ошибкам. К счастью, есть более удобная альтернатива: используйте источник данных `aws_ami`, который благодаря набору фильтров может автоматически отыскивать подходящие идентификаторы AMI. Вот как можно использовать этот источник данных дважды, по одному разу в каждом регионе, чтобы отыскать идентификаторы AMI с Ubuntu 20.04:

```
data "aws_ami" "ubuntu_region_1" {
  provider = aws.region_1

  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

data "aws_ami" "ubuntu_region_2" {
  provider = aws.region_2
```

```

most_recent = true
owners      = ["099720109477"] # Canonical

filter {
  name     = "name"
  values   = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
}
}

```

Отметьте также, как в каждом источнике данных определен параметр `provider`, гарантирующий поиск идентификатора АМІ в нужном регионе. Вернитесь к коду `aws_instance` и обновите параметр `ami`, добавив значения, возвращаемые этими источниками данных:

```

resource "aws_instance" "region_1" {
  provider = aws.region_1

  ami          = data.aws_ami.ubuntu_region_1.id
  instance_type = "t2.micro"
}

resource "aws_instance" "region_2" {
  provider = aws.region_2

  ami          = data.aws_ami.ubuntu_region_2.id
  instance_type = "t2.micro"
}

```

Так намного лучше. Теперь независимо от региона, в котором выполняется развертывание, вы автоматически получите правильный идентификатор АМІ образа с Ubuntu. Чтобы убедиться, что эти серверы ЕС2 действительно развертываются в разных регионах, добавьте выходные переменные, возвращающие зоны доступности (каждая такая зона находится в одном регионе), где фактически развернут каждый сервер:

```

output "instance_region_1_az" {
  value     = aws_instance.region_1.availability_zone
  description = "The AZ where the instance in the first region deployed"
}

output "instance_region_2_az" {
  value     = aws_instance.region_2.availability_zone
  description = "The AZ where the instance in the second region deployed"
}

```

И выполните `apply`:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
instance_region_1_az = "us-east-2a"
instance_region_2_az = "us-west-1b"
```

Итак, теперь вы знаете, как разворачивать источники данных и ресурсы в разных регионах. А что насчет модулей? Например, в главе 3 вы использовали Amazon RDS для разворачивания одного экземпляра базы данных MySQL в тестовой среде (stage/data-stores/mysql):

```
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"
  skip_final_snapshot = true

  username = var.db_username
  password = var.db_password
}
```

Такой вариант хорошо подходит для тестовой среды, но в промышленной среде единственная база данных образует единую точку отказа. К счастью, Amazon RDS изначально поддерживает *репликацию* и автоматически копирует данные из основной базы данных в дополнительную — реплику, доступную только для чтения, что может пригодиться для масштабируемости и в качестве резерва на случай выхода из строя основной базы данных. Вы даже можете запустить реплику в совершенно другом регионе AWS, поэтому, если один регион выйдет из строя (например, произойдет серьезный сбой в регионе `us-east-2`), вы сможете переключиться на другой (например, `us-west-1`).

Давайте превратим код разворачивания MySQL в тестовой среде в многократно используемый модуль `mysql`, поддерживающий репликацию. Сначала скопируйте все содержимое `stage/data-stores/mysql`, включая `main.tf`, `variables.tf` и `outputs.tf`, в новую папку `modules/data-stores/mysql`. Затем откройте `modules/data-stores/mysql/variables.tf` и объявите две новые переменные:

```
variable "backup_retention_period" {
  description = "Days to retain backups. Must be > 0 to enable replication."
  type       = number
  default    = null
}

variable "replicate_source_db" {
  description = "If specified, replicate the RDS database at the given ARN."
```

```

    type      = string
    default   = null
}

```

Как вы увидите далее, переменную `backup_retention_period` мы используем в базе данных, играющей роль источника, чтобы включить репликацию, а переменную `replication_source_db` — в базе данных, играющей роль получателя, чтобы превратить ее в реплику. Откройте `modules/data-stores/mysql/main.tf` и обновите ресурс `aws_db_instance`, как описано ниже.

1. Передайте значения переменных `backup_retention_period` и `replication_source_db` в одноименные параметры ресурса `aws_db_instance`.
2. В экземпляре базы данных, представляющем реплику, AWS не позволит вам установить параметры `engine`, `db_name`, `username` и `password`, потому что все они наследуются от основного экземпляра. Поэтому необходимо добавить в ресурс `aws_db_instance` некоторую условную логику, чтобы не устанавливать эти параметры, когда установлена переменная `replication_source_db`.

Вот как должен выглядеть ресурс после внесения перечисленных изменений:

```

resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  allocated_storage = 10
  instance_class    = "db.t2.micro"
  skip_final_snapshot = true

  # Включить резервное копирование
  backup_retention_period = var.backup_retention_period

  # Если эта переменная задана, то БД играет роль реплики
  replicate_source_db = var.replicate_source_db

  # Эти параметры устанавливаются, только если
  # переменная replicate_source_db имеет непустое значение
  engine      = var.replicate_source_db == null ? "mysql" : null
  db_name     = var.replicate_source_db == null ? var.db_name : null
  username    = var.replicate_source_db == null ? var.db_username : null
  password    = var.replicate_source_db == null ? var.db_password : null
}

```

Обратите внимание, что в реплике входные переменные `db_name`, `db_username` и `db_password` должны быть необязательными, поэтому вернитесь к `modules/data-stores/mysql/variables.tf` и установите для них `null` как значение по умолчанию:

```

variable "db_name" {
  description = "Name for the DB."
  type        = string
  default     = null
}

```

```
variable "db_username" {
  description = "Username for the DB."
  type        = string
  sensitive    = true
  default      = null
}
```

```
variable "db_password" {
  description = "Password for the DB."
  type        = string
  sensitive    = true
  default      = null
}
```

Чтобы использовать переменную `replication_source_db`, необходимо установить для нее ARN другой базы данных RDS, поэтому в `modules/data-stores/mysql/outputs.tf` также следует добавить выходную переменную с ARN базы данных:

```
output "arn" {
  value        = aws_db_instance.example.arn
  description = "The ARN of the database"
}
```

И еще: добавьте в этот модуль блок `require_providers`, где укажите, что он предполагает использование провайдера AWS, а также номер ожидаемой версии провайдера:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Вскоре вы поймете, почему это важно и при работе с несколькими регионами!

Теперь этот модуль `mysql` можно использовать в промышленной среде для развертывания первичной БД MySQL и ее реплики. Сначала создайте `live/prod/data-stores/mysql/variables.tf` и объявите входные переменные для передачи имени пользователя и пароля базы данных (чтобы получить возможность передавать эти секреты через переменные окружения, как описано в главе 6):

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive    = true
}
```

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

Затем создайте `live/prod/data-stores/mysql/main.tf` и используйте модуль `mysql` для настройки основной БД:

```
module "mysql_primary" {
  source = "../../../../../modules/data-stores/mysql"

  db_name      = "prod_db"
  db_username  = var.db_username
  db_password  = var.db_password

  # Должна иметь непустое значение для поддержки репликации
  backup_retention_period = 1
}
```

Теперь добавьте второй вызов модуля `mysql` для создания реплики:

```
module "mysql_replica" {
  source = "../../../../../modules/data-stores/mysql"

  # Сделать этот экземпляр репликой основной БД
  replicate_source_db = module.mysql_primary.arn
}
```

Красиво и компактно! Теперь осталось только передать ARN основной базы данных в параметр `replication_source_db`, который должен запустить базу данных RDS в качестве реплики.

Есть только одна проблема: как указать коду, что он должен развернуть основную БД и реплику в разных регионах? Для этого создайте два блока `provider`, каждый со своим псевдонимом:

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

provider "aws" {
  region = "us-west-1"
  alias  = "replica"
}
```

Чтобы указать модулю, какой провайдер должен использоваться, установите параметр `provider`. Вот как настраивается использование провайдера `primary` (того, что в `us-east-2`) в основной БД MySQL:


```
module "mysql_primary" {
  source = "../../../modules/data-stores/mysql"

  providers = {
    aws = aws.primary
  }

  db_name      = "prod_db"
  db_username  = var.db_username
  db_password  = var.db_password

  # Должна иметь непустое значение для поддержки репликации
  backup_retention_period = 1
}
```

А так настраивается использование провайдера `replica` (того, что в `us-west-1`) в реплике MySQL:

```
module "mysql_replica" {
  source = "../../../modules/data-stores/mysql"

  providers = {
    aws = aws.replica
  }

  # Сделать этот экземпляр репликой основной БД
  replicate_source_db = module.mysql_primary.arn
}
```

В модулях параметр `providers` (обратите внимание на множественное число) определяет ассоциативный массив, тогда как в ресурсах и источниках данных параметр `provider` (единственное число) определяет единственное значение. Это связано с тем, что каждый ресурс и каждый источник данных развертывается только в одном провайдере, но модуль может содержать несколько источников данных и ресурсов и использовать несколько провайдеров (пример модуля с несколькими провайдерами вы увидите ниже). В ассоциативном массиве `providers`, который передается в модуль, ключ должен соответствовать локальному имени провайдера в ассоциативном массиве `require_providers` внутри модуля (в данном случае в обоих установлено значение `aws`). Это еще одна причина, почему желательно явно определять `required_providers` практически в каждом модуле.

Последний шаг — создать `live/prod/data-stores/mysql/outputs.tf` со следующими выходными переменными:

```
output "primary_address" {
  value      = module.mysql_primary.address
  description = "Connect to the primary database at this endpoint"
}
```

```

output "primary_port" {
  value      = module.mysql_primary.port
  description = "The port the primary database is listening on"
}

output "primary_arn" {
  value      = module.mysql_primary.arn
  description = "The ARN of the primary database"
}

output "replica_address" {
  value      = module.mysql_replica.address
  description = "Connect to the replica database at this endpoint"
}

output "replica_port" {
  value      = module.mysql_replica.port
  description = "The port the replica database is listening on"
}

output "replica_arn" {
  value      = module.mysql_replica.arn
  description = "The ARN of the replica database"
}

```

И теперь наконец вы готовы к развертыванию! Обратите внимание, что запуск `apply` для развертывания основной БД и ее реплики может занять много времени, около 20–30 минут, поэтому наберитесь терпения.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```

primary_address = "terraform-up-and-running.cmyd6qwb.us-east-2.rds.amazonaws.com"
primary_arn     = "arn:aws:rds:us-east-2:111111111111:db:terraform-up-and-running"
primary_port    = 3306
replica_address = "terraform-up-and-running.drctpdoe.us-west-1.rds.amazonaws.com"
replica_arn     = "arn:aws:rds:us-west-1:111111111111:db:terraform-up-and-running"
replica_port    = 3306

```

И вот она, репликация между регионами! Войдите в консоль RDS, чтобы убедиться, что репликация работает (рис. 7.2).

В качестве самостоятельного упражнения я оставляю вам возможность изменить тестовую среду (`stage/data-stores/mysql`) так, чтобы она тоже использовала модуль `mysql` (`modules/data-stores/mysql`), но настраивала развертывание БД без репликации, поскольку в тестовых средах столь высокий уровень доступности обычно не требуется.

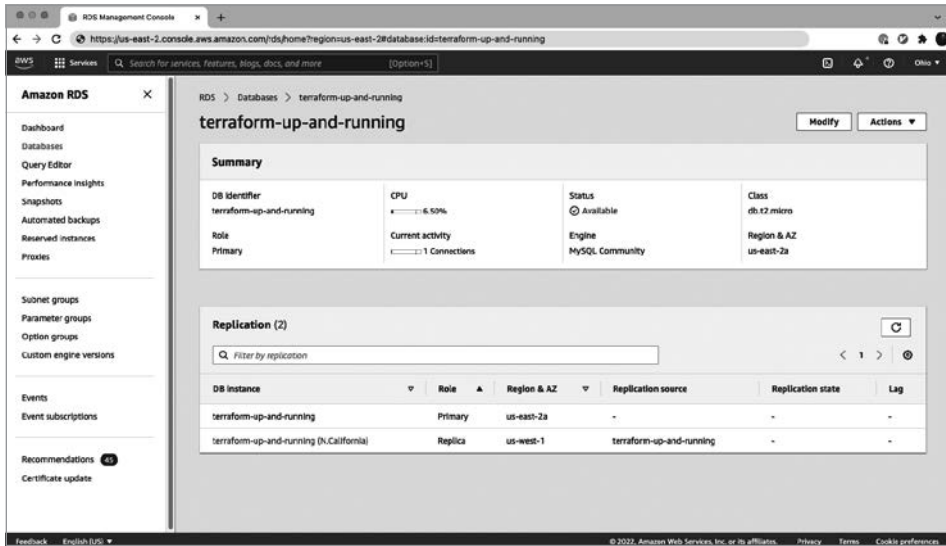


Рис. 7.2. Как можно видеть в консоли RDS, первичный сервер находится в us-east-2, а реплика — в us-west-1

Как можно заметить в этих примерах, используя несколько провайдеров с псевдонимами, с помощью Terraform легко развернуть ресурсы в нескольких регионах. Однако, прежде чем двигаться дальше, я хочу сделать два предупреждения.

- *Предупреждение 1: работать в нескольких регионах сложно.* Чтобы запустить инфраструктуру в нескольких регионах по всему миру, особенно в «активно-активном» режиме, когда несколько регионов одновременно активно отвечают на запросы пользователей (в отличие от случая, когда один регион является резервным), необходимо решить множество сложных задач, таких как обработка задержек между регионами; выбор между одним регионом, доступным для записи (что означает низкую доступность и большую задержку), или несколькими (что означает необходимость принять согласованность в конечном итоге или предусмотреть сегментирование); выяснение, как генерировать уникальные идентификаторы (стандартного автоинкремента идентификатора, поддерживаемого большинством баз данных, уже недостаточно); работа над соблюдением местного законодательства в отношении хранения данных и т. д. Обсуждение всех этих проблем выходит за рамки книги, но я решил, что должен хотя бы упомянуть их, чтобы прояснить, что развертывание в нескольких регионах в реальном мире — это не просто вопрос добавления нескольких псевдонимов провайдеров в код Terraform!
- *Предупреждение 2: используйте псевдонимы с осторожностью.* В Terraform легко использовать псевдонимы, но я бы предостерег от слишком частого

их употребления, *особенно* при настройке многорегиональной инфраструктуры. Одна из основных причин создания многорегиональной инфраструктуры заключается в возможности обеспечить отказоустойчивость: например, если `us-east-2` выйдет из строя, то ваша инфраструктура в `us-west-1` сможет продолжать работать. Но если для развертывания в обоих регионах вы используете один модуль Terraform с псевдонимами, то, когда один из этих регионов окажется недоступен, модуль не сможет подключиться к этому региону и любая попытка запустить `plan` или `apply` завершится неудачей. Поэтому, когда вам понадобится применить изменения и одновременно с этим произойдет серьезный сбой, ваш код Terraform перестанет работать.

В общем случае, как обсуждалось в главе 3, желательно полностью изолировать среды: вместо управления несколькими регионами в одном модуле с помощью псевдонимов лучше управлять каждым регионом в отдельных модулях. Такой подход поможет минимизировать область влияния ваших собственных ошибок (например, если вы случайно сломаете что-то в одном регионе, это с меньшей вероятностью повлияет на другой) и проблем в окружающем мире (например, отключение электроэнергии в одном регионе с меньшей вероятностью повлияет на другой регион).

Так когда имеет смысл использовать псевдонимы? Обычно они хорошо подходят для случаев, когда инфраструктуры, развертываемые с несколькими псевдонимами провайдеров, действительно связаны между собой и должны всегда развертываться вместе. Например, если в качестве CDN (Content Distribution Network — сеть распространения контента) вы хотите использовать Amazon CloudFront и предоставить для нее сертификат TLS с помощью AWS Certification Manager (ACM), тогда для создания сертификата AWS вам потребуется использовать регион `us-east-1`, независимо от того, какие еще регионы прописаны для самого CloudFront. В этом случае ваш код может иметь два блока `provider`: один для основного региона, в котором развертывается CloudFront, и один с жестко запрограммированным псевдонимом для `us-east-1`, в котором настраивается сертификат TLS. Другой вариант использования псевдонимов — для развертывания ресурсов, предназначенных для многих регионов: например, AWS рекомендует развернуть службу автоматического обнаружения угроз GuardDuty в каждом отдельном регионе, который указан в учетной записи AWS. В этом случае может иметь смысл создать модуль с блоком `provider` и отдельными псевдонимами для каждого региона AWS.

За исключением нескольких пограничных случаев, псевдонимы для обработки нескольких регионов используются довольно редко. Более распространенный вариант их применения — когда есть несколько провайдеров, в которых необходимо проходить аутентификацию разными способами, например, когда каждый из них выполняет аутентификацию в разных учетных записях AWS.

Работа с несколькими учетными записями AWS

До сих пор в книге мы использовали одну учетную запись AWS для всей инфраструктуры. Однако на практике часто используется несколько учетных записей AWS: например, тестовая среда разворачивается в тестовой учетной записи, промышленная среда — в промышленной и т. д. Эта концепция применима и к другим облачным окружениям, таким как Azure и Google Cloud. Обратите внимание, что в книге я использую термин «учетная запись», тогда как в некоторых облаках для тех же понятий употребляются разные термины (например, в Google Cloud они называются *проектами*, а не учетными записями).

Перечислю основные причины работы с несколькими учетными записями.

- *Изоляция (или разделение)*. Разные учетные записи используются, чтобы изолировать разные среды друг от друга и ограничить «радиус взрыва», когда что-то пойдет не так. Например, размещение тестовой и промышленной сред в отдельных учетных записях гарантирует, что, проникнув в тестовую среду, злоумышленник не сможет получить доступ к промышленной среде. Аналогично изоляция гарантирует, что разработчик, вносящий изменения в тестовую среду, с меньшей вероятностью нарушит что-то в промышленной среде.
- *Аутентификация и авторизация*. Когда все находится в одной учетной записи, сложно разграничить доступ (например, предоставить доступ к тестовой среде, но не дать по ошибке доступа к промышленной). Использование нескольких учетных записей упрощает такой контроль, поскольку любые разрешения, которые предоставляются в одной учетной записи, не влияют ни на какую другую.

Необходимость аутентификации в нескольких учетных записях также помогает снизить вероятность ошибок. Имея все в одной учетной записи, слишком легко допустить ошибку, думая, что изменения вносятся, скажем, в тестовую среду, а на самом деле они вносятся в промышленную среду (что может закончиться катастрофой, если изменение предполагает удаление всех таблиц в базе данных). При использовании нескольких учетных записей такое менее вероятно, потому что для аутентификации в каждой учетной записи требуется выполнить отдельную последовательность шагов.

Обратите внимание: наличие нескольких учетных записей *не* означает, что у разработчиков имеется несколько отдельных профилей пользователей (например, отдельных пользователей IAM в каждой учетной записи AWS). Фактически это было бы антипаттерном, потому что потребовало бы управлять несколькими наборами учетных данных, разрешений и т. д. Вместо этого практически во всех облаках для каждого разработчика можно создать ровно один профиль, который он мог бы использовать для аутентификации в любой доступной ему учетной записи. Механизм аутентификации между учетными записями различается в разных облаках: например, в AWS можно

выполнять аутентификацию между учетными записями AWS, присваивая роли IAM, как будет показано ниже.

- *Аудит и отчетность.* Правильно настроенная структура учетной записи позволит отслеживать все изменения во всех ваших средах, проверять соблюдение требований соответствия и выявлять аномалии. Более того, можно даже иметь единый счет, отражающий все расходы по всем вашим учетным записям в одном месте, включая разбивку затрат по учетным записям, сервисам, тегам и т. д. Это особенно полезно в крупных организациях, поскольку позволяет финансировать, контролировать и утверждать бюджет расходов команды, просто просматривая, из какой учетной записи поступают платежи.

Рассмотрим пример с несколькими учетными записями в AWS. Сначала создайте новую учетную запись AWS для тестирования. У вас уже есть одна учетная запись AWS, поэтому для создания новых *дочерних учетных записей* вы можете использовать AWS Organizations, чтобы гарантировать объединение счетов из всех дочерних учетных записей с родительской (ее иногда еще называют *корневой*) и получить доступ к информационной панели, которую можно использовать для управления всеми дочерними учетными записями.

Перейдите на страницу AWS Organizations Console (<https://console.aws.amazon.com/organizations/v2/home/accounts>) и нажмите кнопку Add an AWS account (Добавить учетную запись AWS) (рис. 7.3).

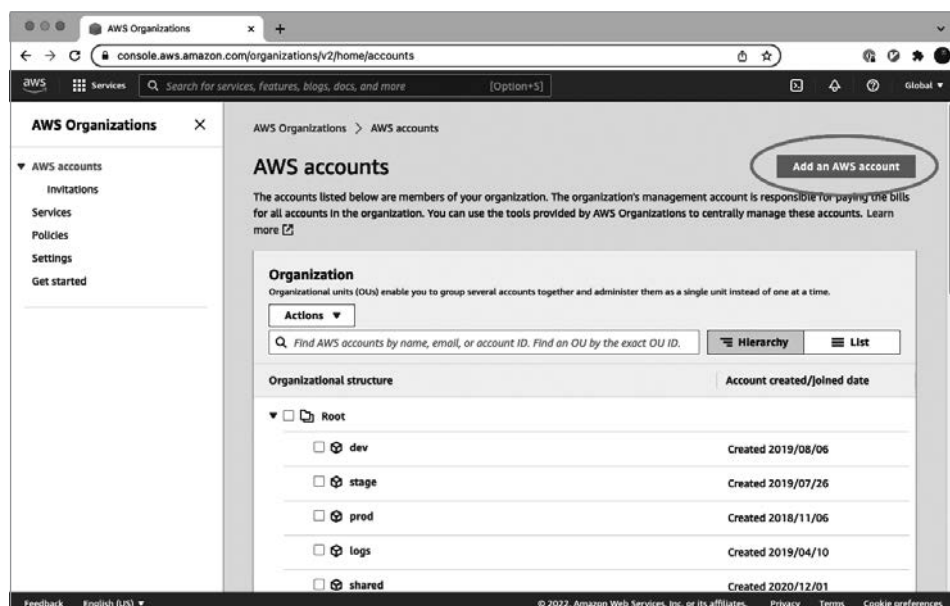


Рис. 7.3. Создание новой учетной записи AWS в AWS Organizations

Далее заполните следующие поля (рис. 7.4).

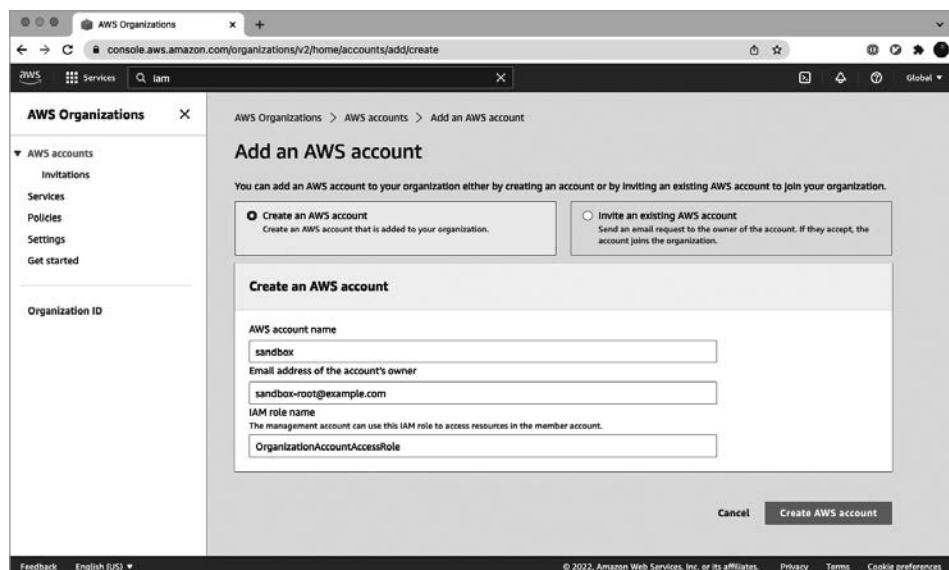


Рис. 7.4. Внесите информацию о новой учетной записи AWS

- **AWS account name** (Имя учетной записи AWS) — имя, которое будет использоваться для учетной записи. Например, если эта учетная запись предназначена для развертывания тестовой среды, то ей можно дать имя **staging**.
- **Email address of the account's owner** (Адрес электронной почты владельца учетной записи) — адрес электронной почты пользователя **root** учетной записи AWS. Обратите внимание, что для каждой учетной записи AWS должен указываться свой уникальный адрес электронной почты пользователя **root**, поэтому вы не сможете повторно прописать адрес электронной почты, указанный при создании первой (корневой) учетной записи AWS (см. врезку «Как получить несколько псевдонимов для одного адреса электронной почты» ниже, где описывается возможное обходное решение). А как насчет пароля пользователя **root**? По умолчанию AWS не требует устанавливать пароль пользователя **root** для дочерней учетной записи (вскоре вы увидите альтернативный способ аутентификации в дочерней учетной записи). Если вам когда-нибудь может понадобиться выполнить вход с этой учетной записью, то после ее создания нужно пройти процедуру сброса пароля, используя адрес электронной почты, указанный здесь.
- **IAM role name** (Имя роли IAM) — при создании дочерней учетной записи AWS AWS Organizations автоматически создает в ней роль IAM с правами

администратора и появляется возможность присвоить роли из родительской учетной записи. Это удобно, потому что можно выполнять аутентификацию в дочерней учетной записи AWS, не создавая пользователей IAM или роли IAM. Я рекомендую оставить в этом поле значение по умолчанию — `OrganizationAccountAccessRole`.

КАК ПОЛУЧИТЬ НЕСКОЛЬКО ПСЕВДОНИМОВ ДЛЯ ОДНОГО АДРЕСА ЭЛЕКТРОННОЙ ПОЧТЫ

Пользователи Gmail могут получить несколько псевдонимов для одного адреса электронной почты, воспользовавшись тем фактом, что Gmail игнорирует все, что стоит после знака плюс в адресе. Например, имея адрес Gmail `example@gmail.com`, вы можете отправить электронное письмо на адреса `example+foo@gmail.com` и `example+any-text-you-want@gmail.com`, и все электронные письма будут отправлены на адрес `example@gmail.com`. Этот прием также подойдет, даже если ваша компания использует Gmail через Google Workspace с собственным доменом: например, `example+dev@company.com` и `example+stage@company.com` будут интерпретироваться как `example@company.com`.

Эта особенность может пригодиться, когда понадобится создать десяток дочерних учетных записей AWS, поскольку вместо создания десятка отдельных адресов электронной почты можно использовать `example+dev@company.com` для учетной записи разработчика, `example+stage@company.com` для учетной записи тестировщика и т. д.; AWS будет видеть каждый из этих адресов электронной почты как отдельный уникальный адрес, но в действительности все электронные письма будут отправляться на один и тот же адрес.

Нажмите кнопку **Create AWS Account** (Создать учетную запись AWS), подождите несколько минут, пока AWS создаст учетную запись, а затем запишите 12-значный идентификатор созданной учетной записи. Далее в главе мы будем предполагать следующее:

- идентификатор родительской учетной записи AWS: 111111111111;
- идентификатор дочерней учетной записи AWS: 222222222222.

Пройти аутентификацию в своей новой дочерней учетной записи можно в консоли AWS, щелкнув на своем имени пользователя и нажав кнопку **Switch role** (Сменить роль) (рис. 7.5).

Затем введите данные о роли IAM, которую вы хотите себе присвоить (рис. 7.6).

- **Account** (Учетная запись) — 12-значный идентификатор учетной записи AWS, на которую необходимо переключиться. Вы должны ввести идентификатор новой дочерней учетной записи.
- **Role** (Роль) — имя роли IAM, которую следует использовать в этой учетной записи AWS. Введите имя роли IAM, которое вы указывали при создании новой дочерней учетной записи (по умолчанию `OrganizationAccountAccessRole`).

- Display Name (Отображаемое имя) — AWS создаст ярлык на навигационной панели (чтобы вы могли переключиться на эту учетную запись в будущем одним щелчком кнопкой мыши) и будет отображать это имя в ярлыке. Это влияет только на вашего пользователя IAM в данном браузере.

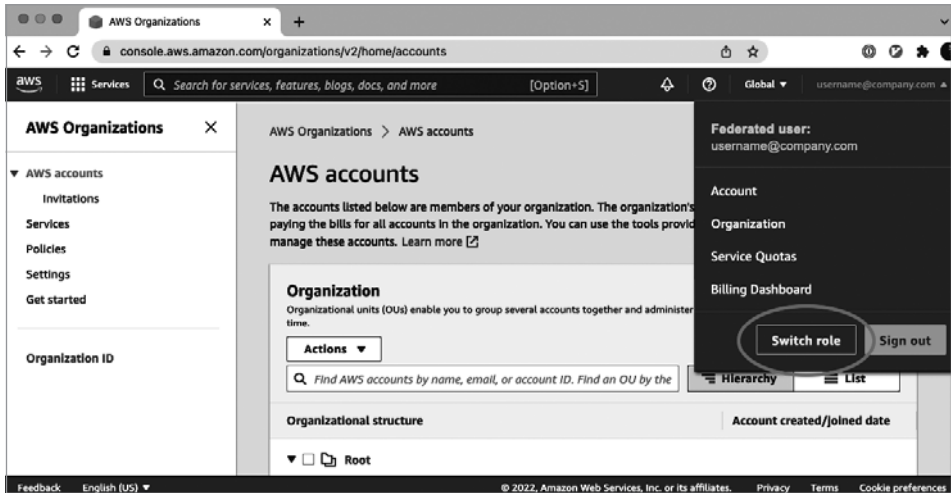


Рис. 7.5. Нажмите кнопку, выбрав Switch Role (Сменить роль)

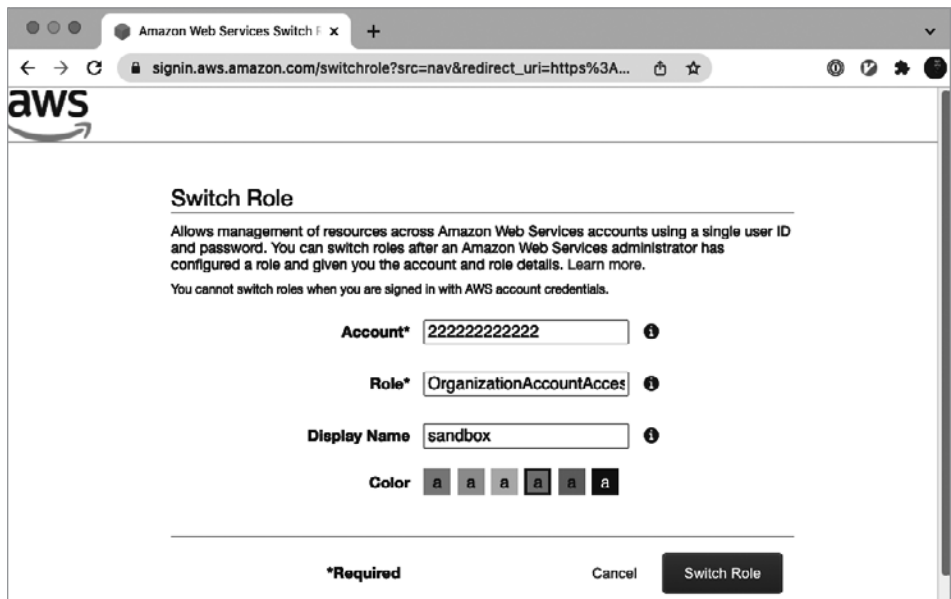


Рис. 7.6. Введите данные о роли IAM, на которую хотите переключиться

Нажмите кнопку **Switch role** (Сменить роль), и после этого в консоли будет выполнен вход в новую учетную запись AWS.

Теперь напишем модуль Terraform в `example/multi-account-root`, который будет выполнять аутентификацию в нескольких учетных записях AWS. Как и в примере использования нескольких регионов AWS, добавьте в `main.tf` два блока `provider`, каждый со своим псевдонимом. Первый блок `provider` предназначен для родительской учетной записи AWS:

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

А второй — для дочерней:

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

Чтобы пройти аутентификацию в дочерней учетной записи AWS, нужно присвоить себе роль IAM. В веб-консоли вы сделали это, нажав кнопку **Switch role** (Сменить роль); в коде Terraform то же самое можно сделать, добавив блок `submit_role` в блок `provider` для дочерней учетной записи:

```
provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::<ID_УЧЕТНОЙ_ЗАПИСИ>:role/<ИМЯ_РОЛИ>"
  }
}
```

В параметре `role_arn` нужно заменить `ID_УЧЕТНОЙ_ЗАПИСИ` идентификатором вашей дочерней учетной записи, а `ИМЯ_РОЛИ` — фактическим именем роли IAM в этой учетной записи, подобно тому как вы это делали при переключении ролей в веб-консоли. Вот как это выглядит в случае с идентификатором учетной записи `22222222222` и именем роли `OrganizationAccountAccessRole`:

```
provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::22222222222:role/OrganizationAccountAccessRole"
  }
}
```

Теперь, чтобы проверить работу модуля, добавьте два источника данных `aws_caller_identity` и настройте каждый на использование разных провайдеров:

```
data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}
```

Наконец, добавьте в файл `outputs.tf` выходные переменные, чтобы вывести идентификаторы учетных записей:

```
output "parent_account_id" {
  value      = data.aws_caller_identity.parent.account_id
  description = "The ID of the parent AWS account"
}

output "child_account_id" {
  value      = data.aws_caller_identity.child.account_id
  description = "The ID of the child AWS account"
}
```

Запустите `apply`, и вы увидите разные идентификаторы учетных записей:

```
$ terraform apply
```

```
(...)
```

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

```
Outputs:
```

```
parent_account_id = "111111111111"
child_account_id  = "222222222222"
```

Вот и все: теперь вы знаете, как, используя псевдонимы провайдеров и блоки `take_role`, писать код Terraform, способный работать с несколькими учетными записями AWS.

По аналогии с использованием нескольких регионов хочу сделать несколько предупреждений.

- *Предупреждение 1. Для использования ролей IAM в нескольких учетных записях требуется двойное согласование.* Чтобы роль IAM разрешала доступ из одной учетной записи AWS к другой (например, чтобы разрешить учетной записи 222222222222 присваивать роль IAM из учетной записи 111111111111), необходимо предоставить разрешения в *обеих* учетных записях AWS.
 - Во-первых, в учетной записи AWS, которая принимает роль IAM (например, в дочерней учетной записи 222222222222), следует настроить

политику передачи роли, оказывающую доверие другой учетной записи AWS (например, родительской учетной записи `111111111111`). В случае с ролью `IAM OrganizationAccountAccessRole` это произошло автоматически, потому что AWS Organizations автоматически настраивает политику присваивания этой роли IAM, подтверждающую доверие к родительской учетной записи. Однако, создавая собственные роли IAM, не забудьте явно предоставить разрешение `sts:AssumeRole`.

- Во-вторых, в учетной записи AWS, от которой принимается роль (например, в родительской учетной записи `111111111111`), тоже следует предоставить пользователю разрешения на присваивание этой роли IAM. И снова в нашем примере это было сделано автоматически, потому что в главе 2 вы дали своему пользователю IAM привилегии администратора, позволяющие вам делать в родительской учетной записи AWS практически все, что угодно, включая присваивание ролей IAM. На практике же в большинстве случаев ваш пользователь не будет (не должен быть!) администратором, поэтому вам придется явно предоставить пользователю разрешения `sts:AssumeRole` для присваивания нужных вам ролей IAM.
- *Предупреждение 2: используйте псевдонимы с осторожностью.* Я уже упоминал об этом в примере с несколькими регионами, но стоит повторить еще раз: несмотря на простоту использования псевдонимов в Terraform, я не рекомендую злоупотреблять ими, в том числе в коде для нескольких учетных записей. Обычно несколько учетных записей нужны, чтобы разделить данные, поэтому, если что-то пойдет не так в одной учетной записи, это не повлияет на другую. Модули, развертываемые в нескольких учетных записях, противостоят этому принципу. Используйте псевдонимы, только если вы хотите, чтобы ресурсы в нескольких учетных записях были связаны и развернуты вместе.

Создание модулей, способных работать с несколькими провайдерами

Модули Terraform, с которыми вы работаете, обычно делятся на две категории.

- *Многоразовые* — низкоуровневые модули, предназначенные не для развертывания напрямую, а для объединения с другими модулями, ресурсами и источниками данных.
- *Корневые* — высокоуровневые модули, объединяющие несколько много-разовых модулей и предназначенные для непосредственного развертывания путем запуска `apply` (фактически любой модуль, к которому применяется команда `apply`, является корневым).

В примерах с несколькими провайдерами, которые вы встречали до сих пор, все блоки `provider` помещались в корневой модуль. Но как быть, если необходимо создать многоразовый модуль, работающий с несколькими провайдерами? Например, если вдруг возникнет потребность превратить код из предыдущего раздела, использующий несколько учетных записей, в многоразовый модуль? В качестве первого шага можно поместить весь этот код без изменений в папку `modules/multi-account`. Затем создать новый пример для его тестирования в папке `example/multi-account-module` с файлом `main.tf`, который содержит следующий код:

```
module "multi_account_example" {  
  source = "../modules/multi-account"  
}
```

Если применить к нему `apply`, он будет работать, но есть проблема: вся конфигурация провайдера теперь скрыта в самом модуле (в `modules/multi-account`). Определение блоков `provider` в многоразовых модулях считается антипаттерном по нескольким причинам.

- *Проблемы с конфигурацией.* Если в вашем многоразовом модуле определены блоки `provider`, то этот модуль управляет всей конфигурацией данного провайдера. Например, в настоящее время ARN роли IAM и используемые регионы жестко запрограммированы в модуле `modules/multi-account`. Конечно, можно объявить входные переменные, чтобы пользователи могли задавать регионы и ARN ролей IAM, но это только верхушка айсберга. Если заглянуть в документацию провайдера AWS, можно обнаружить, что ему можно передать примерно 50 разных конфигурационных параметров! Многие из них будут важны для пользователей вашего модуля, поскольку определяют порядок аутентификации в AWS, используемый регион, используемую учетную запись (или роль IAM), конечные точки для взаимодействия с AWS, применяемые или игнорируемые теги и многое другое. Необходимость передавать 50 дополнительных переменных в модуль сделает его слишком громоздким и неудобным в обслуживании и использовании.
- *Проблемы дублирования.* Даже если вы реализуете в своем модуле поддержку всех 50 параметров или подмножества наиболее важных из них, то создадите предпосылки для дублирования кода пользователями вашего модуля. Это связано с тем, что нередко требуется объединить несколько модулей и в таких случаях пользователям придется передавать подмножество из 50 параметров в каждый из них, чтобы все они прошли аутентификацию, а для этого понадобится скопировать и вставить много данных, что утомительно и чревато ошибками.

- *Проблемы с производительностью.* Каждый раз, встречая блок `provider` в коде, Terraform запускает новый процесс для этого провайдера и взаимодействует с ним через механизм RPC. Если определить несколько блоков `provider`, то ничего страшного не произойдет, но при масштабировании это может вызвать проблемы с производительностью. Вот реальный пример: несколько лет назад я создал многократно модули для CloudTrail, AWS Config, GuardDuty, IAM Access Analyzer и Macie. Предполагалось, что каждый из этих сервисов AWS должен быть развернут в каждом регионе учетной записи AWS, а поскольку у AWS около 25 регионов, я включил 25 блоков `provider` в каждый из этих модулей. Затем я создал один корневой модуль для развертывания всего этого в моих учетных записях AWS: если посчитать, то получится 5 модулей по 25 блоков `provider` в каждом, или всего 125 блоков `provider`. Когда я вызывал команду `apply`, Terraform запускал 125 процессов, каждый из которых выполнял сотни вызовов API и RPC. Пытаясь одновременно выполнить тысячи сетевых запросов, мой процессор перегружался, и выполнение одной команды `plan` могло занять 20 минут. Хуже того, иногда это приводило к перегрузке сетевого стека и периодическим сбоям в вызовах API, а запуск `apply` порождал спорадические ошибки.

Поэтому я рекомендую не определять никаких блоков `provider` в многократно модулях, а просто позволить пользователям создавать нужные им блоки `provider` в их корневых модулях. Но как создать модуль, который сможет работать с несколькими провайдерами? Если в модуле нет блоков `provider`, то как определить псевдонимы провайдеров, на которые можно было бы сослаться в своих ресурсах и источниках данных?

Решение состоит в использовании *псевдонимов конфигураций*. Они очень похожи на псевдонимы провайдеров, которые вы уже видели, но определяются не в блоке `provider`, а в блоке `require_providers`.

Откройте `modules/multi-account/main.tf`, удалите вложенные блоки `provider` и замените их блоком `require_providers` с псевдонимами конфигурации, как показано ниже:

```
terraform {
  required_providers {
    aws = {
      source      = "hashicorp/aws"
      version     = "~> 4.0"
      configuration_aliases = [aws.parent, aws.child]
    }
  }
}
```

Подобно обычным псевдонимам провайдеров, псевдонимы конфигурации можно передавать в ресурсы и источники данных, используя параметр `provider`:

```
data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}
```

Ключевое отличие от обычных псевдонимов провайдеров заключается в том, что псевдонимы конфигурации сами по себе не создают никаких провайдеров, а вынуждают пользователей вашего модуля явно передать провайдер для каждого из ваших псевдонимов конфигурации, используя ассоциативный массив `providers`.

Откройте `example/multi-account-module/main.tf` и определите блоки `provider`, как это делали раньше:

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

provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole"
  }
}
```

А теперь передайте их в модуль `modules/multi-account`:

```
module "multi_account_example" {
  source = "../modules/multi-account"

  providers = {
    aws.parent = aws.parent
    aws.child  = aws.child
  }
}
```

Ключи в ассоциативном массиве `providers` должны соответствовать именам псевдонимов конфигурации внутри модуля; если какие-либо имена псевдонимов конфигурации отсутствуют в ассоциативном массиве `providers`, то Terraform сообщит об ошибке. Поэтому, создавая многоуровневый модуль, вы можете определить, какие провайдеры ему нужны, а Terraform обеспечит передачу этих провайдеров пользователями; а создавая корневой модуль, вы сможете определить свои блоки `provider` только один раз и передать ссылки на них используемым многоуровневым модулям.

Работа с несколькими провайдерами

Теперь вы увидели, как работать с несколькими провайдерами, относящимися к одному типу: например, с несколькими копиями провайдера `aws`. В этом разделе я расскажу, как работать с несколькими разными провайдерами.

Читатели первых двух изданий этой книги часто просили привести примеры совместного использования нескольких облаков, но у меня не нашлось ничего полезного, чем бы я мог поделиться. Отчасти это связано с тем, что использование нескольких облаков обычно считается плохой практикой¹, но даже если вы вынуждены это делать (большинство крупных компаний используют несколько облаков, хотя бы они этого или нет), то редко когда приходится управлять несколькими облаками в одном модуле по той же причине, по которой редко когда приходится в одном модуле управлять несколькими регионами или учетными записями. Если вы используете несколько облаков, то лучше управлять каждым из них в отдельном модуле.

Более того, переводить каждый отдельный пример AWS в книгу на эквивалентное решение для других облаков (Azure и Google Cloud) нецелесообразно: книга получится слишком объемной — да, вы узнаете некоторые детали о каждом облаке, но я не смогу познакомить вас с новыми концепциями, поддерживаемыми в Terraform, что и является настоящей целью книги. Если вы хотите увидеть примеры кода Terraform для развертывания аналогичной инфраструктуры в разных облаках, загляните в папку `examples` в репозитории Terratest (<https://github.com/gruntwork-io/terratest>). Как будет показано в главе 9, Terratest предоставляет набор инструментов для создания автоматических тестов для разных инфраструктур и разных облаков, поэтому в папке `examples` вы найдете код Terraform для развертывания аналогичной инфраструктуры в AWS, Google Cloud и Azure, включая отдельные серверы, группы серверов, базы данных и многое другое. Также в папке `test` вы найдете автоматические тесты для всех этих примеров.

В этой книге вместо нереалистичного примера развертывания инфраструктуры в нескольких облаках я решил показать, как использовать несколько провайдеров при немного более реалистичном сценарии (который также просили проиллюстрировать многие читатели первых двух изданий), а именно: как использовать провайдер AWS с провайдером Kubernetes для развертывания приложений в контейнерах Docker. Кластер Kubernetes по сути является облаком — в нем могут работать приложения, сети, хранилища данных, балансировщики нагрузки, хранилища секретов и многое другое, так что в некотором смысле пример использования AWS и Kubernetes — это пример использования нескольких

¹ См. статью *Multi-Cloud is the Worst Practice* (<https://www.lastweekinaws.com/blog/multi-cloud-is-the-worst-practice/>).

провайдеров и нескольких облаков одновременно. А поскольку Kubernetes — облако, это означает, что вам придется многое узнать. Но все по очереди. Сначала я кратко расскажу о Docker и Kubernetes, а затем перейду к полному примеру использования нескольких провайдеров — AWS и Kubernetes:

- краткий курс Docker;
- краткий курс Kubernetes;
- развертывание контейнеров Docker в AWS с помощью Elastic Kubernetes Service (EKS).

Краткий курс Docker

Как рассказывалось в главе 1, образы Docker представляют собой моментальные снимки операционной системы (ОС), программного обеспечения, файлов и всех других важных деталей. Теперь посмотрим, как действует Docker.

Во-первых, если вы еще не установили Docker, последуйте инструкциям на веб-сайте <https://docs.docker.com/get-docker/>, чтобы установить Docker Desktop в своей операционной системе. После установки вам должна быть доступна команда `docker`. Вы можете использовать команду `docker run` для запуска образов Docker на локальном компьютере:

```
$ docker run <IMAGE> [COMMAND]
```

IMAGE — это запускаемый образ Docker, а **COMMAND** — необязательная команда для выполнения внутри запущенного образа. Например, вот как можно запустить командную оболочку Bash в Docker-образе Ubuntu 20.04 (обратите внимание, что в команде ниже есть флаг `-it`, обеспечивающий получение интерактивной оболочки, в которой можно вводить и выполнять команды):

```
$ docker run -it ubuntu:20.04 bash
```

```
Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
Digest: sha256:669e010b58baf5beb2836b253c1fd5768333f0d1dbcb834f7c07a4dc93f474be
Status: Downloaded newer image for ubuntu:20.04
```

```
root@d96ad3779966:/#
```

И теперь вы оказались в Ubuntu! Если вы никогда раньше не использовали Docker, то произошедшее может показаться волшебством. Попробуйте выполнить несколько команд. Например, вы можете просмотреть содержимое файла `/etc/os-release` и убедиться, что действительно находитесь в Ubuntu:

```
root@d96ad3779966:/# cat /etc/os-release
NAME="Ubuntu"
```

```

VERSION="20.04.3 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.3 LTS"
VERSION_ID="20.04"
VERSION_CODENAME=focal

```

Как это произошло? Во-первых, Docker отыскал в вашей локальной файловой системе образ `ubuntu:20.04`. Если этот образ еще не был загружен, то Docker автоматически загрузит его из *реестра* Docker Hub, содержащего общедоступные образы Docker. Образ `ubuntu:20.04` является общедоступным — официально поддерживаемым командой Docker, поэтому его можно загрузить без аутентификации. Однако также можно создавать частные образы Docker, доступные ограниченному кругу аутентифицированных пользователей.

Загрузив образ, Docker запускает его и выполняет команду `bash`, которая выводит интерактивное приглашение к вводу, где вы можете вводить команды. Попробуйте запустить команду `ls`, чтобы получить список файлов:

```

root@d96ad3779966:/# ls -al
total 56
drwxr-xr-x  1 root root 4096 Feb 22 14:22 .
drwxr-xr-x  1 root root 4096 Feb 22 14:22 ..
lrwxrwxrwx  1 root root    7 Jan 13 16:59 bin -> usr/bin
drwxr-xr-x  2 root root 4096 Apr 15 2020 boot
drwxr-xr-x  5 root root 360 Feb 22 14:22 dev
drwxr-xr-x  1 root root 4096 Feb 22 14:22 etc
drwxr-xr-x  2 root root 4096 Apr 15 2020 home
lrwxrwxrwx  1 root root    7 Jan 13 16:59 lib -> usr/lib
drwxr-xr-x  2 root root 4096 Jan 13 16:59 media
(...)

```

Вы можете заметить, что это не ваша файловая система. Это объясняется тем, что образы Docker запускаются в контейнерах, изолированных на уровне пользовательского пространства: внутри вы можете видеть только файловую систему, память, сеть и т. д., имеющиеся в этом контейнере. Любые данные в других контейнерах или в операционной системе хоста недоступны из вашего контейнера, и любые данные в вашем контейнере недоступны этим другим контейнерам или операционной системе хоста. Это один из аспектов, делающих Docker полезным для запуска приложений: образы являются самодостаточными и поэтому работают одинаково независимо от того, где они запускаются и что в них выполняется.

Чтобы убедиться в этом, запишите текст в файл `test.txt`, как показано ниже:

```

root@d96ad3779966:/# echo "Hello, World!" > test.txt

```

Затем выйдите из контейнера, нажав **Ctrl+D** в Windows и Linux или **Cmd+D** в macOS, чтобы вернуться в исходную командную строку в своей операционной системе. Если вы попытаетесь найти только что созданный файл `test.txt`, то обнаружите, что он отсутствует: файловая система контейнера полностью изолирована от вашей ОС.

Теперь снова запустите тот же образ Docker:

```
$ docker run -it ubuntu:20.04 bash
root@3e0081565a5d:/#
```

Обратите внимание, что на этот раз контейнер запустился почти мгновенно, потому что образ `ubuntu:20.04` уже загружен и находится в локальной файловой системе. Это еще одна особенность Docker, делающая его привлекательным для запуска приложений: в отличие от виртуальных машин контейнеры быстро загружаются и оказывают небольшую нагрузку на процессор или память.

Также можно заметить, что после второго запуска контейнера приглашение к вводу в командной строке выглядит иначе. Это объясняется тем, что теперь вы находитесь в совершенно новом контейнере; любые данные, сохраненные в прошлый раз, вам больше не доступны. Запустите `ls -al`, и вы увидите, что файл `test.txt` не существует. Контейнеры изолированы не только от системы-носителя, но и друг от друга.

Еще раз нажмите **Ctrl+D** или **Cmd+D**, чтобы выйти из контейнера и вернуться в свою ОС, и запустите команду `docker ps -a`:

```
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND        CREATED        STATUS
3e0081565a5d   ubuntu:20.04   "bash"        5 min ago     Exited (0) 16 sec ago
d96ad3779966   ubuntu:20.04   "bash"        14 min ago     Exited (0) 5 min ago
```

Эта команда выведет список всех контейнеров в вашей системе, включая остановленные (те, из которых вы вышли). Запустить остановленный контейнер можно командой `docker start <ID>`, указав ID — идентификатор из столбца `CONTAINER ID` в выводе команды `docker ps`. Например, вот как можно снова запустить первый контейнер (в интерактивном режиме, указав флаги `-ia`):

```
$ docker start -ia d96ad3779966
root@d96ad3779966:/#
```

Чтобы убедиться, что это действительно первый контейнер, выведите содержимое `test.txt`:

```
root@d96ad3779966:/# cat test.txt
Hello, World!
```

Теперь посмотрим, как можно использовать контейнер для запуска веб-приложения. Нажмите **Ctrl+D** или **Cmd+D** еще раз, чтобы выйти из контейнера, вернитесь в свою ОС и запустите новый контейнер:

```
$ docker run training/webapp
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Образ `training/webapp` (<https://github.com/docker-training/webapp>) содержит простое веб-приложение Hello, World на Python. После запуска этот образ запускает веб-приложение, по умолчанию прослушивающее порт 5000. Однако если вы откроете новый терминал в своей операционной системе и попытаетесь получить доступ к веб-приложению, то у вас ничего не получится:

```
$ curl localhost:5000
curl: (7) Failed to connect to localhost port 5000: Connection refused
```

В чем проблема? На самом деле это не проблема, а достоинство! Контейнеры Docker изолированы от операционной системы хоста и других контейнеров не только на уровне файловой системы, но и на уровне сети. Поэтому, хотя контейнер действительно прослушивает порт 5000, этот порт доступен только внутри контейнера, но недоступен в операционной системе хоста. Чтобы открыть доступ к порту контейнера из ОС хоста, нужно добавить флаг `-p`.

Сначала нажмите **Ctrl+C**, чтобы закрыть контейнер `training/webapp`. Обратите внимание, что на сей раз это **C**, а не **D**, и **Ctrl** независимо от ОС, потому что вам нужно завершить процесс, а не выйти из интерактивного режима. Теперь перезапустите контейнер, но уже с флагом `-p`, как показано ниже:

```
$ docker run -p 5000:5000 training/webapp
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Добавление `-p 5000:5000` в команду требует от Docker открыть порт 5000 внутри контейнера и связать его с портом 5000 ОС хоста. Теперь в другом терминале хост-системы вы сможете увидеть работу веб-приложения:

```
$ curl localhost:5000
Hello world!
```



Очистка контейнеров

Каждый раз, выполняя команду `docker run` и завершая работу, вы оставляете после себя контейнеры, которые занимают дисковое пространство. При желании эти контейнеры можно удалить с помощью команды `docker rm <CONTAINER_ID>`, где `CONTAINER_ID` — это идентификатор контейнера из вывода `docker ps`. В качестве альтернативы можно включить флаг `--rm` в команду `docker run`, чтобы Docker автоматически удалял контейнеры после выхода из них.

Краткий курс по Kubernetes

Kubernetes — это инструмент оркестрации для Docker, то есть платформа для запуска контейнеров Docker и управления ими на ваших серверах, включая планирование (выбор серверов, которые должны выполнять определенные контейнеры), автоматическое восстановление (автоматическое повторное развертывание вышедших из строя контейнеров), автоматическое масштабирование (изменение количества контейнеров в большую и меньшую сторону в зависимости от нагрузки), балансировка нагрузки (распределение трафика между контейнерами) и многое другое.

Внутри Kubernetes делится на две основные части.

- *Плоскость управления* — отвечает за управление кластером Kubernetes. Это «мозг» кластера, отвечающий за сохранение его состояния, мониторинг контейнеров и координацию действий в кластере. Она также запускает сервер API, который можно использовать с помощью инструментов командной строки (например, `kubectl`), веб-интерфейсов (например, Kubernetes Dashboard) и инструментов IaC (например, Terraform) для управления происходящим в кластере.
- *Рабочие узлы* — это серверы, используемые для фактического запуска контейнеров. Рабочие узлы управляются плоскостью управления, которая сообщает каждому рабочему узлу, какие контейнеры он должен запускать.

Kubernetes — это открытое программное обеспечение, и одна из его сильных сторон — возможность запускаться где угодно: в любом общедоступном облаке (например, AWS, Azure, Google Cloud), в своем собственном центре обработки данных и даже на рабочей станции разработчика. Чуть позже в этой главе я покажу, как можно запускать Kubernetes в облаке (в AWS), а пока начнем с малого и сделаем это локально. Это не составит труда, если установить относительно недавнюю версию Docker Desktop, позволяющую запустить кластер Kubernetes локально несколькими щелчками кнопкой мыши.

Если откроете настройки Docker Desktop на своем компьютере, то увидите навигационную форму Kubernetes (рис. 7.7).

Если флажок **Enable Kubernetes** (Включить Kubernetes) еще не установлен, то установите его, затем нажмите кнопку **Apply & Restart** (Применить и перезапустить) и подождите несколько минут, пока процесс запуска завершится. А пока следуйте инструкциям на сайте Kubernetes (<https://kubernetes.io/docs/tasks/tools/>), чтобы установить `kubectl` — инструмент командной строки для взаимодействия с Kubernetes.

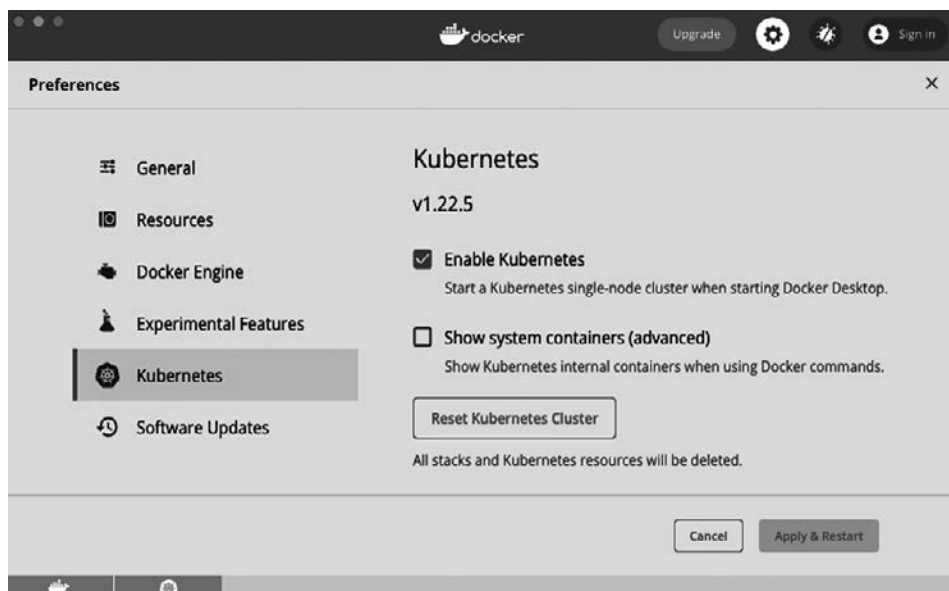


Рис. 7.7. Включение Kubernetes в Docker Desktop

Чтобы использовать `kubectl`, нужно сначала обновить его конфигурационный файл, который находится в `$HOME/.kube/config` (то есть в папке `.kube` в вашем домашнем каталоге), и указать, к какому кластеру Kubernetes подключаться. Удобно, что, когда вы включаете Kubernetes в Docker Desktop, этот файл обновляется автоматически и в него добавляется запись `docker-desktop`, поэтому вам остается только указать `kubectl` использовать эту конфигурацию:

```
$ kubectl config use-context docker-desktop
Switched to context "docker-desktop".
```

Теперь можно проверить, работает ли кластер Kubernetes, выполнив команду `get nodes`:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
docker-desktop	Ready	control-plane,master	95m	v1.22.5

Команда `get nodes` выводит информацию обо всех узлах в кластере. Поскольку вы используете Kubernetes локально, то ваш компьютер является единственным узлом, поэтому на нем выполняется плоскость управления, и одновременно он играет роль рабочего узла. Теперь можно попробовать запустить несколько контейнеров Docker.

Чтобы развернуть что-то в Kubernetes, нужно создать ряд *объектов* Kubernetes — постоянно хранимые сущности, которые вы передаете в кластер Kubernetes

(через сервер API) и которые описывают ваше намерение, например, запустить определенные образы Docker. Кластер запускает *цикл согласования*, который постоянно проверяет объекты, переданные вами, и обеспечивает соответствие состояния кластера вашим намерениям.

В Kubernetes поддерживается множество разных объектов. В примерах мы используем два объекта.

- *Kubernetes Deployment* — предлагает декларативный способ управления приложением в Kubernetes. Вы указываете, какие образы Docker запускать, количество их копий (называемых *репликами*), различные настройки для этих образов (например, количество процессоров, объем памяти, номера портов, переменные окружения) и стратегию развертывания обновлений для этих образов. После этого объект Kubernetes Deployment будет стремиться обеспечить постоянное выполнение заявленных вами требований. Например, если вы указали, что хотите иметь три реплики, но один из рабочих узлов вышел из строя и осталось только две реплики, то Deployment автоматически развернет третью на одном из других рабочих узлов.
- *Kubernetes Service* — позволяет представить веб-приложение, работающее в Kubernetes, как сетевой сервис. Объект Kubernetes Service, например, можно использовать для настройки балансировщика нагрузки, предоставляющего общедоступную конечную точку и распределяющего поступающий в нее трафик между репликами, определяемыми объектом Kubernetes Deployment.

Идиоматический способ взаимодействия с Kubernetes — создать файлы YAML, описывающие то, что вы хотите (например, один файл YAML, определяющий объект Kubernetes Deployment, а другой — Kubernetes Service), и вызвать команду `kubectl apply` для отправки этих объектов в кластер. Однако использование формата YAML имеет недостатки, такие как отсутствие поддержки повторного использования кода (например, переменных, модулей), абстракций (таких как циклы, операторы `if`), четких стандартов хранения файлов YAML и управления ими (например, для отслеживания изменений в кластере с течением времени) и т. д. Поэтому многие пользователи Kubernetes обращаются к альтернативам, как вариант к Helm или Terraform. Поскольку это книга о Terraform, я покажу, как создать модуль Terraform с именем `k8s-app` (K8S — это акроним, образованный от слова *Kubernetes*, точно так же как I18N — это акроним, образованный от слова *internationalization* — «интернационализация»). Этот модуль развертывает приложение в Kubernetes с помощью Kubernetes Deployment и Kubernetes Service.

Создайте новый модуль в папке `elements/services/k8s-app`. В ней же создайте файл `variables.tf`, определяющий API модуля в виде следующих входных переменных:

```
variable "name" {  
  description = "The name to use for all resources created by this module"
```

```
    type          = string
  }

  variable "image" {
    description = "The Docker image to run"
    type       = string
  }

  variable "container_port" {
    description = "The port the Docker image listens on"
    type       = number
  }

  variable "replicas" {
    description = "How many replicas to run"
    type       = number
  }

  variable "environment_variables" {
    description = "Environment variables to set for the app"
    type       = map(string)
    default    = {}
  }
```

С их помощью вы сможете передать в модуль практически все данные, необходимые для создания объектов Deployment и Service. Далее добавьте файл `main.tf` и в начало этого файла — блок `require_providers` с провайдером Kubernetes:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    kubernetes = {
      source = "hashicorp/kubernetes"
      version = "~> 2.0"
    }
  }
}
```

Обратите внимание: это новый для вас провайдер! Теперь воспользуемся им для создания Kubernetes Deployment с помощью ресурса `kubernetes_deployment`:

```
resource "kubernetes_deployment" "app" {
}
```

В ресурсе `kubernetes_deployment` нужно настроить довольно много параметров, поэтому не будем спешить и рассмотрим их по одному. Для начала настроим блок `metadata`:

```
resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }
}
```


Каждый объект Kubernetes включает метаданные, которые можно использовать для идентификации этого объекта в вызовах API. В коде выше я присвоил объекту Deployment имя из входной переменной `name`.

Остальная часть конфигурации ресурса `kubernetes_deployment` находится в блоке `spec`:

```
resource "kubernetes_deployment" "app" {  
  metadata {  
    name = var.name  
  }  
  
  spec {  
  }  
}
```

Первым делом поместим в блок `spec` параметр `replicas`, определяющий количество создаваемых реплик:

```
spec {  
  replicas = var.replicas  
}
```

Затем определим блок `template`:

```
spec {  
  replicas = var.replicas  
  
  template {  
  }  
}
```

В Kubernetes вместо отдельных контейнеров принято развертывать так называемые *поды* (или модули) — *Pod* — группы контейнеров, которые должны развертываться совместно. Например, у вас может быть Pod с одним контейнером для запуска веб-приложения (например, приложением на Python, которое вы видели ранее) и с другим контейнером, собирающим метрики веб-приложения и отправляющим их в центральную службу (например, DataDog). Блок `template` определяет шаблон модуля Pod, который описывает, какие контейнеры нужно запускать, какие порты использовать, какие переменные окружения установить и т. д.

Один из важных компонентов шаблона модуля Pod — метки. Эти метки вам понадобятся в нескольких местах, например, Kubernetes Service использует метки для идентификации модулей Pod, которые необходимо балансировать по нагрузке. Поэтому определим эти метки в локальной переменной с именем `pod_labels`:

```
locals {  
  pod_labels = {  
    app = var.name  
  }  
}
```

А теперь используем `pod_labels` в блоке `metadata` шаблона модуля `Pod`:

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }
  }
}
```

Далее добавим блок `spec` внутрь шаблона:

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }

    spec {
      container {
        name = var.name
        image = var.image

        port {
          container_port = var.container_port
        }

        dynamic "env" {
          for_each = var.environment_variables
          content {
            name = env.key
            value = env.value
          }
        }
      }
    }
  }
}
```

Здесь довольно много интересного, поэтому рассмотрим код внимательнее.

- `container` — внутри блока `spec` можно определить несколько блоков `container`, чтобы указать, какие контейнеры Docker будут запускаться в этом модуле `Pod`. Для простоты поместим в наш `Pod` только один блок `container`. Остальные элементы, перечисленные ниже, находятся в нем.
- `name` — имя, которое будет использоваться для идентификации контейнера. Я присваиваю этому параметру значение входной переменной `name`.

- `image` — образ Docker для запуска в контейнере. Я присваиваю этому параметру значение входной переменной `image`.
- `port` — порты в контейнере, к которым должен быть открыт доступ. Для простоты я предполагаю, что контейнеру потребуется прослушивать только один порт, номер которого определяет входная переменная `container_port`.
- `env` — переменные окружения, которые будут доступны в контейнере. Здесь я использую блок `dynamic` с `for_each` (две концепции, которые мы рассматривали в главе 5), чтобы настроить переменные окружения, исходя из значения входной переменной `environment_variables`.

На этом определение шаблона модуля Pod завершается. Осталось добавить в ресурс `kubernetes_deployment` блок `selector`:

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }

    spec {
      container {
        name = var.name
        image = var.image

        port {
          container_port = var.container_port
        }

        dynamic "env" {
          for_each = var.environment_variables
          content {
            name = env.key
            value = env.value
          }
        }
      }
    }
  }

  selector {
    match_labels = local.pod_labels
  }
}
```

Блок `selector` сообщает объекту Kubernetes Deployment его цель. Присвоив ему значение `pod_labels`, мы сообщаем, что он должен управлять развертыванием только что определенного шаблона модуля Pod. Почему бы объекту Deployment

просто не предположить, что целевым для него является шаблон Pod, прописанный в нем? Дело в том, что Kubernetes пытается быть чрезвычайно гибкой и слабосвязанной системой: например, позволяет определить Deployment для развертывания модулей Pod, которые прописаны отдельно, поэтому всегда нужно указывать `selector`, чтобы объект Deployment знал свою цель.

На этом определение ресурса `kubernetes_deployment` завершается. Следующий шаг — использование ресурса `kubernetes_service` для создания объекта Kubernetes Service:

```
resource "kubernetes_service" "app" {
  metadata {
    name = var.name
  }

  spec {
    type = "LoadBalancer"
    port {
      port          = 80
      target_port    = var.container_port
      protocol       = "TCP"
    }
    selector = local.pod_labels
  }
}
```

Пройдемся по его параметрам.

- `metadata` — подобно объекту Deployment, объект Service использует метаданные для идентификации этого объекта в вызовах API. В коде выше я присвоил объекту Service имя из входной переменной `name`.
- `type` — я присвоил этому объекту Service тип `LoadBalancer`, и теперь в зависимости от настроек кластера Kubernetes будет развертывать балансировщик нагрузки того или иного типа: например, в AWS с EKS будет использоваться Elastic Load Balancer, а в Google Cloud с GKE — Cloud Load Balancer.
- `port` — как я предполагаю, балансировщик нагрузки будет пересылать трафик, получаемый им через порт 80 (порт HTTP по умолчанию), в порт, который прослушивает контейнер.
- `selector` — подобно Deployment, объект Service использует параметр `selector`, чтобы сообщить объекту Service его цель. Если присвоить параметру `selector` значение `pod_labels`, то Service и Deployment будут работать с одними и теми же модулями Pod.

Последний шаг — экспортирование конечной точки Service (имя хоста балансировщика нагрузки) через выходную переменную, прописанную в файле `outputs.tf`:

```
locals {
  status = kubernetes_service.app.status
}

output "service_endpoint" {
  value = try(
    "http://${local.status[0]["load_balancer"][0]["ingress"][0]["hostname"]}",
    "(error parsing hostname from status)"
  )
  description = "The K8S Service endpoint"
}
```

Этот запутанный код требует некоторого объяснения. Ресурс `kubernetes_service` имеет выходной атрибут `status`, возвращающий последнее состояние Service. Я сохранил этот атрибут в локальной переменной `status`. Для объекта Service типа `LoadBalancer` атрибут `status` будет содержать сложный объект, который выглядит примерно так:

```
[
  {
    load_balancer = [
      {
        ingress = [
          {
            hostname = "<ИМЯ_ХОСТА>"
          }
        ]
      }
    ]
  }
]
```

Внутри этого объекта скрыто имя хоста балансировщика нагрузки. Вот почему в определении выходной переменной `service_endpoint` пришлось использовать сложную последовательность операций поиска в массиве (например, `[0]`) и в ассоциативном массиве (например, `["load_balancer"]`), чтобы извлечь имя хоста. Но что произойдет, если атрибут `status`, возвращаемый ресурсом `kubernetes_service`, будет выглядеть немного иначе? Тогда любая операция поиска в массиве и ассоциативном массиве может завершиться неудачей, что приведет к запутанной ошибке.

Чтобы корректно обработать эту ошибку, я поместил все выражение в функцию с именем `try`. Она имеет следующий синтаксис:

```
try(ARG1, ARG2, ..., ARGN)
```

Функция вычисляет все переданные ей аргументы и возвращает первый аргумент, который не вызовет ошибки. Таким образом, выходная переменная `service_endpoint` будет содержать имя хоста (первый аргумент) или, если

возникнет ошибка чтения имени хоста, текст `error parsing hostname from status` (ошибка синтаксического анализа имени хоста из атрибута `status`), определяемый вторым аргументом.

Хорошо, на этом определение модуля `k8s-app` можно считать завершенным. Чтобы использовать его, добавьте новый пример в `example/kubernetes-local` и создайте файл `main.tf` с таким содержимым:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name          = "simple-webapp"
  image         = "training/webapp"
  replicas      = 2
  container_port = 5000
}
```

Этот пример настраивает модуль для развертывания образа Docker `training/webapp`, который вы запускали ранее, с двумя репликами, прослушивающими порт 5000, и присваивает всем объектам Kubernetes (опираясь на их метаданные) имя `simple-webapp`. Чтобы развернуть этот модуль в локальном кластере Kubernetes, добавьте следующий блок `provider`:

```
provider "kubernetes" {
  config_path = "~/.kube/config"
  config_context = "docker-desktop"
}
```

Код сообщает провайдеру Kubernetes, что он должен аутентифицироваться в локальном кластере Kubernetes, используя контекст `docker-desktop` из конфигурации `kubectl`. Запустите `terraform apply`, чтобы увидеть, как все это работает:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
service_endpoint = "http://localhost"
```

Дайте приложению несколько секунд для загрузки, а затем попробуйте обратиться к `service_endpoint`:

```
$ curl http://localhost
Hello world!
```

Отлично!

Результат выглядит почти идентично выводу команды `docker run`, так имело ли смысл выполнять всю эту сложную работу? Что ж, заглянем внутрь и посмотрим, что происходит. Для изучения кластера можно использовать команду `kubectl`. Сначала запустите команду `get deployments`:

```
$ kubectl get deployments
NAME          READY  UP-TO-DATE  AVAILABLE  AGE
simple-webapp  2/2    2           2          3m21s
```

Здесь можно видеть объект Kubernetes Deployment с именем `simple-webapp`, которое вы задали в блоке `metadata`. Этот объект сообщает, что два из двух модулей Pod (две реплики) готовы к работе. Чтобы увидеть эти модули Pod, запустите команду `get pods`:

```
$ kubectl get pods
NAME                                READY  STATUS   RESTARTS  AGE
simple-webapp-d45b496fd-7d447       1/1    Running   0          2m36s
simple-webapp-d45b496fd-vl6j7       1/1    Running   0          2m36s
```

Вот первое отличие от команды `docker run`: здесь работает несколько контейнеров, а не один. Более того, эти контейнеры активно контролируются и управляются. Например, в случае сбоя одного из них автоматически будет развернута замена. Вы можете убедиться в этом, выполнив команду `docker ps`:

```
$ docker ps
CONTAINER ID  IMAGE                COMMAND                  CREATED          STATUS
b60f5147954a  training/webapp      "python app.py"         3 seconds ago   Up 2 seconds
c350ec648185  training/webapp      "python app.py"         12 minutes ago  Up 12 minutes
```

Передайте идентификатор `CONTAINER ID` одного из этих контейнеров команде `docker kill`, чтобы закрыть его:

```
$ docker kill b60f5147954a
```

Если очень быстро снова запустить команду `docker ps`, то вы увидите, что остался работать только один контейнер:

```
$ docker ps
CONTAINER ID  IMAGE                COMMAND                  CREATED          STATUS
c350ec648185  training/webapp      "python app.py"         12 minutes ago  Up 12 minutes
```

Но всего через несколько секунд Kubernetes Deployment обнаружит, что осталась только одна реплика вместо запрошенных двух, и автоматически запустит новый контейнер:

```
$ docker ps
CONTAINER ID  IMAGE                COMMAND                  CREATED          STATUS
56a216b8a829  training/webapp      "python app.py"         1 second ago    Up 5 seconds
c350ec648185  training/webapp      "python app.py"         12 minutes ago  Up 12 minutes
```

Таким образом, Kubernetes гарантирует, что всегда будет работать ожидаемое количество реплик. Более того, он также запускает балансировщик нагрузки для распределения трафика между репликами, что можно увидеть, выполнив команду `kubectl get services`:

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	4h26m
simple-webapp	LoadBalancer	10.110.25.79	localhost	80:30234/TCP	4m58s

Первый сервис в списке — сам Kubernetes, его можно игнорировать. Второй — созданный вами объект Service с именем `simple-webapp` (заданным в блоке `metadata`). Этот сервис запускает балансировщик нагрузки для вашего приложения: вы можете видеть IP-адрес, по которому он доступен (`localhost`), и порт, который он прослушивает (80).

Объекты Kubernetes Deployment также обеспечивают автоматическое развертывание обновлений. Вот забавный трюк с Docker-образом `training/webapp`: если создать переменную окружения `PROVIDER` с каким-либо значением, он будет использовать это значение вместо слова `world` в тексте `Hello, world`. Обновите `example/kubernetes-local/main.tf`, чтобы установить эту переменную окружения:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }
}
```

Запустите команду `apply` еще раз:

```
$ terraform apply
```

```
(...)
```

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

```
Outputs:
```

```
service_endpoint = "http://localhost"
```


Спустя несколько секунд снова попробуйте обратиться к конечной точке:

```
$ curl http://localhost
Hello Terraform!
```

Объект Deployment автоматически развернул ваши изменения: по умолчанию эти объекты автоматически накатывают обновления, подобно группам автомасштабирования (обратите внимание, что настройки развертывания можно изменить, добавив блок `strategy` в ресурс `kubernetes_deployment`).

Развертывание контейнеров Docker в AWS с помощью Elastic Kubernetes Service (EKS)

В Kubernetes есть еще одна хитрость: он достаточно универсален. То есть те же образы Docker и конфигурации Kubernetes можно повторно использовать в совершенно другом кластере и получить аналогичные результаты. Чтобы убедиться в этом, развернем кластер Kubernetes в AWS.

Настройка безопасного, высокодоступного и масштабируемого кластера Kubernetes в облаке с нуля и управление им довольно сложны. К счастью, большинство облачных провайдеров предлагают управляемые сервисы Kubernetes и автоматически запускают плоскость управления и рабочие узлы: например, Elastic Kubernetes Service (EKS) в AWS, Azure Kubernetes Service (AKS) в Azure и Google Kubernetes Engine (GKE) в Google Cloud. Я хочу показать, как развернуть очень простой кластер EKS в AWS.

Создайте новый модуль в `modules/services/eks-cluster` и определите API модуля в файле `variables.tf` со следующими входными переменными:

```
variable "name" {
  description = "The name to use for the EKS cluster"
  type        = string
}

variable "min_size" {
  description = "Minimum number of nodes to have in the EKS cluster"
  type        = number
}

variable "max_size" {
  description = "Maximum number of nodes to have in the EKS cluster"
  type        = number
}

variable "desired_size" {
  description = "Desired number of nodes to have in the EKS cluster"
```

```

    type          = number
  }

  variable "instance_types" {
    description = "The types of EC2 instances to run in the node group"
    type        = list(string)
  }

```

Этот код экспортирует входные переменные, определяющие имена, размеры и типы серверов в кластере EKS, которые будут использоваться для запуска рабочих узлов. Далее в `main.tf` создайте роль IAM для плоскости управления:

```

# Создать роль IAM для плоскости управления
resource "aws_iam_role" "cluster" {
  name           = "${var.name}-cluster-role"
  assume_role_policy = data.aws_iam_policy_document.cluster_assume_role.json
}

# Разрешить EKS присваивать себе роль IAM
data "aws_iam_policy_document" "cluster_assume_role" {
  statement {
    effect = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type       = "Service"
      identifiers = ["eks.amazonaws.com"]
    }
  }
}

# Определить разрешения для роли IAM
resource "aws_iam_role_policy_attachment" "AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.cluster.name
}

```

Эту роль IAM может присвоить себе сервис EKS, и для нее определена управляемая политика IAM, которая предоставляет плоскости управления необходимые разрешения. Теперь добавьте источники данных `aws_vpc` и `aws_subnets`, чтобы получить информацию о VPC по умолчанию и ее подсетях:

```

# Этот код предназначен только для экспериментов и обучения,
# поэтому он использует МЗС и ее подсети по умолчанию.
# В промышленной среде следует использовать специально
# настроенную VPC и частные подсети.

```

```

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

data "aws_subnets" "default" {
  filter {

```

```

    name    = "vpc-id"
    values  = [data.aws_vpc.default.id]
  }
}

```

Теперь можно создать плоскость управления для кластера EKS, используя ресурс `aws_eks_cluster`:

```

resource "aws_eks_cluster" "cluster" {
  name      = var.name
  role_arn  = aws_iam_role.cluster.arn
  version   = "1.21"

  vpc_config {
    subnet_ids = data.aws_subnets.default.ids
  }

  # Разрешения для роли IAM должны определяться до запуска и после остановки
  # кластера EKS. В противном случае EKS не сможет правильно удалить
  # инфраструктуру EC2, в частности группы безопасности, управляемые EKS.
  depends_on = [
    aws_iam_role_policy_attachment.AmazonEKSClusterPolicy
  ]
}

```

Приведенный выше код настраивает плоскость управления для использования только что созданной роли IAM и для развертывания в VPC и подсетях по умолчанию.

Далее идут рабочие узлы. EKS поддерживает несколько типов рабочих узлов: самоуправляемые серверы EC2 (например, в создаваемой вами ASG), серверы EC2, управляемые AWS (известные как *группа управляемых узлов*), и Fargate (бессерверные)¹. Для простоты в примерах этой главы будут использоваться группы управляемых узлов.

Чтобы развернуть группу управляемых узлов, сначала необходимо создать еще одну роль IAM:

```

# Создать роль IAM для группы узлов
resource "aws_iam_role" "node_group" {
  name      = "${var.name}-node-group"
  assume_role_policy = data.aws_iam_policy_document.node_assume_role.json
}

# Разрешить серверам EC2 присваивать себе роль IAM
data "aws_iam_policy_document" "node_assume_role" {
  statement {
    effect = "Allow"

```

¹ Сравнение различных типов рабочих узлов EKS можно найти в статье по адресу <https://blog.gruntwork.io/comprehensive-guide-to-eks-worker-nodes-94e241092cbe>.

```

    actions = ["sts:AssumeRole"]
    principals {
      type       = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}

# Определить разрешения для группы узлов
resource "aws_iam_role_policy_attachment" "AmazonEKSWorkerNodePolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEC2ContainerRegistryReadOnly" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEKS_CNI_Policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.node_group.name
}

```

Эту роль IAM может присвоить себе сервис EC2 (что имеет смысл, поскольку внутри группы управляемых узлов используют серверы EC2), и к ней прикреплено несколько управляемых политик IAM, которые предоставляют группе необходимые разрешения. Теперь для создания самой группы управляемых узлов можно использовать ресурс `aws_eks_node_group`:

```

resource "aws_eks_node_group" "nodes" {
  cluster_name   = aws_eks_cluster.cluster.name
  node_group_name = var.name
  node_role_arn  = aws_iam_role.node_group.arn
  subnet_ids     = data.aws_subnets.default.ids
  instance_types = var.instance_types

  scaling_config {
    min_size     = var.min_size
    max_size     = var.max_size
    desired_size = var.desired_size
  }
}

# Разрешения для роли IAM должны определяться до запуска и после остановки
# группы узлов EKS. В противном случае EKS не сможет правильно удалить
# серверы EC2 и сетевые интерфейсы Elastic.
depends_on = [
  aws_iam_role_policy_attachment.AmazonEKSWorkerNodePolicy,
  aws_iam_role_policy_attachment.AmazonEC2ContainerRegistryReadOnly,
  aws_iam_role_policy_attachment.AmazonEKS_CNI_Policy,
]
}

```

Этот код настраивает группу управляемых узлов для использования только что созданной плоскости управления и роли IAM, для развертывания в VPC по умолчанию и установки имен, размеров и типов серверов, заданных во входных переменных.

Добавьте в файл `outputs.tf` следующие выходные переменные:

```
output "cluster_name" {
  value      = aws_eks_cluster.cluster.name
  description = "Name of the EKS cluster"
}

output "cluster_arn" {
  value      = aws_eks_cluster.cluster.arn
  description = "ARN of the EKS cluster"
}

output "cluster_endpoint" {
  value      = aws_eks_cluster.cluster.endpoint
  description = "Endpoint of the EKS cluster"
}

output "cluster_certificate_authority" {
  value      = aws_eks_cluster.cluster.certificate_authority
  description = "Certificate authority of the EKS cluster"
}
```

Теперь модуль `eks-cluster` готов к работе. Давайте воспользуемся им и предыдущим модулем `k8s-app` и развернем кластер EKS на основе образа `Docker training/webapp`. Создайте файл `example/kubernetes-eks/main.tf` и настройте модуль `eks-cluster`, как показано ниже:

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

module "eks_cluster" {
  source = "../../modules/services/eks-cluster"

  name      = "example-eks-cluster"
  min_size  = 1
  max_size  = 2
  desired_size = 1

  # Из-за особенностей работы EKS с ENI, t3.small является самым маленьким
  # типом серверов, который можно использовать для рабочих узлов. Если
  # попытаться использовать что-то меньшее, например t2.micro, у которого
  # всего четыре ENI, они будут использованы системными службами (например,
  # kube-proxy) и вы не сможете развернуть собственные модули Pod.
  instance_types = ["t3.small"]
}
```

Далее настройте модуль `k8s-app`:

```
provider "kubernetes" {
  host = module.eks_cluster.cluster_endpoint
  cluster_ca_certificate = base64decode(
    module.eks_cluster.cluster_certificate_authority[0].data
  )
  token = data.aws_eks_cluster_auth.cluster.token
}

data "aws_eks_cluster_auth" "cluster" {
  name = module.eks_cluster.cluster_name
}

module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name          = "simple-webapp"
  image         = "training/webapp"
  replicas      = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }

  # Приложения должны развертываться только после развертывания кластера
  depends_on = [module.eks_cluster]
}
```

Приведенный выше код настраивает провайдер Kubernetes для аутентификации в кластере EKS, а не в локальном кластере Kubernetes (запущенном из Docker Desktop). Затем использует модуль `k8s-app` для развертывания образа Docker `training/webapp`, как и при развертывании в Docker Desktop; единственное отличие — дополнительный параметр `depend_on`, который гарантирует развертывание образа Docker только после развертывания кластера EKS.

Затем передайте конечную точку сервиса в выходной переменной:

```
output "service_endpoint" {
  value          = module.simple_webapp.service_endpoint
  description = "The K8S Service endpoint"
}
```

Теперь все готово к развертыванию! Запустите `terraform apply` как обычно (обратите внимание, что развертывание кластеров EKS может занять 10–20 минут, поэтому наберитесь терпения):

```
$ terraform apply
```

```
(...)
```

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

Outputs:

```
service_endpoint = "http://774696355.us-east-2.elb.amazonaws.com"
```

Подождите немного, пока веб-приложение запустится и пройдет проверку работоспособности, а затем протестируйте `service_endpoint`:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Terraform!
```

Вот и все! Тот же образ Docker и код Kubernetes теперь работают в кластере EKS в AWS точно так же, как работали на локальном компьютере. Здесь работают все те же функции. Попробуйте обновить `environment_variables` и присвоить другое значение переменной `PROVIDER`, например `"Readers"`:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name          = "simple-webapp"
  image         = "training/webapp"
  replicas      = 2

  container_port = 5000

  environment_variables = {
    PROVIDER = "Readers"
  }

  # Приложения должны развертываться только после развертывания кластера
  depends_on = [module.eks_cluster]
}
```

Еще раз запустите `apply`, и всего через несколько секунд Kubernetes Deployment развернет изменения:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Readers!
```

Это одно из преимуществ использования Docker: изменения можно развернуть очень быстро.

Вы можете снова воспользоваться инструментом `kubectl`, чтобы увидеть, что происходит в кластере. Чтобы аутентифицировать `kubectl` в кластере EKS, введите команду `aws eks update-kubeconfig` для автоматического обновления вашего файла `$HOME/.kube/config`:

```
$ aws eks update-kubeconfig --region <REGION> --name <EKS_CLUSTER_NAME>
```

REGION — это регион AWS, а EKS_CLUSTER_NAME — имя вашего кластера EKS. В модуле Terraform вы указали регион us-east-2 и дали кластеру имя kubernetes-example, поэтому команда будет выглядеть так:

```
$ aws eks update-kubeconfig --region us-east-2 --name kubernetes-example
```

Теперь, как и раньше, можете вызвать команду `get nodes` для получения списка рабочих узлов в кластере, но на этот раз добавьте флаг `-o wide`, чтобы увидеть немного больше информации:

```
$ kubectl get nodes
NAME                                STATUS    AGE    EXTERNAL-IP    OS-IMAGE
xxx.us-east-2.compute.internal    Ready    22m    3.134.78.187    Amazon Linux 2
```

Предыдущий фрагмент вывода сильно урезан, чтобы можно было уместить его по ширине книжной страницы, но в реальности вы увидите один рабочий узел, его внутренний и внешний IP-адрес, номер версии, информацию об ОС и многое другое.

Чтобы получить список объектов Deployment, можно использовать команду `get deployments`:

```
$ kubectl get deployments
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
simple-webapp  2/2      2              2            19m
```

Затем запустите `get pods`, чтобы получить список модулей Pod:

```
$ kubectl get pods
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
simple-webapp  2/2      2              2            19m
```

И наконец, запустите `get services`, чтобы получить список объектов Service:

```
$ kubectl get services
NAME          TYPE          EXTERNAL-IP    PORT(S)
kubernetes    ClusterIP     <none>          443/TCP
simple-webapp  LoadBalancer 774696355.us-east-2.elb.amazonaws.com  80/TCP
```

Вы должны увидеть свой балансировщик нагрузки и URL, который использовали для его тестирования.

Вот и все: два разных провайдера работают в одном облаке, помогая вам развертывать контейнерные рабочие нагрузки.

Тем не менее, как и в предыдущих разделах, я хочу сделать несколько предупреждений.

- *Предупреждение 1: эти примеры Kubernetes очень упрощены!* Kubernetes — сложный фреймворк. Он быстро развивается и меняется. Попытка объяснить все детали легко потянет на целую книгу. Поскольку данная книга посвящена Terraform, а не Kubernetes, в этой главе я старался максимально упростить

примеры, демонстрирующие использование Kubernetes. Поэтому, хотя я надеюсь, что мои примеры кода были полезны для обучения и экспериментов, если вы собираетесь использовать Kubernetes в реальной ситуации, вам придется изменить многие аспекты этого кода, такие как настройки дополнительных сервисов и настройки в модуле `eks-cluster` (например, входные контроллеры, шифрование, группы безопасности, аутентификация OIDC, отображение RBAC, VPC CNI, kube-proxy, CoreDNS), экспортировать множество других настроек в модуль `k8s-app` (например, управление секретами, тома, проверки работоспособности, проверки готовности, метки, аннотации, несколько портов, несколько контейнеров) и использование настраиваемой VPC с частными подсетями для кластера EKS вместо VPC по умолчанию и общедоступных подсетей¹.

- *Предупреждение 2: не злоупотребляйте возможностью использовать несколько провайдеров.* Несмотря на возможность использовать несколько провайдеров в одном модуле, я не рекомендую делать это слишком часто по тем же причинам, по которым не рекомендую слишком часто использовать псевдонимы провайдеров: в большинстве случаев предпочтительнее изолировать провайдеры в отдельных модулях, чтобы управлять им отдельно и ограничить радиус поражения от ошибок или злоумышленников.

Кроме того, Terraform не может похвастаться хорошей поддержкой упорядочения зависимостей между провайдерами. Так, в примере с Kubernetes мы использовали один модуль для развертывания кластера EKS с помощью провайдера AWS и приложения Kubernetes в этом кластере — с помощью провайдера Kubernetes. Однако, как оказывается, в документации провайдера Kubernetes (<https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs>) явно не рекомендуется применять такой подход.

При использовании интерполяции для передачи учетных данных провайдера Kubernetes из других ресурсов эти ресурсы *не следует* создавать в том же модуле Terraform, где также используются ресурсы провайдера Kubernetes. Это приведет к периодическим и непредсказуемым ошибкам, которые трудно выявить и устранить. Основная проблема заключается в том, что Terraform оценивает блоки `provider` в порядке, отличном от порядка оценки фактических ресурсов.

Пример кода в книге обходит эти проблемы, используя источник данных `aws_eks_cluster_auth`, но это не универсальное решение. Поэтому в промышленном коде я рекомендую развертывать кластер EKS в одном модуле, а приложения Kubernetes — в отдельных модулях, запускаемых уже после развертывания кластера.

¹ В качестве альтернативы можете использовать готовые модули Kubernetes промышленного уровня, например входящие в «инфраструктуру как библиотека кода» Gruntwork (<https://gruntwork.io/infrastructure-as-code-library/>).

Резюме

Надеюсь, вы поняли, как работать с несколькими провайдерами в коде Terraform, и сможете ответить на три вопроса из начала этой главы.

- *Как поступить, если потребуется выполнить развертывание в нескольких регионах AWS?* Использовать несколько блоков `provider` с разными параметрами `region` и `alias`.
- *Как поступить, если понадобится выполнить развертывание в нескольких учетных записях AWS?* Использовать несколько блоков `provider` с разными настройками в блоках `take_role` и в параметре `alias`.
- *Как поступить, если потребуется выполнить развертывание в других облаках, например Azure или GCP?* Использовать несколько блоков `provider` с разными настройками для соответствующих облаков.

Вы также наверняка заметили, что использование нескольких провайдеров в одном модуле обычно считается антипаттерном. Поэтому реальный ответ на эти вопросы, особенно в промышленных сценариях: использовать каждый провайдер в своем модуле, чтобы отделить разные регионы, учетные записи и облака друг от друга и ограничить радиус влияния неприятностей.

Теперь перейдем к главе 8, где я покажу несколько других шаблонов создания модулей Terraform для промышленных сценариев — на такие модули вы можете сделать ставку в своей компании.

Код Terraform промышленного уровня

Построение инфраструктуры промышленного уровня — сложный и напряженный процесс, который отнимает много времени. Под *инфраструктурой промышленного уровня* я имею в виду все, без чего не может обойтись ваша компания. На вас полагаются с надеждой, 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 без хранимого состояния, 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 хранимым состоянием, 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 один-два 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. 8.1.

Таблица 8.1. 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 хранимого состояния)	Кластер приложений Node.js	2–4 недели
Распределенная система с самостоятельным размещением (с хранимым 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 масштабной миграции в облако 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;
- компонентные модули;
- тестируемые модули;
- модули, готовые к выпуску;
- модули вне Terraform.



Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу <https://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 приближена. И эта оценка удваивается для 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 Хофштадтера¹.

Закон Хофштадтера: работа всегда занимает больше времени, чем вы ожидаете, даже с учетом закона Хофштадтера.

Даглас Хофштадтер

¹ Хофштадтер Д. Гедель, Эшер, Бах: эта бесконечная гирлянда. — Бахрах-М, 2001.

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 не знаком этот термин, уверяю, вы его полюбите (и возненавидите). Лучшее его определение, которое мне встречалось, опубликовано в блоге Сета Година¹.

«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 отполировать 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 приступить 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

¹ Don't Shave That Yak! (блог Сета, 5 марта 2005 года), <https://bit.ly/2OK45uL>.

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 связывания 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 работаете¹. 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.

¹ Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы. Юбилейное издание. — М.: Символ-Плюс, 2015.

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 защиты серверов. Если вам повезет, один из респондентов может вспомнить о резервном копировании данных и агрегировании журналов. У большинства компаний нет четкого списка требований к выпуску промышленной системы. Значит, каждый элемент инфраструктуры развертывается немного по-своему и ему может нехватать каких-то критически важных функций.

Чтобы улучшить эту ситуацию, хочу поделиться с вами *контрольным списком задач для подготовки инфраструктуры промышленного уровня* (табл. 8.2). Этот список охватывает большинство ключевых моментов, которые необходимо учитывать при развертывании инфраструктуры в промышленной среде.

Таблица 8.2. Список задач для подготовки инфраструктуры промышленного уровня

Задача	Описание	Примеры инструментов
Установка	Установите исполняемые файлы ПО и все зависимости	Bash, Ansible, Docker, Packer
Конфигурация	Сконфигурируйте ПО на этапе выполнения, включая настройки портов, сертификаты TLS, обнаружение сервисов, выбор ведущих и ведомых серверов, репликацию и т. д.	Chef, Ansible, Kubernetes
Выделение	Выделите 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, пакетные и канареечные развертывания	ASG, Kubernetes, ECS
Высокая доступность	Система должна выдерживать перебои в работе отдельных процессов, серверов, сервисов, вычислительных центров и регионов	Несколько вычислительных центров и регионов

Задача	Описание	Примеры инструментов
Масштабируемость	Масштабируйте систему в зависимости от нагрузки. Это касается как горизонтального (больше серверов), так и вертикального масштабирования (более крупные серверы)	Автомасштабирование, репликация
Производительность	Оптимизируйте использование процессора, памяти, диска, сети и графического адаптера. Это относится к ускорению запросов, эталонному тестированию, нагрузочному тестированию и профилированию	Dynatrace, valgrind, VisualVM
Сеть	Сконфигурируйте статические и динамические IP-адреса, порты, обнаружение сервисов, брандмауэры, DNS, а также доступ по SSH и VPN	Облака VPC, брандмауэры, Route 53
Безопасность	Шифрование при передаче (TLS) и на диске, аутентификация, авторизация, управление конфиденциальными данными, укрепление серверов	ACM, Let's Encrypt, KMS, Vault
Показатели	Показатели доступности, бизнес-показатели, показатели приложения, показатели серверов, события, наблюдаемость, трассировка и оповещения	CloudWatch, DataDog,
Журналы	Организируйте ротацию журналов на диске. Агрегируйте журналы в центральном хранилище	Elastic Stack, Sumo Logic
Резервное копирование и восстановление	Проводите плановое резервное копирование БД, кэшей и других данных. Реплицируйте данные для разделения регионов или учетных записей	AWS Backup, моментальные снимки RDS
Оптимизация расходов	Выбирайте подходящие типы серверов, используйте прерываемые и резервируемые серверы, применяйте автомасштабирование и избавляйтесь от ненужных ресурсов	Автомасштабирование, уменьшение расходов на инфраструктуру
Документация	Документируйте свой код, архитектуру и практикуемые методики. Создавайте инструкции на случай разных происшествий	Файлы README, вики, Slack, IaC
Тесты	Пишите для своего инфраструктурного кода автоматические тесты. Выполняйте их после каждой фиксации кода и по ночам	Terratest, tfint, OPA, inspect

Большинству разработчиков известно о первых нескольких задачах: установке, конфигурации, выделении 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;
- компонентные модули;
- тестируемые модули;
- модули, готовые к выпуску;
- модули вне Terraform.

Мелкие 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» главы 3, это плохая идея. Я на этом не останавливаюсь и утверждаю следующее: большие модули, которые содержат более нескольких сотен строк 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 `terraform plan` уходило пять-шесть 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 `terraform plan`, 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 9.

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»¹.

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. Вы сразу же понимаете, что этот код дурно пахнет и его лучше разбить на ряд небольших, автономных функций, каждая из которых делает что-то одно. Ту же стратегию нужно применять и к Terraform.

¹ Мартин Р. Чистый код. Создание, анализ и рефакторинг. Библиотека программиста. — СПб.: Питер, 2019.

Представьте, что вы имеете дело с архитектурой, изображенной на рис. 8.1.

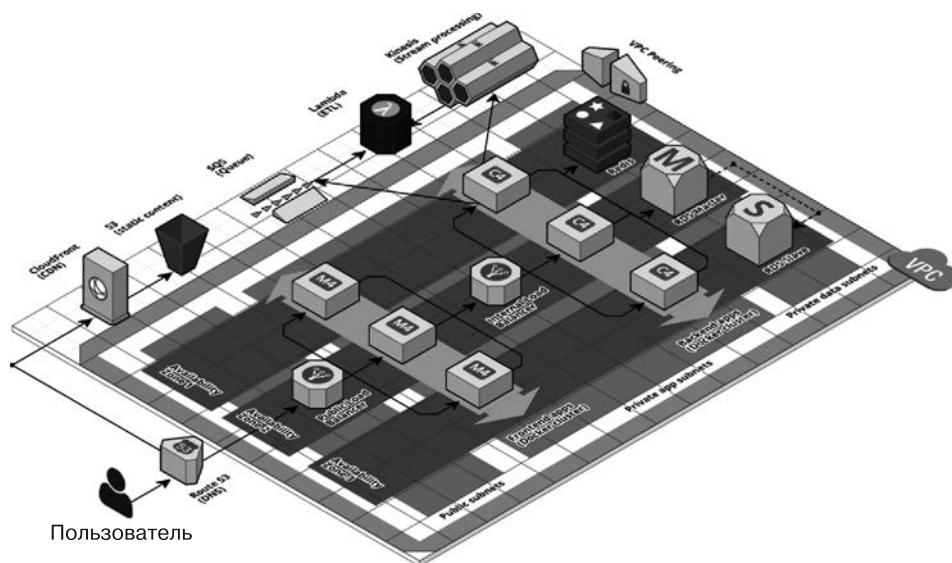


Рис. 8.1. Относительно сложная архитектура AWS

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. 8.2).

Модуль `webserver-cluster`, над которым вы работали в главе 5, начинает разрабатываться. К тому же он отвечает сразу за три малосвязанные между собой задачи.

- *Группа автомасштабирования (ASG)*. Модуль `webserver-cluster` разворачивает группу ASG, которая умеет выполнять пакетные обновления с нулевым временем простоя.
- *Балансировщик нагрузки (ALB)*. Модуль `webserver-cluster` разворачивает ALB.
- *Демонстрационное приложение*. Модуль `webserver-cluster` также разворачивает простое демонстрационное приложение.

Разделим этот код на три небольших модуля.

- `modules/cluster/asg-rolling-deploy`. Обобщенный универсальный автономный модуль для разворачивания группы ASG, которая умеет выполнять пакетные обновления с нулевым временем простоя.

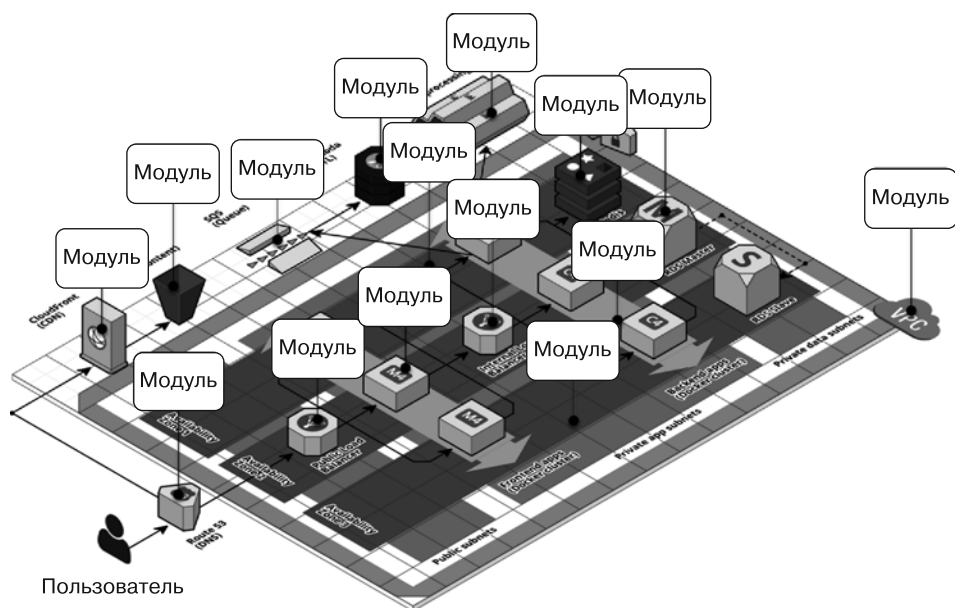


Рис. 8.2. Относительно сложная архитектура AWS, разбитая на множество мелких модулей

- `modules/networking/alb`. Обобщенный универсальный автономный модуль для развертывания ALB.
- `modules/services/hello-world-app`. Модуль для развертывания демонстрационного приложения, использующий модули `asg-rolling-deploy` и `alb`.

Прежде чем начинать, не забудьте выполнить команду `terraform destroy` для удаления всех копий `webserver-cluster`, которые могли остаться после чтения предыдущих глав. После этого можете приступать к написанию модулей `asg-rolling-deploy` и `alb`. Создайте новую папку `modules/cluster/asg-rolling-deploy` и скопируйте 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 скопируйте 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 скопируйте 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 для настройки параметра `name` ресурса `aws_lb`:

```
resource "aws_lb" "example" {  
  name           = var.alb_name  
  load_balancer_type = "application"  
  subnets       = data.aws_subnets.default.ids  
  security_groups = [aws_security_group.alb.id]  
}
```

и параметра `name` 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 <https://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. Прочитую Дуга Макилроя¹, 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 на вход другой. Представьте, что у вас есть следующие небольшие функции на Ruby:

```
# Простая функция сложения
def add(x, y)
  return x + y
end

# Простая функция вычитания
def sub(x, y)
  return x - y
end

# Простая функция умножения
def multiply(x, y)
  return x * y
end
```

С помощью композиции функций вы можете их объединить, 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. Также вместо записи состояния вовне лучше возвращать результаты своих вычислений через выходные параметры. Минимизация побочных эффектов — это один из основных принципов функционального программирования, который упрощает

¹ *Salus P. H.* A Quarter-Century of Unix. — New York: Addison-Wesley Professional, 1994.

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     = null
}
```

Первая переменная, `subnet_ids`, сообщает модулю `asg-rolling-deploy`, в каких подсетях осуществлять развертывание. 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, 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 входящих 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_subnets`) и не были скопированы в наш модуль `asg-rolling-deploy`:

```
resource "aws_autoscaling_group" "example" {
  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

  # (...)
}
```

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`:

```
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 конфигурации запуска
  # вместе с группой автомасштабирования.
  # 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 исходящих значений его пользователи смогут изменять его поведение, подключая собственные правила к группе безопасности.

По аналогичным причинам несколько исходящих 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.

Последний шаг — 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 и источники данных:

- `aws_lb_target_group`;
- `aws_lb_listener_rule`;
- `terraform_remote_state` (для БД);
- `aws_vpc`;
- `aws_subnets`.

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
  instance_type = var.instance_type

  user_data = templatefile("${path.module}/user-data.sh", {
    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
  })

  min_size      = var.min_size
  max_size      = var.max_size
  enable_autoscaling = var.enable_autoscaling

  subnet_ids      = data.aws_subnets.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_subnets.default.ids
}

```

Обратите внимание, 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 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` ссылался на выходную переменную `alb_http_listener_arn` модуля `alb`:

```

resource "aws_lb_listener_rule" "asg" {
  listener_arn = module.alb.alb_http_listener_arn
  priority     = 100

  condition {
    path_pattern {
      values = ["*"]
    }
  }

  action {
    type           = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}

```

И передайте важные выходные переменные из `asg-rolling-deploy` и `alb` в модуль `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: здесь вы выстраиваете сложное поведение (приложение Hello, World) из более простых частей (модули ASG и ALB).

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.

Эти модули не корневые и не предназначены для развертывания напрямую. Чтобы их развернуть, нужно написать какой-то код Terraform, передать нужные 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          = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  min_size      = 1
  max_size      = 1
  enable_autoscaling = false

  subnet_ids = data.aws_subnets.default.ids
}

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

data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

Этот фрагмент кода развертывает группу ASG с одним сервером, применяя модуль `asg-rolling-deploy`. Попробуйте выполнить команды `terraform init` и `terraform apply` и убедитесь, что все работает без ошибок и ASG действительно создается. Теперь добавьте файл `README.md` с этими инструкциями, после чего этот крошечный пример станет куда более мощным. С помощью всего нескольких файлов и строк кода вы получаете следующее.

- *Механизм тестирования вручную.* Этот демонстрационный код можно использовать при работе над модулем `asg-rolling-deploy`, развертывая и удаляя его по много раз путем выполнения команд `terraform apply` и `terraform destroy`. Это позволит убедиться в его предсказуемом поведении.
- *Механизм автоматического тестирования.* Как вы увидите в главе 9, 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, находящегося 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 объединить все это вместе, 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
```

```
L modules
  L alb
  L asg-rolling-deploy
  L hello-world-app
  L mysql
L test
  L alb
  L asg-rolling-deploy
  L hello-world-app
  L mysql
```

В качестве упражнения предлагаю вам добавить дополнительные примеры для модулей `alb`, `asg-rolling-deploy`, `mysql` и `hello-world-app`.

Разработку нового модуля крайне полезно *начинать* с написания демонстрационного кода. Если сразу заняться реализацией, легко погрязнуть в деталях, и к моменту, когда вы вернетесь к API, ваш модуль будет малопонятным и сложным в применении. Если же начать с 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 9.

В этом разделе я сосредоточусь на создании самопроверяемых модулей, то есть модулей, способных проверять свое поведение и предотвращать некоторые виды ошибок. В Terraform для этой цели реализовано два встроенных способа:

- валидация;
- пред- и постусловия.

Валидации

В Terraform 0.13 появилась возможность добавлять *блоки валидации* `validation` в определения любых входных переменных, чтобы обеспечить проверку соответствия передаваемых значений некоторым ограничениям. Например, можно добавить блок `validation` в определение переменной `instance_type`, чтобы убедиться, что значение, переданное пользователем, является строкой (это обеспечивается ограничением типа), но и имеет одно из двух значений, соответствующих бесплатному тарифу AWS:

```
variable "instance_type" {
  description = "The type of EC2 Instances to run (e.g. t2.micro)"
  type        = string
```

```
validation {
  condition      = contains(["t2.micro", "t3.micro"], var.instance_type)
  error_message = "Only free tier is allowed: t2.micro | t3.micro."
}
}
```

Блок `validation` работает следующим образом: параметр условия должен получить значение `true`, если значение действительное, и `false` в противном случае. В параметре `error_message` можно указать сообщение, которое будет показано пользователю при передаче недопустимого значения. Например, вот что произойдет, если попытаться установить для параметра `instance_type` значение `m4.large`, не предусмотренное бесплатным тарифом AWS:

```
$ terraform apply -var instance_type="m4.large"
Error: Invalid value for variable

   on main.tf line 17:
    1: variable "instance_type" {
      |   var.instance_type is "m4.large"

Only free tier is allowed: t2.micro | t3.micro.

This was checked by the validation rule at main.tf:21,3-13.
```

Для проверки нескольких условий можно включить в определение переменной несколько блоков `validation`:

```
variable "min_size" {
  description = "The minimum number of EC2 Instances in the ASG"
  type        = number

  validation {
    condition      = var.min_size > 0
    error_message = "ASGs can't be empty or we'll have an outage!"
  }

  validation {
    condition      = var.min_size <= 10
    error_message = "ASGs must have 10 or fewer instances to keep costs down."
  }
}
```

Обратите внимание, что блоки `validation` имеют существенное ограничение: параметр `condition` может ссылаться *только* на входные переменные, полученные из вмещающего окружения. Если попытаться сослаться на любые другие входные переменные, локальные переменные, ресурсы или источники данных, Terraform сообщит об ошибке. Поэтому, несмотря на очевидную полезность для проверки входных данных, блоки `validation` нельзя использовать для более сложных условий, например включающих проверку нескольких переменных

одновременно (таких как «только одна из двух входных переменных должна иметь значение»), или любых динамических проверок (например, АМІ, запрошенный пользователем, использует архитектуру x86_64). Для выполнения подобных динамических проверок необходимо использовать блоки пред- и пост-условий, которые описываются далее.

Пред- и постусловия

В Terraform 1.2 появилась возможность добавлять блоки пред- и постусловий (`precondition` и `postcondition` соответственно) в определения ресурсов, источников данных и выходных переменных для выполнения более динамических проверок. Блоки `precondition` предназначены для выявления ошибок перед запуском `apply`. Например, блок `precondition` можно использовать для более надежной проверки принадлежности типа сервера, который передает пользователь, к бесплатному тарифу AWS. В предыдущем разделе такая проверка выполнялась с помощью блока `validation` и жестко запрограммированного списка типов экземпляров, но такие списки быстро устаревают. Вместо этого можно использовать источник данных `instance_type_data` и всегда получать самую актуальную информацию от AWS:

```
data "aws_ec2_instance_type" "instance" {
  instance_type = var.instance_type
}
```

Затем в определение ресурса `aws_launch_configuration` можно добавить блок `precondition`, проверяющий соответствие типа сервера бесплатному тарифу AWS:

```
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 конфигурации запуска
  # вместе с группой автомасштабирования
  lifecycle {
    create_before_destroy = true
    precondition {
      condition      = data.aws_ec2_instance_type.instance.free_tier_eligible
      error_message = "${var.instance_type} is not part of the AWS Free Tier!"
    }
  }
}
```

Как и блоки `validation`, блоки `precondition` (и `postcondition`, как вы вскоре увидите) имеют параметр `condition`, который должен получать значение `true`

или `false`, и параметр `error_message`, определяющий текст сообщения об ошибке, которое выводится, когда `condition` получает значение `false`. Если теперь попытаться запустить `apply` с типом сервера, не входящим в бесплатный тариф AWS, то вы увидите сообщение об ошибке:

```
$ terraform apply -var instance_type="m4.large"
Error: Resource precondition failed

   on main.tf line 25, in resource "aws_launch_configuration" "example":
  18:     condition = data.aws_ec2_instance_type.instance.free_tier_eligible
      |_____
      | data.aws_ec2_instance_type.instance.free_tier_eligible is false
      |
m4.large is not part of the AWS Free Tier!
```

Блоки постусловий `postcondition` предназначены для выявления ошибок после выполнения `apply`. Например, блок `postcondition` можно добавить в определение ресурса `aws_autoscaling_group`, чтобы убедиться, что группа ASG была развернута более чем в одной зоне доступности (AZ), и тем самым гарантировать допустимость сбоя хотя бы в одной AZ:

```
resource "aws_autoscaling_group" "example" {
  name                = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier = var.subnet_ids

  lifecycle {
    postcondition {
      condition     = length(self.availability_zones) > 1
      error_message = "You must use more than one AZ for high availability!"
    }
  }

  # (...)
}
```

Обратите внимание на использование ключевого слова `self` в параметре `condition`. *Выражения* `self` поддерживают следующий синтаксис:

```
self.<ATTRIBUTE>
```

Этот синтаксис можно использовать исключительно в блоках `postcondition`, `connection` и `provisioner` (примеры последних двух вы увидите далее в этой главе) для ссылки на выходной атрибут окружающего ресурса. Если попытаться использовать стандартный синтаксис `aws_autoscaling_group.example.<ATTRIBUTE>`, то вы получите ошибку циклической зависимости, поскольку ресурсы не могут ссылаться сами на себя, соответственно выражение `self` можно рассматривать как обходное решение, добавленное специально для подобных вариантов использования.

Если запустить `apply` для этого модуля, Terraform развернет модуль, но, если после этого выяснится, что все подсети, переданные пользователем через входную переменную `subnet_ids`, находятся в одной и той же зоне доступности, блок `postcondition` покажет ошибку и вы будете знать, что ваша ASG не настроена на высокую доступность.

Когда использовать блоки `validation`, `precondition` и `postcondition`

Как вы увидели, блоки `validation`, `precondition` и `postcondition` во многом похожи, поэтому возникает вопрос: когда следует использовать каждый из них?

- *Используйте блоки `validation` для простых проверок входных данных.* Делайте это во всех модулях промышленного уровня, чтобы воспрепятствовать передаче недопустимых значений в модули. Цель состоит в том, чтобы выявить основные ошибки во входных данных *до того, как* будут развернуты какие-либо изменения. Хотя блоки `precondition` являются более мощными, всегда, когда это возможно, для проверок переменных лучше использовать блоки `validation`, потому что в их определениях используются переменные, благодаря чему код получается более читаемый и удобный в обслуживании.
- *Используйте блоки `precondition` для проверки основных условий.* Делайте это во всех модулях промышленного уровня для проверки условий, которые должны соблюдаться перед развертыванием изменений. К ним относятся любые проверки переменных, которые невозможно выполнить с помощью блоков `validation` (например, проверки, ссылающиеся на несколько переменных или источников данных), а также проверки ресурсов и источников данных. Цель состоит в том, чтобы выявить как можно больше ошибок и как можно раньше, прежде чем они нанесут какой-либо ущерб.
- *Используйте блоки `postcondition` для проверки основных гарантий.* Делайте это во всех модулях промышленного уровня, чтобы проверить соблюдение гарантий его поведения *после* развертывания изменений. Цель в том, чтобы дать пользователям модуля уверенность в том, что после запуска `apply` он выполнит все заявленное либо завершится с ошибкой, а сопровождающим модуль — послать более четкий сигнал о желательном поведении модуля, чтобы оно случайно не потерялось в результате рефакторинга.
- *Для обеспечения более сложных условий и гарантий используйте инструменты автоматического тестирования.* Блоки `validation`, `precondition` и `postcondition` — полезные инструменты, но они могут выполнять только простые проверки, потому что позволяют ссылаться лишь на источники данных, ресурсы и языковые конструкции, встроенные в Terraform, а для проверки более сложного поведения этих возможностей обычно недостаточно.

Например, создав модуль для развертывания веб-сервиса, вы можете добавить проверку после развертывания, чтобы убедиться, что веб-сервис отвечает на HTTP-запросы. Это можно реализовать в блоке `postcondition`, отправив HTTP-запрос сервису с помощью провайдера `http` (<https://registry.terraform.io/providers/hashicorp/http/latest/docs>) в Terraform, но большинство развертываний происходит асинхронно, поэтому вам может потребоваться повторить HTTP-запрос несколько раз, но проблема в том, что в этот провайдер не встроено механизма повтора. Более того, если вы развернули внутренний веб-сервис, он может быть недоступен через общедоступный Интернет, поэтому вам сначала нужно будет подключиться к какой-либо внутренней сети или VPN, что тоже довольно сложно реализовать в коде Terraform. Поэтому для таких сложных проверок лучше использовать инструменты автоматического тестирования, например OPA и Terratest, с которыми вы познакомитесь в главе 9.

Версионирование модулей

Существует два типа версионирования модулей, о которых вы должны знать:

- версионирование зависимостей модуля;
- версионирование самого модуля.

Начнем с версионирования зависимостей модуля. Код Terraform имеет три типа зависимостей.

1. От версии ядра Terraform — версии двоичного файла `terraform`.
2. От версий провайдеров — версий каждого провайдера, используемого в коде, например `aws`.
3. От версий модулей — версий каждого модуля, получаемого через блок `module`.

Вам следует попрактиковаться в *закреплении версий* для всех зависимостей. Под этим подразумевается закрепление версий всех трех типов зависимостей и использование определенных, фиксированных и известных версий. Развертывания должны быть предсказуемыми и повторяемыми: если код не изменился, запуск `apply` всегда должен давать один и тот же результат, независимо от того, когда он запускается — сегодня, через три месяца или через три года. Для этого необходимо избегать случайного добавления новых версий зависимостей: действие по обновлению версии всегда должно быть явным, преднамеренным и хорошо видимым в коде, который вы отправляете в систему управления версиями.

Давайте посмотрим, как закрепить версию каждого из трех типов зависимостей Terraform.

Для закрепления версии зависимости первого типа — версии ядра Terraform — можно использовать аргумент `required_version`. Как минимум вам потребуется определить основную версию Terraform:

```
terraform {  
  # Требуем любую версию Terraform вида 1.x  
  required_version = ">= 1.0.0, < 2.0.0"  
}
```

Это очень важно, потому что при каждом изменении основного номера версии Terraform теряет обратную совместимость: например, переход с 1.0.0 на 2.0.0, скорее всего, будет включать критические изменения, поэтому вам вряд ли понравится, если такой переход произойдет случайно. Приведенный выше код позволит вам использовать с этим модулем только версии Terraform 1.xx, поэтому версии 1.0.0 и 1.2.3 будут работать, но если вы попытаетесь использовать (возможно, случайно) 0.14.3 или 2.0.0 и запустите `terraform apply`, то сразу получите ошибку:

```
$ terraform apply
```

```
Error: Unsupported Terraform Core version
```

```
This configuration does not support Terraform version 0.14.3. 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 1.2.3  
  required_version = "= 1.2.3"  
}
```

Раньше, до выпуска Terraform 1.0.0, это было абсолютно необходимо, поскольку каждая версия Terraform потенциально включала обратно несовместимые изменения, в том числе в формате файла состояния: например, файл состояния в Terraform 0.12.1 не читался в Terraform 0.12.0. К счастью, после выпуска версии 1.0.0 ситуация изменилась: согласно официально опубликованным обещаниям совместимости Terraform v1.0, обновления между промежуточными выпусками v1.x не должны требовать никаких изменений в коде или рабочих процессах.

Тем не менее вы едва ли захотите *случайно* перейти на новую версию Terraform. В новых версиях появляются новые функции, и если некоторые из ваших компьютеров (рабочие станции разработчиков и серверы CI) начнут использовать эти функции, а другие продолжают применять старые версии, вы столкнетесь с проблемами. Более того, новые версии Terraform могут содержать ошибки, и их следует опробовать в тестовых средах, прежде чем развертывать

в промышленной среде. Поэтому, хотя закрепления старшего номера версии часто бывает достаточно, я также рекомендую закрепить младший номер версии и номер исправления и затем осторожно и последовательно применять обновления Terraform в каждой среде.

Обратите внимание, что иногда может потребоваться использовать разные версии Terraform в одной кодовой базе. Например, вы можете начать тестировать Terraform 1.2.3 в тестовой среде, продолжая использовать Terraform 1.0.0 в промышленной среде. Поддержка нескольких версий Terraform на локальном компьютере или на серверах CI может оказаться непростой задачей. К счастью, существует инструмент с открытым исходным кодом `tfenv` (<https://github.com/tfutils/tfenv>) — менеджер версий Terraform, значительно упрощающий ее.

С помощью `tfenv` можно установить нескольких версий Terraform и переключаться между ними по мере необходимости. Например, установить определенную версию Terraform можно с помощью команды `tfenv install`:

```
$ tfenv install 1.2.3
Installing Terraform v1.2.3
Downloading release tarball from
https://releases.hashicorp.com/terraform/1.2.3/terraform_1.2.3_darwin_amd64.zip
Archive: tfenv_download.ZUS3Qn/terraform_1.2.3_darwin_amd64.zip
  inflating: /opt/homebrew/Cellar/tfenv/2.2.2/versions/1.2.3/terraform
Installation of terraform v1.2.3 successful.
```



tfenv на Apple Silicon (M1, M2)

По состоянию на июнь 2022 `tfenv` не устанавливал подходящую версию Terraform на Apple Silicon, например, на компьютеры Mac с процессорами M1 и M2 (подробности см. в этом отчете о проблеме: <https://github.com/tfutils/tfenv/issues/306>). Обходное решение — установить значение `arm64` в переменной окружения `TFENV_ARCH`:

```
$ export TFENV_ARCH=arm64
$ tfenv install 1.2.3
```

Получить список установленных версий можно с помощью команды `list`:

```
$ tfenv list
  1.2.3
  1.1.4
  1.1.0
* 1.0.0 (set by /opt/homebrew/Cellar/tfenv/2.2.2/version)
```

А выбрать версию Terraform из этого списка для использования — с помощью команды `use`:

```
$ tfenv use 1.2.3
Switching default version to v1.2.3
Switching completed
```

Все эти команды удобны для работы с несколькими версиями Terraform, но настоящая мощь `tfenv` заключается в поддержке файлов `.terraform-version`. `tfenv` автоматически ищет файл `.terraform-version` в текущей папке, а также во всех родительских, вплоть до корня проекта, то есть до корня управления версиями (например, до папки, содержащей папку `.git`), и если такой файл будет найден, то любая команда `terraform` будет автоматически использовать версию, указанную в этом файле.

Например, если вы решите опробовать Terraform 1.2.3 в тестовой среде, продолжая пользоваться Terraform 1.0.0 в промышленной, используйте следующую структуру папок:

```
live
├── stage
│   ├── vpc
│   ├── mysql
│   ├── frontend-app
│   └── .terraform-version
└── prod
    ├── vpc
    ├── mysql
    ├── frontend-app
    └── .terraform-version
```

Файл `live/stage/.terraform-version` должен содержать текст:

```
1.2.3
```

А файл `live/prod/.terraform-version` — такой текст:

```
1.0.0
```

Теперь любая команда `terraform`, которую вы запустите в папке `stage` или в любой ее подпапке, автоматически использует Terraform 1.2.3. Убедиться в этом можно, выполнив команду `terraform version`:

```
$ cd stage/vpc
$ terraform version
Terraform v1.2.3
```

Аналогично любая команда `terraform`, которую вы запускаете в папке `prod`, будет автоматически использовать Terraform 1.0.0:

```
$ cd prod/vpc
$ terraform version
Terraform v1.0.0
```

Требуемая версия будет выбираться автоматически на любой рабочей станции разработчика и на сервере CI, если вы повсюду установите `tfenv`. Пользователи `Terragrunt` могут для той же цели обратиться к инструменту `tgswitch`, автоматически выбирающему версию `Terragrunt` на основе файла `.terragrunt-version`.

Теперь обратим внимание на второй тип зависимостей в коде Terraform — зависимости от провайдеров. Как было показано в главе 7, для закрепления версии провайдера можете использовать блок `require_providers`:

```
terraform {  
  required_version = ">= 1.0.0, < 2.0.0"  
  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = "~> 4.0"  
    }  
  }  
}
```

Этот пример привязывает код к любой версии провайдера AWS 4.x (синтаксис `~> 4.0` эквивалентен `>= 4.0, < 5.0`). И снова, чтобы избежать случайного внесения обратно несовместимых изменений, желательно всегда закреплять старший номер версии. Начиная с версии Terraform 0.14.0, отпала необходимость фиксировать младшие номера версий или номера исправлений провайдеров, так как это происходит автоматически благодаря *файлу блокировки*. При первом запуске `terraform init` Terraform создает файл `.terraform.lock.hcl` и записывает в него следующую информацию.

- *Точную версию каждого используемого провайдера.* Если сохранить файл `.terraform.lock.hcl` в системе управления версиями (что и нужно сделать!), то в будущем команда `terraform init`, запущенная на этом или любом другом компьютере, загрузит ту же версию каждого провайдера. Вот почему можно не закреплять младший номер версии и номер исправления в блоке `require_providers` — это поведение обеспечивается по умолчанию. Если вы решите явно обновить версию провайдера, то для этого достаточно будет обновить ограничение версии в блоке `require_providers` и запустить команду `terraform init -upgrade`. Terraform загрузит новые версии провайдеров, соответствующие ограничениям, и обновит файл `.terraform.lock.hcl`; вам останется только проверить эти обновления и отправить их в систему управления версиями.
- *Контрольные суммы для каждого провайдера.* Terraform записывает контрольные суммы для всех загруженных провайдеров, и если контрольная сумма изменится, то следующий же запуск `terraform init` сообщит об ошибке. Эта мера безопасности гарантирует, что никто не сможет незаметно подменить код провайдера вредоносным кодом. Если провайдер имеет криптографическую подпись (а большинство провайдеров, официально поддерживаемых компанией HashiCorp, имеют такую подпись), то Terraform дополнительно проверит и ее.



Использование файлов блокировки в нескольких операционных системах

По умолчанию Terraform записывает контрольные суммы только для платформы, где запускалась команда `init`: например, если `init` запускалась в Linux, то Terraform запишет в `.terraform.lock.hcl` лишь контрольные суммы двоичных файлов провайдеров для Linux. Если сохранить этот файл в системе управления версиями, а затем получить его и запустить `init` в Mac, то `init` сообщит об ошибке, потому что контрольные суммы для Mac отсутствуют в `.terraform.lock.hcl`. Если ваша команда работает с несколькими операционными системами, придется запустить команду `terraform providers lock` в каждой из них, чтобы записать контрольные суммы для каждой используемой платформы:

```
terraform providers lock \  
-platform=windows_amd64 \  
-platform=darwin_amd64 \  
-platform=darwin_arm64 \  
-platform=linux_amd64
```

64-разрядная версия Windows
64-разрядная версия macOS
64-разрядная версия macOS (ARM)
64-разрядная версия Linux

И наконец, рассмотрим третий тип зависимостей — зависимости от версий модулей. Как обсуждалось в разделе «Версионирование модулей» главы 4, я настоятельно рекомендую фиксировать версии модулей, используя в параметре `source` URL (а не пути к локальным файлам) с параметром запроса `ref`, ссылающимся на тег Git:

```
source = "git@github.com:foo/modules.git//services/hello-world-app?ref=v0.0.5"
```

При использовании таких URL Terraform всегда будет загружать одну и ту же версию модуля при каждом запуске `terraform init`.

Теперь, когда вы узнали, как версионировать зависимости вашего кода, поговорим о том, как версионировать сам код. Как вы уже видели в разделе «Версионирование модулей» главы 4, 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"  
}  
  
module "hello_world_app" {  
  # TODO: подставьте 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 = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

instance_type        = "t2.micro"
min_size              = 2
max_size              = 2
enable_autoscaling    = false
ami                   = data.aws_ami.ubuntu.id
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

```

Затем вернем доменное имя ALB в выходной переменной, объявленной 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 `terraform init` и `terraform apply`:

```
$ 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
```

Убедившись, что все работает хорошо, эту же версию (и, следовательно, этот же код) можно развернуть в других средах, включая промышленную. А если у вас возникнет проблема, вы сможете откатиться назад, развернув более старую версию.

Еще один способ выпуска модулей — их публикация в реестре Terraform. Публичный реестр Terraform находится по адресу <https://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 требований¹.

- Модуль должен находиться в общедоступном репозитории GitHub.
- Репозиторий должен иметь название вида `terraform-<PROVIDER>-<NAME>`, где `PROVIDER` — это имя провайдера, на которого рассчитан модуль (например, `aws`), а `NAME` — имя модуля (скажем, `vault`).
- Модуль должен иметь определенную структуру файлов: код Terraform должен находиться в корне репозитория, должен иметься файл `README.md` и должно соблюдаться соглашение об именовании файлов `main.tf`, `variables.tf` и `outputs.tf`.
- Репозиторий должен использовать теги Git и семантическое версионирование (`x.y.z`) выпусков.

Если ваш модуль отвечает всем этим требованиям, вы можете сделать его доступным для всех желающих. Для этого войдите в реестр Terraform с помощью своей учетной записи GitHub и опубликуйте свой модуль в веб-интерфейсе.

Terraform поддерживает специальный синтаксис для подключения модулей из реестра. Вместо длинных Git URL с малозаметными параметрами `ref` можно использовать в аргументе `source` более короткий URL реестра, указывая версию в отдельном аргументе `version`. Вот как это выглядит:

```
module "<NAME>" {
  source = "<OWNER>/<REPO>/<PROVIDER>"
  version = "<VERSION>"

  # (...)
}
```

`NAME` — это идентификатор модуля в вашем коде Terraform, `OWNER` — владелец репозитория на 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 подключения модуля RDS из реестра Terraform:

```
module "rds" {
  source = "terraform-aws-modules/rds/aws"
  version = "4.4.0"

  # (...)
}
```

¹ Все подробности о публикации модулей можно найти на странице <https://www.terraform.io/registry/modules/publish>.

Если вы клиент HashiCorp Terraform Cloud или у вас есть подписка на сервис Terraform Enterprise, то же самое вам будет доступно при работе с закрытым реестром Terraform в ваших закрытых Git-репозиториях. Это отличный способ распространения модулей внутри компании.

За границами модулей Terraform

Эта книга посвящена Terraform, однако для построения инфраструктуры промышленного уровня вам понадобятся и другие инструменты: Docker, Packer, Chef, Puppet и, конечно же, изолента, клей и рабочая лошадка мира DevOps старый добрый Bash-скрипт.

Большую часть этого кода можно разместить в папке `modules` рядом с кодом Terraform. Например, у вас может иметься папка `elements/packer`, содержащая шаблон Packer и некоторые скрипты на Bash для настройки образа AMI рядом с файлом `elements/asg-rolling-deploy` модуля Terraform, используемого для развертывания этого образа.

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 выделения ресурсов;
- средства выделения ресурсов совместно с `null_resource`;
- внешний источник данных.

Пройдемся по ним по очереди.

Средства выделения ресурсов

Для выполнения скриптов на локальном или удаленном компьютере во время запуска Terraform использует *средства выделения ресурсов*. Обычно это связано с развертыванием/очисткой ресурсов или управлением конфигурацией. Таких средств несколько, включая `local-exec` (выполняет скрипты в локальной

системе), `remote-exec` (выполняет скрипты на удаленном компьютере) и `file` (копирует файл на удаленный ресурс)¹.

Чтобы добавить в ресурс средства выделения, можно воспользоваться блоком `provisioner`. Например, ниже показано, как с помощью `local-exec` выполнить скрипт на локальном компьютере:

```
resource "aws_instance" "example" {
  ami      = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

Если выполнить для этого кода команду `terraform apply`, она выведет текст `Hello, World from`, 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 выделения `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 выделения `remote-exec` поддерживает протоколы SSH и WinRM.

Поскольку вы будете запускать сервер под управлением Linux (Ubuntu), вам придется использовать аутентификацию по SSH, а это значит, вам понадобится

¹ Полный список средств выделения ресурсов можно найти на странице <https://www.terraform.io/language/resources/provisioners/syntax>.

настроить SSH-ключи. Для начала создадим группу безопасности, которая разрешает входящие соединения через стандартный порт SSH 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 закрытый. Первый следует выгрузить в AWS, а второй сохранить в надежном месте, откуда его сможет получить Terraform. Но, чтобы проще было попробовать этот пример, используйте ресурс `tls_private_key`, который сгенерирует закрытый ключ автоматически:

```
# Для простоты сгенерируем закрытый 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 закрытый ключ хранится в файле состояния Terraform, что не очень хорошо для промышленного использования, но подойдет для этого примера. Теперь выгрузим 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:

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

resource "aws_instance" "example" {
  ami = data.aws_ami.ubuntu.id
```

```

instance_type      = "t2.micro"
vpc_security_group_ids = [aws_security_group.instance.id]
key_name           = aws_key_pair.generated_key.key_name
}

```

Практически весь этот код должен показаться вам знакомым: он использует источник данных `aws_ami` для поиска образа AMI Ubuntu и ресурс `aws_instance` для развертывания этого образа на `t2.micro` и связывают этот сервер EC2 с группой безопасности, созданной ранее. Единственное нововведение — использование атрибута `key_name` в ресурсе `aws_instance`, который предписывает AWS связать открытый ключ с сервером EC2. AWS добавит этот ключ в его файл `authorized_keys`, благодаря чему вы сможете зайти на этот сервер по SSH с соответствующим закрытым ключом.

Теперь добавим `remote-exec` в ресурс `aws_instance`:

```

resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  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)\""]
  }
}

```

Как видите, использование `remote-exec` почти не отличается от `local-exec`. Единственное различие: вместо одной команды (параметр `command` в `local-exec`) здесь в параметре `inline` определяется список команд. 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           = data.aws_ami.ubuntu.id
  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 именем пользователя "ubuntu" (так по умолчанию называется пользователь root в образах AMI с Ubuntu) и автоматически сгенерированным закрытым ключом. Если выполнить для этого кода команду `terraform apply`, она выведет 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...
```

```
(... эти строки повторяются еще несколько раз ...)
```

```
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.
```

Средство выделения `remote-exec` не знает, когда именно сервер EC2 завершит загрузку и будет готов принимать соединения, поэтому периодически пытается подключиться по SSH, пока это не получится или пока не истечет время ожидания. Время ожидания по умолчанию равно пяти минутам, но его можно настроить. Рано или поздно соединение будет установлено, и вы получите от сервера ответ `Hello, World`.

Следует отметить, что по умолчанию средство выделения действует *во время создания ресурсов*, то есть выполняется после запуска команды `terraform apply` и только во время начального создания ресурса. Оно не будет срабатывать при всех последующих запусках `terraform apply`, поэтому его основное применение — начальное развертывание кода. Если в средстве выделения ресурсов указать `when = destroy`, оно будет действовать *во время их удаления*: то есть после запуска `terraform destroy` и непосредственно перед удалением ресурса.

Вы можете указать несколько средств выделения для одного и того же ресурса, и Terraform запустит их по очереди, сверху вниз. Чтобы объяснить Terraform, как обрабатывать ошибки, полученные в результате выделения, 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.

СРАВНЕНИЕ СРЕДСТВ ВЫДЕЛЕНИЯ РЕСУРСОВ И ПОЛЬЗОВАТЕЛЬСКИХ ДАННЫХ

Вы познакомились с двумя разными способами выполнения скриптов на сервере с помощью Terraform: один с использованием средства выделения `remote-exec`, а другой — с применением скрипта пользовательских данных. Последний мне кажется более полезным по нескольким причинам.

- `remote-exec` требует открыть доступ 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 AWS API (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 по событиям автомасштабирования или автовосстановления) выполнят скрипт во время своей загрузки. Средства выделения ресурсов действуют только во время работы Terraform, но не с ASG.
- Скрипт пользовательских данных можно видеть в консоли EC2 (выберите сервер и щелкните по 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 выделения ресурсов.

Единственное реальное преимущество использования `remote-exec` для выполнения кода на сервере EC2 — отсутствие ограничения на объем кода, тогда как для скриптов пользовательских данных задано ограничение в размере 16 Кбайт.

Средства выделения ресурсов с `null_resource`

Средства выделения можно определять только внутри ресурса, но иногда желательно обойтись без привязки к конкретному ресурсу. Эту задачу можно решить с помощью пустого ресурса с именем `null_resource`, который ведет себя подобно обычному ресурсу Terraform, но при этом ничего не создает. Определив средство выделения в `null_resource`, можно запустить свой скрипт как часть 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 `triggers`, который принимает ассоциативный массив с ключами и значениями. При любом изменении значений ресурс `null_resource` будет создаваться заново, запуская тем самым все сред-

ства выделения, которые в нем определены. Например, чтобы организовать запуск средства выделения внутри `null_resource` при каждом запуске команды `terraform apply`, можно воспользоваться встроенной функцией `uuid()`, которая при каждом вызове внутри аргумента `triggers` 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 `terraform apply` будет запускаться средство выделения `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 выделения 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, как описано ниже.

- 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 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, переданное через аргумент `query`, должно вернуться без изменений в выходном атрибуте `result`. Вот что получится, если выполнить `terraform apply` 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, отправленный 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 нужно обратиться 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, а это значит, что вы не сможете развернуть этот модуль Terraform из Windows.

Резюме

Итак, вы познакомились со всеми ингредиентами, необходимыми для создания кода Terraform промышленного уровня. В следующий раз, когда вы начнете работать над новым модулем, используйте такой процесс.

1. Пройдитесь по списку задач для подготовки инфраструктуры промышленного уровня, представленному в табл. 8.2, и определите, какие пункты вы будете реализовывать, а какие — нет. Сопоставьте полученный результат с табл. 8.1, чтобы предоставить начальству примерные сроки выполнения.
2. Создайте папку `examples` и сначала напишите демонстрационный код. Выработайте на его основе максимально удобный и опрятный API для своих модулей. Создайте по одному примеру для каждого существенного 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 компокуемых модулей. Используйте для этого Terraform в сочетании с другими инструментами, такими как Docker, Packer и Bash. Не забудьте закрепить версии всех ваших зависимостей, включая ядро Terraform, провайдеры и модули.
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 9.

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 ставки растут, 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 поломкам.

Если вы управляете своей инфраструктурой как кодом, 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 велика вероятность, 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 последнее.

- 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 <https://github.com/brikis98/terraform-up-and-running-code>.

Ручные тесты

При размышлении 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'
    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
```

Этот код вернет ответ **200 OK** с телом `Hello, World` для URL `/`; ответ **201** с телом `json` для URL `/api`; для любого другого URL 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 по нажатию Ctrl+C
trap 'INT' do server.shutdown end

# Запускаем сервер
server.start
end
```

Если запустить этот файл в терминале, он запустит веб-сервер, прослушивающий порт 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
```

Основы ручного тестирования

Как будет выглядеть подобного рода ручное тестирование кода 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
}

# (...)

```

Сравнив этот листинг с кодом на Ruby, можно заметить одно довольно очевидное отличие: AWS ALB, целевые группы, прослушиватели, группы безопасности и любые другие ресурсы нельзя развернуть на собственном компьютере.

Из этого следует *ключевой вывод о тестировании № 1*: тестирование кода Terraform не может выполняться локально.

Это относится не только к Terraform, но и к большинству инструментов IaC. Единственный практичный способ выполнить ручное тестирование в Terraform — развернуть код в настоящем окружении (то есть в AWS). Иными словами, запуск вручную команд `terraform apply` и `terraform destroy`, чем вы занимались на страницах этой книги, — 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 8). Чтобы протестировать модуль `alb`, проще всего воспользоваться демонстрационным кодом, который вы создали в `examples/alb`:

```

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

module "alb" {
    source = "../modules/networking/alb"

    alb_name     = "terraform-up-and-running"
    subnet_ids = data.aws_subnets.default.ids
}

```

Чтобы развернуть этот пример, нужно выполнить команду `terraform apply`, 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
```


По окончании 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 иных методов 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 принимает 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.

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 захочется¹.

¹ 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.

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 захлестят 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 `terraform destroy`. Возможно, удастся найти инструменты для очистки лишних или старых ресурсов, которые можно запускать на регулярной основе (скажем, с помощью `cron`), такие как `cloud-nuke` (<https://bit.ly/2OIgM9r>) и `aws-nuke` (<https://bit.ly/2ZB8IOe>).

Например, распространенной практикой является запуск `cloud-nuke` из задания `cron` один раз в день в каждой изолированной среде для удаления всех ресурсов, возраст которых превышает 48 часов. Этот подход базируется на предположении, что любая инфраструктура, которую разработчик запустил для ручного тестирования, становится ненужной через пару дней:

```
$ cloud-nuke aws --older-than 48h
```

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 листаете книгу, удовлетворяя начальное любопытство, то можете пропустить его. Но если вы действительно хотите научиться тестировать свой инфраструктурный код, закатайте рукава и готовьтесь к настоящему программированию! Код на Ruby можно не запускать (он нужен лишь для того, чтобы сформировать общее понимание происходящего), а вот код на Go следует записывать и выполнять как можно активней.

Идея автоматического тестирования состоит в том, что для проверки поведения настоящего кода пишутся тесты. В главе 10 вы узнаете, что с помощью сервера CI эти тесты можно запускать после каждой отдельной фиксации. Если они не будут пройдены, фиксацию сразу же можно откатить 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 макетами*. Это позволяет полностью контролировать их поведение (например, макет базы данных может возвращать ответ, прописанный вручную) и помогать 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 взаперемешку с макетами: например, если вы тестируете участок приложения, который обращается к базе данных, стоит тестировать его с настоящей БД, а другие зависимости, скажем систему аутентификации, заменить макетами.
- *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: скажем, Selenium может автоматизировать взаимодействие с вашим продуктом через браузер. В сквозном тестировании везде используются реальные компоненты без каких-либо макетов, 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
```

Написать модульный тест, напрямую вызывающий метод `dog_GET`, довольно непросто, потому что для этого вам придется создать экземпляры реальных объектов `WebServer`, `request` и `response` или их тестовые макеты. И то и другое требует немалых 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 входные значения.* Класс `Handlers` не зависит от `HTTPServer`, `HTTPRequest` или `HTTPResponse`. Все входные данные он принимает через параметры, например `path`, через который передается строка с URL.

- *Простые выходные значения.* Вместо записи значений в 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 и возвращает простые значения, обычно легче понимать, изменять и тестировать. 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 простым и понятным. Кроме того, это облегчает тестирование. Вот как выглядит модульный тест, проверяющий каждую конечную точку в `Handlers`:

```
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

  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
end
```

Вот как запустить эти тесты:

```
$ 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 0,000572 секунды вы можете проверить, работает ли код вашего веб-сервера так, как ожидалось. В этом сила модульных тестов: быстрые результаты, которые помогают укрепить уверенность в своем коде.

Модульное тестирование кода Terraform

Возможно ли эквивалентное модульное тестирование кода Terraform? Для начала нужно определиться с тем, что в мире Terraform считается *модулем*. Ближайший аналог отдельной функции или класса — обобщенный модуль, такой как `alb` из главы 8. 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 AWS API 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 тестирования¹.

Это подводит нас к *ключевому выводу о тестировании № 3*: вы не можете проводить *чистое* тестирование кода Terraform.

¹ В ограниченных случаях конечные точки, которые использует Terraform, можно переопределить. Например, можно переопределить конечную точку, с помощью которой Terraform взаимодействует с AWS, и заменить ее макетом, реализованным с помощью LocalStack (<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/guides/custom-service-endpoints#localstack>). Это нормальное решение для небольшого количества конечных точек, но код Terraform обычно выполняет сотни разных API-вызовов к исходному провайдеру, и создавать макеты для каждого из них было бы непрактично. Более того, если вы пошли на создание этих макетов, это вовсе не означает, что итоговый модульный тест даст вам достаточно уверенности в корректной работе вашего кода. Если вы создадите макеты для конечных точек 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?

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 `terraform apply` 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 `terraform destroy` 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 этот 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 (<https://terratest.gruntwork.io/>), 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 `terraform apply`, убеждаетесь, что он работает, и затем выполняете в конце `terraform destroy`, чтобы очистить ресурсы.

Чтобы использовать Terratest, вам нужно сделать следующее.

1. Установить Go (<https://golang.org/doc/install>), как минимум версию 1.13.
2. Создать папку для тестов, например, с именем `test`.

3. Запустите команду `go mod init <NAME>` в только что созданной папке, где `NAME` — имя, которое будет использоваться для этого набора тестов, обычно в формате `github.com/<ORG_NAME>/<PROJECT_NAME>` (например, `go mod init github.com/brikis98/terraform-up-and-running`). При этом должен быть создан файл `go.mod`, который используется для отслеживания зависимостей вашего кода Go.

Чтобы быстро проверить, верно ли сконфигурировано ваше окружение, создайте в своей новой папке файл `go_sanity_test.go` со следующим содержимым:

```
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 `go test`:

```
$ go test -v
```

Флаг `-v` означает `verbose` («подробно»). Он делает так, чтобы тест генерировал подробный вывод. Вы должны получить примерно такой результат:

```
=== RUN    TestGoIsWorking
If you see this text, it's working!
--- PASS: TestGoIsWorking (0.00s)
PASS
ok github.com/brikis98/terraform-up-and-running-code 0.192s
```

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).

Следующий этап автоматического тестирования — выполнение команд `terraform init` и `terraform apply`, 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 это 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 `terraform init` и `terraform apply` и проваливает тест, если эти команды не завершаются успешно (например, из-за проблем в 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 надо получить доменное имя развернутого балансировщика. К счастью, пример `alb` возвращает его в 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 выходных переменных 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 значение выходной переменной с указанным именем или проваливает тест, если получает пустое значение или не находит такой переменной. На основе полученного значения предыдущий код формирует URL, используя встроенную в Go функцию `fmt.Sprintf` (не забудьте импортировать пакет `fmt`). Следующим шагом будет выполнение HTTP-запросов по этому URL-адресу (не забудьте импортировать `github.com/gruntwork-io/terratest/modules/http-helper`):

```
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,
    nil,
    expectedStatus,
    expectedBody,
    maxRetries,
    timeBetweenRetries,
)
}
```

Метод `http_helper.HttpGetWithRetry` отправит запрос HTTP GET по указанному адресу URL и проверит, содержит ли ответ ожидаемый код состояния и тело. Если это не так, он будет повторять попытки до достижения указанного максимального количества попыток через заданные интервалы времени между попытками. Если в какой-то момент он получит ожидаемый ответ, тест будет пройден. Если же после истечения максимального количества попыток ожидаемый ответ так и не пришел, тест считается проваленным. Такая логика повторных попыток очень распространена при тестировании инфраструктуры, поскольку обычно между завершением `terraform apply` и полной готовностью развернутой инфраструктуры обычно проходит какое-то время (необходимое для прохождения проверок работоспособности, распространения обновлений DNS и т. д.), а так как заранее не известно, сколько времени это займет, лучший вариант — повторять попытки, пока одна из них не окажется успешной или пока не истечет тайм-аут.

В конце теста нужно выполнить команду `terraform destroy`, чтобы очистить ресурсы. Для этого у Terratest есть вспомогательный метод: `terraform.Destroy`. Но если вызывать его в конце теста, то из-за любой программной ошибки выше по коду (например, в `HttpGetWithRetry` из-за неправильной конфигурации ALB) тест может завершиться, не доходя до него, в результате чего развернутая инфраструктура не будет удалена.

Таким образом, вам нужно гарантировать выполнение `terraform.Destroy` в *любом случае*, 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 по 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 ресурсы по окончании 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,
    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.



Версия Terratest

Тестовый код в этой книге написан с использованием Terratest *v0.39.0*. Terratest являлся основным инструментом до версии 1.0.0, поэтому новые версии могут содержать обратно несовместимые изменения. Чтобы тестовые примеры в этой книге работали так, как написано, я рекомендую устанавливать Terratest именно версии *v0.39.0*, а не более новой. Для этого откройте файл `go.mod` и добавьте в конец строку:

```
require github.com/gruntwork-io/terratest v0.39.0
```

Поскольку это совершенно новый проект Go, необходимо выполнить разовое действие — указать Go, что следует загрузить зависимости (включая Terratest). Самый простой способ сделать это на данном этапе — запустить команду:

```
go mod tidy
```

Она загрузит все ваши зависимости и создаст файл `go.sum`, чтобы зафиксировать используемые вами версии.

Далее, поскольку этот тест развертывает инфраструктуру в AWS, перед его выполнением следует аутентифицировать его, используя свои учетные данные (см. врезку «Способы аутентификации» в главе 2). Как упоминалось выше в этой главе, ручное тестирование следует проводить в изолированной учетной записи. Это вдвойне справедливо для автоматических тестов, поэтому советую настроить 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 вы сможете без опаски 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 `-timeout 30m` в команде `go test`. Go по умолчанию ограничивает время работы тестов десятью минутами; когда это время истекает, выполнение теста принудительно завершается, в результате чего тест не просто проваливается, но даже не доходит до кода очистки (`terraform destroy`). 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 вывод, но, если внимательно почитать, можно заметить все его ключевые моменты.

1. Выполнение `terraform init`.
2. Выполнение `terraform apply`.
3. Чтение выходной переменной с помощью `terraform apply`.
4. Многократная отправка HTTP-запросов к ALB.
5. Выполнение `terraform destroy`.

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 довольно быстро, и в результате вы получаете уверенность в том, что ваш код работает как следует.

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 конечную точку `/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 конечная точка, 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 `/web-service`, посылая HTTP-запрос типа GET к `example.org` и возвращая полученный ответ. Если обратиться к этой конечной точке 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`). Это влечет ряд отрицательных последствий.

- Если в 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, находящиеся 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 URL и предоставляет метод `proxy` для отправки запроса HTTP GET на этот адрес и возврата полученного ответа. Мы можем сделать так, чтобы класс `Handlers` принимал 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 конечная точка, которая вызывает веб-сервис
      @web_service.proxy
    end
  end
end
```



```
    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 макет 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 теста, добавленные ранее, по-прежнему успешно выполняются, вы можете быть уверены, что в ходе рефакторинга ничего не сломали.

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"
}

module "hello_world_app" {
  source = ".././../modules/services/hello-world-app"

  server_text = "Hello, World"
  environment = "example"

  db_remote_state_bucket = "(ИМЯ_ВАШЕЙ_КОРЗИНЫ)"
  db_remote_state_key     = "examples/terraform.tfstate"

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
  ami                = data.aws_ami.ubuntu.id
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners     = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

Проблема с зависимостями сразу бросается в глаза, стоит лишь взглянуть на параметры `db_remote_state_bucket` и `db_remote_state_key`: модуль `hello-world-app` предполагает, что вы уже развернули модуль `mysql`, и требует, предоставить в этих параметрах информацию о корзине S3, в которой `mysql` хранит свое состояние. 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.

Первым делом нужно прояснить, от чего зависит ваш модуль. Для этого все источники данных и ресурсы, представляющие внешние зависимости, можно вынести в отдельный файл `dependencies.tf`. Вот как 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_subnets" "default" {
  filter {
    name     = "vpc-id"
    values   = [data.aws_vpc.default.id]
  }
}
```

Это поможет 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 извне, 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 значение по умолчанию, которое равно **null**.

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 переменным 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 такого решения 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
ami                = data.aws_ami.ubuntu.id
}

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 строкой. И если типы когда-нибудь поменяются и перестанут совпадать, Terraform сразу же выдаст сообщение об ошибке, 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 через входные переменные, переменные `db_remote_state_bucket` и `db_remote_state_key` должны теперь быть опциональными, поэтому присвойте им `null` как значение 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
}
```

Далее используйте параметр `count`, чтобы создать в файле `modules/services/hello-world-app/dependencies.tf` соответствующие источники данных, если какие-то входные переменные будут иметь значение `null`:

```
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_subnets" "default" {
  count = var.subnet_ids == null ? 1 : 0
  filter {
    name = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}

```

Теперь нужно обновить все ссылки на эти источники данных, 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 в источнике `aws_subnets` используйте `local.vpc_id`:

```

data "aws_subnets" "default" {
  count = var.subnet_ids == null ? 1 : 0
  filter {
    name = "vpc-id"
    values = [local.vpc_id]
  }
}

```

Затем присвойте параметр `subnet_ids` из модуля `alb` параметру `local.subnet_ids`:

```
module "alb" {
  source = "../../networking/alb"

  alb_name    = "hello-world-${var.environment}"
  subnet_ids = local.subnet_ids
}
```

Обновите также модуль `asg`: присвойте параметру `subnet_ids` значение `local.subnet_ids`, а в `user_data` обновите переменные `db_address` и `db_port`, чтобы они получали данные из `local.mysql_config`:

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"

  cluster_name = "hello-world-${var.environment}"
  ami          = var.ami
  instance_type = var.instance_type

  user_data = templatefile("${path.module}/user-data.sh", {
    server_port = var.server_port
    db_address  = local.mysql_config.address
    db_port     = local.mysql_config.port
    server_text = var.server_text
  })

  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
}
```

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
  ami              = data.aws_ami.ubuntu.id
}

```

Теперь в модульном тесте присвойте переменной `mysql_config` любое значение по своему выбору. Создайте модульный тест `test/hello_world_app_example_test.go`:


```
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)

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetry(
        t,
        url,
        nil,
        maxRetries,
        timeBetweenRetries,
        func(status int, body string) bool {
            return status == 200 &&
                strings.Contains(body, "Hello, World")
        },
    )
}
```

Этот модульный тест почти идентичен тому, который мы создали для примера с `alb`. Он имеет всего два отличия.

- Параметр `TerraformDir` указывает на пример `hello-world-app` (не забудьте обновить путь в соответствии со своей файловой системой).
- Вместо `http_helper.HttpGetWithRetry` для проверки ответа `404` тест использует метод `http_helper.HttpGetWithRetryWithCustomValidation`, проверяющий код ответа `200` и тело, содержащее текст `Hello, World`. Это связано с тем, что скрипт в пользовательских данных модуля `hello-world-app` возвращает ответ `200 OK`, который включает не только текст, возвращаемый сервером, но и разметку `HTML`.

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, размещенной в оперативной памяти, которая запускается на время тестирования.

Выполните команду `go test` 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
```

В случае успеха тест выполнит `terraform apply` и начнет слать HTTP-запросы балансировщику нагрузки. Как только придет ответ, он выполнит команду `terraform destroy`, 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 разделе вы запускали тесты по одному, используя команду `go test` 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 запускают 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",
    }

    // (...)
}

```

Если теперь запустить `go test`, она выполнит тесты параллельно. Но есть одна загвоздка: некоторые ресурсы, создаваемые тестами (например, ASG, группа безопасности и ALB), имеют одинаковые имена. В результате из-за конфликта имен тесты потерпят неудачу. 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 так, 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_subnets.default.ids
}
```

Теперь откройте файл `test/alb_example_test.go` и присвойте этой переменной уникальное значение:

```
package test

import (
    "fmt"
    "github.com/stretchr/testify/require"

    "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()),
        },
    }

    // (...)
}
```

Этот код присваивает переменной `alb_name` значение `test-<RANDOM_ID>`, где `RANDOM_ID` — случайный идентификатор, возвращаемый вспомогательным методом `random.UniqueId()` из Terratest. Этот 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 случаен для того, чтобы сделать конфликты крайне маловероятными (62^6 — это более 56 миллиардов комбина-

ций). 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
  ami              = data.aws_ami.ubuntu.id
}
```

И присвойте переменной `environment` в файле `test/hello_world_app_example_test.go` значение, которое включает вызов `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 выполнения самого медленного теста в вашем наборе (а не совокупное время работы всех тестов, как в случае с последовательным запуском).

Обратите внимание, что по умолчанию количество тестов, которые будут выполняться параллельно, равно количеству процессоров на вашем компьютере. То есть, если у вас только один процессор, по умолчанию тесты все равно будут выполняться последовательно, а не параллельно. Вы можете переопределить это поведение, настроив переменную окружения `GOMAXPROCS` или передав аргумент `-parallel` команде `go test`. Например, вот как выглядит команда, заставляющая Go запускать до двух тестов параллельно:

```
$ go test -v -timeout 30m -parallel 2
```



Параллельное выполнение тестов в одной и той же папке

Еще один аспект параллелизма, который следует учитывать, — 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 попытаетесь 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. В 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 интеграционному 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.
3. Убедиться в получении полученных ожидаемых ответов.

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` настраивает веб-сервер на прослушивание порта 8000, запускает его в фоновом потоке (чтобы он не блокировал выполнение теста), посылает 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 конечной точки /:

```
def test_integration_hello
  do_integration_test('/', lambda { |response|
    assert_equal(200, response.code.to_i)
  })
end
```

```

    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 лямбда-выражение (вложенную функцию), которая проверяет, равны ли код и тело ответа `200 OK` и соответственно `Hello, World`. Интеграционные тесты для других конечных точек выглядят аналогично. Давайте запустим их все:

```

$ 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. В предыдущем разделе вы 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 тестовой среды (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 тестовой среды все тесты следует запускать от имени изолированного тестового пользователя. Если в ваших модулях есть код, который специально прописан для тестовой среды, самое время сделать его конфигурируемым, чтобы вы могли внедрять тестовые значения. В частности, добавьте в файл `live/stage/data-stores/mysql/variables.tf` новую входную переменную `db_name` для передачи имени базы данных:


```
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)
}
```

Тест выполняет следующие действия: развертывает `mysql` и `hello-world-app`, проверяет приложение, удаляет `hello-world-app` (выполняется в конце благодаря `defer`) и в завершение удаляет `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_username": "admin",
            "db_password": "password",
        },
    },
}

```

Пока ничего нового: код передает методу `terraform.Options` заданную папку и устанавливает переменные `db_name`, `db_username` и `db_password`.

Дальше нужно разобраться с тем, где модуль `mysql` будет хранить свое состояние. До сих пор конфигурация `backend` содержала значения, прописанные вручную:

```

backend "s3" {
    # Подставьте имя своей корзины!
    bucket      = "terraform-up-and-running-state"
    key         = "stage/data-stores/mysql/terraform.tfstate"
    region      = "us-east-2"

    # Подставьте имя 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 тестовой среды! Одно из обходных решений — использовать рабочие области (как обсуждалось в подразделе «Изоляция через рабочие области» в главе 3), но для этого все равно нужен доступ к корзине S3, принадлежащей учетной записи тестовой среды, 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» главы 3. 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:

```

backend "s3" {
}

```

При развертывании модуля `mysql` в настоящей тестовой среде 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` можно 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_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_username": "admin",
            "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 одну корзину S3, так как параметр `key` (путь внутри корзины) 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 тестовой среде. 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
    ami              = data.aws_ami.ubuntu.id
}
```

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 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,
        nil,
        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 -v "TestHelloWorldAppStage"
```

```
(...)
```

```
PASS
```

```
ok  terraform-up-and-running  795.63s
```

Превосходно! 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* будет непросто, потому что основное время занимает развертывание RDS, ASG, ALB и так далее в AWS. 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 `terraform apply` для модуля `mysql`.
2. Запустить `terraform apply` для модуля `hello-world-app`.
3. Выполнить проверку и убедиться в том, что все работает.
4. Выполнить `terraform destroy` для модуля `hello-world-app`.
5. Выполнить `terraform destroy` 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 шаг за шагом вносите изменения в свой код, выполнять все стадии не обязательно. 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 `terraform apply` для модуля `mysql`.
2. Запустить `terraform apply` для модуля `hello-world-app`.
3. Переход к пошаговой разработке:
 - а) внести изменение в модуль `hello-world-app`;
 - б) повторно выполнить `terraform apply` 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 3а.
4. Выполнить `terraform destroy` для модуля `hello-world-app`.
5. Выполнить `terraform destroy` 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 выполнить, а какие пропустить.

Terratest имеет встроенную поддержку этой стратегии в виде пакета `test_structure`. 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 тестирования сохраняет тестовые данные на диск, чтобы их можно было прочитать во время последующих выполнений. Попробуем применить эту стратегию к примеру `test/hello_world_integration_test.go`. Сначала набросаем каркас теста, а затем наполним его внутренними методами:

```
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 `dbOps`, 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 `go test` вам захочется 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 новое поведение заключается в вызове методов `test_structure.SaveTerraformOptions` и `test_structure.SaveTerraformOptions` для загрузки `dbOpts` с диска и соответственно сохранения `helloOpts` на диск. 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 соответственно `SKIP_teardown_app`. 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 `terraform apply` 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, как быстро теперь работает каждый из этих тестов: вместо 10–15 минут тестирование каждого нового изменения 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_tearardown_app' environment variable is not set, so executing stage 'teardown_app'.

(...)

The 'SKIP_tearardown_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 и обрабатывать 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 получал такую 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 подобными ошибками, можно 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 5 секунд
        // между попытками, для известных ошибок
        MaxRetries: 3,
        TimeBetweenRetries: 5 * time.Second,
        RetryableTerraformErrors: map[string]string{
            "RequestError: send request failed": "Throttling issue?",
        },
    }
}
```

Аргументы `RetryableTerraformErrors` можно передать ассоциативный массив с известными ошибками, требующими 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 `terraform apply`, проводятся некоторые проверки, а затем все очищается с помощью `terraform destroy`. Но на практике такой подход применяют редко. Это связано с *пирамидой тестирования*, которую вы можете видеть на рис. 9.1.

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.



Рис. 9.1. Пирамида тестирования

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 основные причины.

- *Слишком медленно.* Развертывание целой инфраструктуры с нуля с последующим ее удалением занимает очень много времени: порядка нескольких часов. Наборы тестов выполняются так долго, что приносят относительно мало пользы, просто потому, что обратная связь слишком замедляется. Такие тесты обычно запускаются 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, у 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 %. А если тест развертывает два ресурса? Для его успешного прохождения оба ресурса должны быть развернуты без нерегулярных ошибок. Чтобы просчитать эти шансы, вероятности нужно умножить: $99,9 \times 99,9 = 99,8 \%$. В случае с тремя ресурсами шансы равны $99,9 \times 99,9 \times 99,9 = 99,7 \%$. 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 ста 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 достаточно стабильным, 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 одна-две 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 с именем `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 стратегию сквозного тестирования в сторону инкрементальных изменений, вы уменьшите количество ресурсов, которые развертываются на время проверки, с нескольких сотен до небольшой 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 сквозного тестирования имеет огромное преимущество: вы можете убедиться не только в корректности работы своей инфраструктуры, но и в том, что *процесс ее развертывания* работает как следует.

Другие подходы к тестированию

Большая часть этой главы посвящена автоматическим тестам на основе Terratest, но существует два других подхода к тестированию, которые не помешает иметь в своем арсенале:

- статический анализ;
- тестирование плана;
- тестирование сервера.

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 вместе. По очереди рассмотрим эти новые категории.

Статический анализ

Статический анализ — самый простой вид тестирования кода Terraform: вы просматриваете и анализируете код, не выполняя его. В табл. 9.1 перечислены некоторые инструменты статического анализа, поддерживающие Terraform, и проводится их сравнение с точки зрения популярности и зрелости на основе статистики, которую я собрал на GitHub в феврале 2022 года.

Таблица 9.1. Сравнение популярных инструментов статического анализа для Terraform

	terraform validate	tfsec	tflint	Terrascan
Краткое описание	Встроенная команда Terraform	Помогает выявлять потенциальные проблемы безопасности	Линтер, подключаемый Terraform	Помогает выявлять несоответствия и потенциальные проблемы безопасности
Лицензия	(как и у Terraform)	MIT	MPL 2.0	Apache 2.0
Компания-разработчик	(как и у Terraform)	Aqua Security	(нет)	Accurics
Рейтинг (число звезд)	(как и у Terraform)	3874	2853	2768
Участников проекта	(как и у Terraform)	96	77	63
Первый выпуск	(как и у Terraform)	2019	2016	2017
Последний выпуск	(как и у Terraform)	v1.1.2	v0.34.1	v1.13.0
Встроенные проверки	Только проверка синтаксиса	AWS, Azure, GCP, Kubernetes, Digital Ocean, и т. д.	AWS, Azure и GCP	AWS, Azure, GCP, Kubernetes и т. д.
Возможность реализации своих проверок	Не поддерживается	В формате YAML или JSON	В форме плагинов на Go	На языке Rego

Самый простой из этих инструментов — `terraform validate` — команда, встроенная в Terraform и способная выявить проблемы с синтаксисом. Например, если вы забыли установить параметр `alb_name` в файле `example/alb` и запустили `validate`, то в ответ получите примерно такое сообщение:

```
$ terraform validate
```

```
Error: Missing required argument
   on main.tf line 20, in module "alb":
   20: module "alb" {
      The argument "alb_name" is required, but no definition was found.
```

Обратите внимание, что `validate` ограничивается исключительно проверкой синтаксиса, тогда как другие инструменты позволяют применять другие виды

политик. Например, с помощью инструментов `tfsec` и `tflint` можно обеспечить соблюдение политик, таких как:

- группы безопасности не могут быть слишком открытыми, например, не должны содержать правил для входящего трафика, разрешающих доступ со всех IP-адресов (блок CIDR `0.0.0.0/0`);
- все серверы EC2 должны следовать определенному соглашению о тегах.

Идея заключается в *определении политик в виде кода* — возможности определить требования к безопасности и надежности в виде кода. В следующих нескольких разделах вы увидите несколько других инструментов, продвигающих идею «политика как код».

Достоинства инструментов статического анализа

- Работают быстро.
- Просты в использовании.
- Стабильны (не имеют хрупких тестов).
- Не требуют выполнять аутентификацию у реального провайдера (например, с реальной учетной записью AWS).
- Не требуют развертывать реальные ресурсы и удалять их.

Недостатки инструментов статического анализа

- Обнаруживают ограниченный круг ошибок. В частности, они обнаруживают только ошибки, которые можно выявить путем статического чтения кода без его выполнения: синтаксические ошибки, ошибки типов и небольшое подмножество ошибок бизнес-логики. Например, вы сможете обнаружить нарушение политик для статических значений, таких как жестко запрограммированная группа безопасности, разрешающая входящий трафик из блока CIDR `0.0.0.0/0`, но не сможете обнаружить нарушения политик для динамических значений, таких как та же самая группа безопасности, но получающая блок CIDR через переменную или файл.
- Эти тесты не проверяют функциональность, поэтому могут выполняться успешно, а инфраструктура все равно не будет работать!

Тестирование плана

Другой способ протестировать код — запустить команду `terraform plan` и проанализировать ее вывод. Поскольку в этом случае выполняется код, такая проверка — больше, чем статический анализ, но меньше, чем модульное или

интеграционное тестирование, потому что код выполняется не полностью: в частности, `plan` выполняет шаги чтения (например, читает состояние, выполняет источники данных), но не выполняет шаги записи (например, не создает и не изменяет ресурсы). В табл. 9.2 перечислены некоторые инструменты для тестирования плана и проводится их сравнение с точки зрения популярности и зрелости на основе статистики, которую я собрал на GitHub в феврале 2022 года.

Таблица 9.2. Сравнение популярных инструментов тестирования плана для Terraform

	Terratest	Open Policy Agent (OPA)	HashiCorp Sentinel	Checkov	terraform-compliance
Краткое описание	Библиотека на Go для тестирования IaC	Универсальный механизм проверки политик	Инструмент «политика как код» для корпоративных инструментов HashiCorp	Инструмент «политика как код» для всех	BDD-фреймворк тестирования для Terraform
Лицензия	Apache 2.0	Apache 2.0	Коммерческая	Apache 2.0	MIT
Компания-разработчик	Gruntwork	Styra	HashiCorp	Bridgecrew	(нет)
Рейтинг (число звезд)	5888	6207	(закрытый исходный код)	3758	1104
Участников проекта	157	237	(закрытый исходный код)	199	36
Первый выпуск	2016	2016	2017	2019	2018
Последний выпуск	v0.40.0	v0.37.1	v0.18.5	2.0.810	1.3.31
Встроенные проверки	Нет	Нет	Нет	AWS, Azure, GCP, Kubernetes и т. д.	Нет
Возможность реализации своих проверок	На Go	На Rego	В Sentinel	На Python или YAML	В BDD

Поскольку вы уже знакомы с Terratest, кратко рассмотрим, как можно использовать этот инструмент для тестирования вывода, полученного применением

команды `plan` к коду в `examples/alb`. Запустив `terraform plan`, вы получите такой вывод:

Terraform will perform the following actions:

```
# module.alb.aws_lb.example will be created
+ resource "aws_lb" "example" {
  + arn                = (known after apply)
  + load_balancer_type = "application"
  + name               = "test-4Ti6CP"
  (...)
}
```

(...)

Plan: 5 to add, 0 to change, 0 to destroy.

Как протестировать этот вывод программно? Вот базовая структура теста, который использует вспомогательный метод `InitAndPlan` из `Terratest` для автоматического запуска `init` и `plan`:

```
func TestAlbExamplePlan(t *testing.T) {
    t.Parallel()

    albName := fmt.Sprintf("test-%s", random.UniqueId())

    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": albName,
        },
    }

    planString := terraform.InitAndPlan(t, opts)
}
```

Даже этот минимальный тест имеет определенную ценность — он подтверждает, что команда `plan`, которая проверяет синтаксис и работоспособность всех вызовов API чтения, выполнится успешно. Но вы можете пойти еще дальше, например проверить получение ожидаемого значения: `5 to add`, `0 to change`, `0 to destroy`. Сделать это можно с помощью вспомогательного метода `GetResourceCount`.

```
// Пример проверки ожидаемых значений счетчиков add/change/destroy,
// возвращаемых командой plan
resourceCounts := terraform.GetResourceCount(t, planString)
```

```
require.Equal(t, 5, resourceCounts.Add)
require.Equal(t, 0, resourceCounts.Change)
require.Equal(t, 0, resourceCounts.Destroy)
```

При желании можно выполнить еще более тщательную проверку: поместить выходные данные `plan` в структуру для дальнейшего анализа с помощью вспомогательной функции `InitAndPlanAndShowWithStructNoLogTempPlanFile`. Эта структура дает программный доступ ко всем значениям и изменениям, отраженным в выводе `plan`. Таким способом, например, можно проверить, включает ли вывод команды `plan` ресурс `aws_lb` по адресу `module.alb.aws_lb.example` и содержит ли атрибут `name` этого ресурса ожидаемое значение:

```
// Пример проверки конкретных значений в выводе plan
planStruct :=
    terraform.InitAndPlanAndShowWithStructNoLogTempPlanFile(t, opts)

alb, exists :=
    planStruct.ResourcePlannedValuesMap["module.alb.aws_lb.example"]
require.True(t, exists, "aws_lb resource must exist")

name, exists := alb.AttributeValues["name"]
require.True(t, exists, "missing name parameter")
require.Equal(t, albName, name)
```

Сильная сторона подхода к тестированию плана с использованием Terratest заключается в его чрезвычайной гибкости: вы можете написать произвольный код на Go и проверить все, что захотите. Но это же является и слабым местом, потому что из-за этого аспекта сложнее начать работу.

Некоторые команды предпочитают определять свои политики в виде кода на более декларативном языке. За последние несколько лет особую популярность завоевал Open Policy Agent (OPA) — инструмент проверки политик, позволяющий определять политики компании в виде кода на декларативном языке под названием Rego.

Например, многие компании устанавливают свою политику добавления тегов. Общая особенность кода Terraform — добавление тега `ManagedBy = terraform` к каждому ресурсу, управляемому Terraform. Вот простая политика `enforce_tagged.rego`, которую можно использовать для проверки этого тега:

```
package terraform

allow {
    resource_change := input.resource_changes[_]
    resource_change.change.after.tags["ManagedBy"]
}
```

Эта политика будет просматривать изменения в выводе команды `terraform plan`, извлекать тег `ManagedBy` и присваивать `true` переменной OPA с именем `allow`, если этот тег установлен, и `undefined` в противном случае.

Теперь рассмотрим следующий модуль Terraform:

```
resource "aws_instance" "example" {  
  ami      = data.aws_ami.ubuntu.id  
  instance_type = "t2.micro"  
}
```

Этот модуль не устанавливает необходимого тега `ManagedBy`. Как можно это определить с помощью ОРА?

Первый шаг — запустить `terraform plan` и сохранить вывод в файл:

```
$ terraform plan -out tfplan.binary
```

ОРА может анализировать только текст в формате JSON, поэтому следующим шагом нужно преобразовать файл плана в формат JSON с помощью команды `terraform show`:

```
$ terraform show -json tfplan.binary > tfplan.json
```

Наконец, можно запустить команду `opa eval`, чтобы проверить этот файл плана на соответствие политике `enforce_tagged.rego`:

```
$ opa eval \  
  --data enforce_tagging.rego \  
  --input tfplan.json \  
  --format pretty \  
  data.terraform.allow
```

```
undefined
```

Поскольку тег `ManagedBy` отсутствует, агент ОРА вывел `undefined`. Теперь попробуем установить тег `ManagedBy`:

```
resource "aws_instance" "example" {  
  ami      = data.aws_ami.ubuntu.id  
  instance_type = "t2.micro"  
  
  tags = {  
    ManagedBy = "terraform"  
  }  
}
```

Снова запустим `terraform plan`, `terraform show` и `opa eval`:

```
$ terraform plan -out tfplan.binary
```

```
$ terraform show -json tfplan.binary > tfplan.json
```

```
$ opa eval \  
  --data enforce_tagging.rego \  
  --input tfplan.json
```

```
--input tfplan.json \  
--format pretty \  
data.terraform.allow  
  
true
```

На этот раз команда `opa eval` вывела `true`, что означает соответствие заданной политике.

Используя такие инструменты, как ОРА, можно обеспечить соблюдение требований вашей компании, создав библиотеку таких политик и настроив конвейер CI/CD, который проверяет соответствие ваших модулей Terraform этим политикам после каждой фиксации.

Достоинства инструментов тестирования плана

- Работают быстро. Не так быстро, как чистый статический анализ, но намного быстрее, чем модульные или интеграционные тесты.
- Довольно просты в использовании. Не так просты, как чистый статический анализ, но намного проще, чем модульные или интеграционные тесты.
- Стабильны (несколько хрупких тестов). Не так стабильны, как чистый статический анализ, но гораздо стабильнее, чем модульные или интеграционные тесты.
- Не требуют развертывать реальные ресурсы и удалять их.

Недостатки инструментов тестирования плана

- Обнаруживают ограниченный круг ошибок. Они могут обнаружить больше ошибок, чем простой статический анализ, но не так много, как модульное и интеграционное тестирование.
- Требуют аутентификации у реального провайдера (например, в реальной учетной записи AWS). Это необходимо для работы команды `plan`.
- Эти тесты не проверяют функциональность, поэтому могут выполняться успешно, а инфраструктура все равно не будет работать!

Тестирование сервера

Существует множество инструментов тестирования, проверяющих правильность настройки серверов (включая виртуальные). Нет какого-либо общепринятого названия для подобных инструментов, поэтому я буду называть их инструментами *тестирования серверов*. Это не универсальные инструменты, и они не предназначены для тестирования тех или иных аспектов кода Terraform.

Фактически большинство этих инструментов изначально создавались для использования в комплексе с инструментами управления конфигурацией, такими как Chef и Puppet, полностью ориентированными на запуск серверов. Однако с ростом популярности Terraform их начали применять для проверки работоспособности запущенных серверов. В табл. 9.3 перечислены некоторые инструменты тестирования серверов и проводится их сравнение с точки зрения популярности и зрелости на основе статистики, которую я собрал на GitHub в феврале 2022 года.

Таблица 9.3. Сравнение популярных инструментов тестирования серверов

	inspec	serverspec	goss
Краткое описание	Фреймворк аудита и тестирования	RSpec-тесты для серверов	Простое и быстрое тестирование/проверка серверов
Лицензия	Apache 2.0	MIT	Apache 2.0
Компания-разработчик	Chef	(нет)	(нет)
Рейтинг (число звезд)	2472	2426	4607
Участников проекта	279	128	89
Первый выпуск	2016	2013	2015
Последний выпуск	v4.52.9	v2.42.0	v0.3.16
Встроенные проверки	Нет	Нет	Нет
Возможность реализации своих проверок	На предметном языке на основе Ruby	На предметном языке на основе Ruby	На YAML

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
```

Достоинства инструментов тестирования серверов

- Упрощают проверку определенных свойств серверов. DSL, предлагаемые этими инструментами, гораздо проще использовать для обычных проверок, чем делать все это с нуля.
- Позволяют создать библиотеку проверок политик. Поскольку каждая отдельная проверка выполняется быстро, как указано в предыдущем пункте, эти инструменты, как правило, хорошо подходят для проверки контрольного списка требований, особенно в отношении соответствия (например, соответствия PCI, соответствия HIPAA и т. д.).
- Могут обнаруживать многие типы ошибок. Поскольку на самом деле вам нужно запустить `apply`, чтобы проверить реальный работающий сервер, эти типы тестов выявляют гораздо больше типов ошибок, чем чистый статический анализ или тестирование плана.

Недостатки инструментов тестирования серверов

- Не такие быстрые. Эти тесты работают только на развернутых серверах, поэтому вам придется выполнить полноценную команду `apply` (и, возможно, `destroy`), что может занять много времени.
- Не такие стабильные (имеет место некоторая хрупкость тестов). Поскольку для тестирования приходится запускать `apply` и ждать развертывания реальных серверов, вы будете периодически сталкиваться с различными случайными проблемами, а иногда и с нестабильностью тестов.
- Требуют аутентификации у реального провайдера (например, в реальной учетной записи AWS). Это необходимо, чтобы команда `apply` могла развернуть серверы, плюс сами инструменты тестирования серверов нуждаются в дополнительных методах аутентификации (например, SSH) для подключения к тестируемым серверам.
- Необходимость развертывать/удалять реальные ресурсы, что требует времени и стоит денег.
- Они тщательно проверяют только работу серверов, но не других компонентов инфраструктуры.
- Эти тесты не проверяют функциональность, поэтому могут выполняться успешно, а инфраструктура все равно не будет работать!

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 не-исправным.

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 сами никогда не нашли бы (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 <https://github.com/brikis98/terraform-up-and-running-code>.

В этой главе вы познакомились с основными принципами тестирования кода Terraform.

- *Тестирование кода Terraform не может проходить локально.* В связи с этим для ручного тестирования приходится развертывать настоящие ресурсы в одной или нескольких изолированных средах.
- *Вы не можете проводить чистое модульное тестирование кода Terraform.* Поэтому все автоматическое тестирование должно происходить с помощью кода, который развертывает реальные ресурсы в одной изолированной среде или нескольких.
- *Регулярно чистите свои изолированные среды.* В противном случае ваши ресурсы станут неуправляемыми и расходы выйдут из-под контроля.
- *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 8, но он стоит того, чтобы повторить его еще раз: чем меньше модуль, тем легче его создавать, поддерживать, использовать и тестировать.

В этой главе вы также увидели несколько различных подходов к тестированию: модульное тестирование, интеграционное тестирование, сквозное тестирование, статический анализ и т. д. В табл. 9.4 перечислены компромиссы между этими различными видами тестирования.

Итак, какой подход к тестированию использовать? Ответ: их все! У каждого вида тестирования есть сильные и слабые стороны, поэтому лучшее решение — комбинировать несколько видов тестов, чтобы пребывать в уверенности, что ваш код работает как должно. Это не означает, что вы должны использовать все виды тестов в равных пропорциях: вспомните пирамиду тестирования и что в общем случае требуется много модульных тестов, меньше интеграционных и лишь совсем немного высококачественных сквозных тестов. Более того, совсем не обязательно добавлять все виды тестов одновременно. Выбирайте те, которые дадут максимальную отдачу от вложенных средств, и добавьте их в первую очередь. Практически любое тестирование лучше, чем его отсутствие, поэтому, если в какой-то момент вы сможете добавить только статический анализ, используйте его в качестве отправной точки и постепенно наращивайте.

Теперь перейдем к главе 10, где 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.

Таблица 9.4. Сравнение подходов к тестированию (чем больше черных квадратиков, тем лучше)

	Статический анализ	Тестирование плана	Тестирование сервера	Модульное тестирование	Интеграционное тестирование	Сквозное тестирование
Скорость	■■■■■	■■■■■	■■■■■	■■■■■	■□□□□	□□□□□
Дешевизна	■■■■■	■■■■■	■■■■■	■■■■■	■□□□□	□□□□□
Стабильность и надежность	■■■■■	■■■■■	■■■■■	■■■■■	■□□□□	□□□□□
Простота использования	■■■■■	■■■■■	■■■■■	■■■■■	■□□□□	□□□□□
Проверка синтаксиса	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■
Проверка политик	■■□□□	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■
Проверка работы серверов	□□□□□	□□□□□	■■■■■	■■■■■	■■■■■	■■■■■
Проверка работы других компонентов инфраструктуры	□□□□□	□□□□□	■■■■■	■■■■■	■■■■■	■■■■■
Проверка работы всей инфраструктуры	□□□□□	□□□□□	□□□□□	■■■■■	■□□□□	■■■■■

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 необходимо 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 в единое целое.

Последовательно пройдемся по каждой из этих тем.



Примеры кода

Напоминаю: все примеры кода для этой книги можно найти по адресу <https://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 отличаются, 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 и полный энтузиазма, рассказывает о Terraform всем вокруг... а начальник говорит «нет». Конечно, это его разочаровывает и обескураживает. 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. Приведу несколько примеров.

- *Разный уровень навыков.* Переход на IaC будет означать, что вашим системным администраторам придется тратить значительную часть своего времени на написание больших объемов кода: модулей Terraform, тестов на Go, рецептов для Chef и т. д. Некоторые из них были бы 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 освоения 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 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.

Одним 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, какими удивительными возможностями наделит его 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 и не окупиться. 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 спуска вашему начальнику, ежедневно навещаясь, 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 эту возможность, чтобы обрисовать будущее, в котором ваш процесс развертывания полностью автоматизированный, надежный и воспроизводимый, 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) удастся завершить вовремя и без выхода за рамки бюджета, а каждый третий и вовсе закрывается на полпути¹.

Поэтому я всегда начинаю волноваться, когда вижу, как компания пытается внедрить IaC одним махом в огромную инфраструктуру и с участием каждой

¹ The Standish Group, «CHAOS Manifesto 2013: Think Big, Act Small». 2013. https://www.standishgroup.com/sample_research_files/CM2013-8+9.pdf.

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*¹. Представьте, что вы разбили масштабный процесс миграции на несколько мелких этапов, и только по окончании последнего вы 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 независи-

¹ *Milstein D.* How To Survive a Ground-Up Rewrite Without Losing Your Sanity // OnStartups.com, Apr. 8, 2013. <https://www.onstartups.com/tabid/3339/bid/97052/How-To-Survive-a-Ground-Up-Rewrite-Without-Losing-Your-Sanity.aspx>.

мо 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, то 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. Затем случается неизбежное: сбой в работе системы. Теперь с этой проблемой приходится 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.

4. 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 изменения вручную. Это еще больше 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.
5. 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, 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 `main`, но вы, скорее всего, занимаетесь разработкой в отдельной ветке. Вот как с помощью команды `git checkout` можно создать и переключиться на ветку под названием `example-feature`:

```
$ 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 9? 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 изменить вывод `web-server.rb` на `Hello, World v2`, перезапустить сервер и проверить результат:

```
$ curl http://localhost:8000
Hello, World v2
```

Вы можете также обновить и перезапустить автоматические тесты. 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 в репозитории, четко описывая ваши изменения:

```
$ 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 (как Phabricator или ReviewBoard), либо посредством *запроса на включение 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 на 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 в своем браузере, введите название и описание запроса и нажмите кнопку **Create** (Создать). После этого члены вашей команды смогут просмотреть изменения, как показано на рис. 10.1.

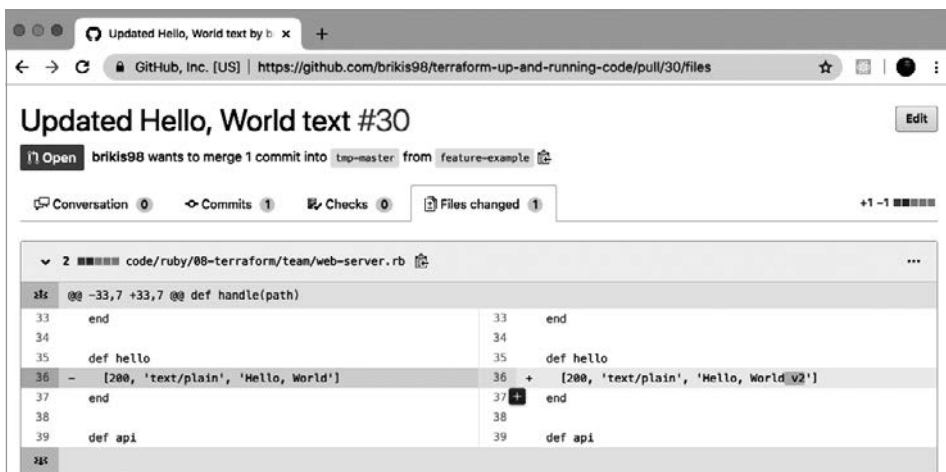


Рис. 10.1. Запрос на включение изменений на GitHub

Выполнение автоматических тестов

Вы должны настроить обработчики событий фиксации, чтобы автоматические тесты запускались после каждой отправки изменений в систему управления версиями. Чаще всего для этого используют сервер *непрерывной интеграции* (continuous integration, CI), такой как Jenkins, CircleCI или GitHub Actions. 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 изменений (рис. 10.2).

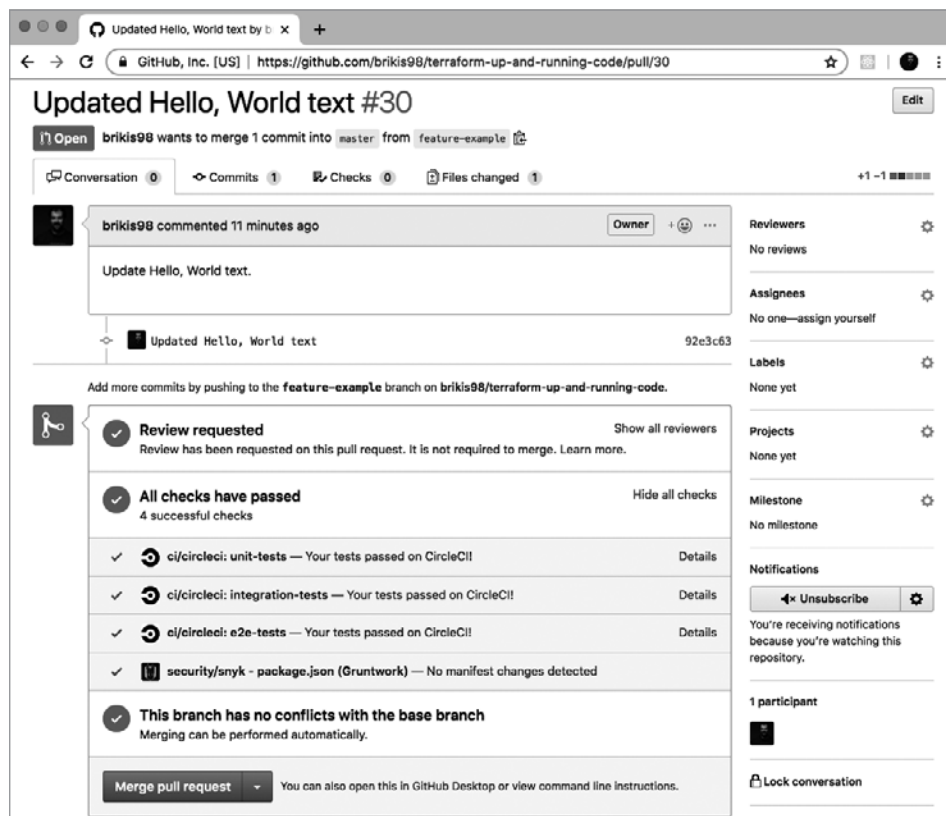


Рис. 10.2. В запросе на включение изменений на GitHub показываются результаты автоматических тестов из CircleCI

На рис. 10.2 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.

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 `main`.

Следующий шаг — выпуск кода. Если вы используете подход с неизменяемой инфраструктурой (как обсуждалось в подразделе «Средства шаблонизации серверов» в главе 1), для выпуска кода приложения его нужно упаковать в новый, неизменяемый артефакт с новым номером версии. 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 фиксации 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, получив артефакт с конкретным номером версии, его можно 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 пакетное 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 `terraform apply`.
- *Средства оркестрации*. Существует ряд средств оркестрации для развертывания и администрирования приложений, включая Kubernetes (наверное, самый популярный инструмент оркестрации контейнеров Docker), Apache Mesos, HashiCorp Nomad и Amazon ECS. В главе 7 вы видели пример использования Kubernetes для развертывания контейнеров Docker.
- *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 для реализации этих требований.

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.

- *Пакетные развертывания с заменой.* Удалите старую копию приложения, разверните вместо нее новую, подождите, пока она 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.
- *Пакетные развертывания без замены.* Разверните одну новую копию приложения, подождите, пока она 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 выделять 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 работать с пакетами большего размера (при наличии такой возможности вы можете развернуть пять новых копий одновременно) и в ходе развертывания у вас будут одновременно работать 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 выделять 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 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, таких как загрузка процессора, использование памяти, задержки, пропускная способность, частота возникновения ошибок в журналах, 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 «пакетных» стратегий. Но если вы обнаружили какие-то отличия в поведении, это может быть признаком проблем в новом коде, поэтому развертывание следует отменить, а канареечную версию удалить, пока ситуация не ухуди́лась.

Название этой стратегии взято 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* — 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 контроля версий). 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, то для выкатывания обновлений одну и ту же версию артефакта необходимо распространить по разным окружениям. Например, если у вас есть окружения для разработки (Dev), обкатки (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 в Staging (это называют *продвижением артефакта*).
4. Если версия v0.0.4 хорошо работает в Staging, повторить шаги 1 и 2 в Production.

Мы везде используем один и тот же артефакт, поэтому, если он работает в одном окружении, высока вероятность, 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 инфраструктурный код, как и прикладной, должен находиться в системе управления версиями. Значит, как и прежде, для загрузки кода используется команда `git clone`. 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 — 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: организовать единую инфраструктурную команду, специализирующуюся на создании надежных модулей промышленного уровня. Создавая библиотеку модулей, реализующих идеи из главы 8, эта команда может дать вашей компании отличное конкурентное преимущество. У каждого модуля будет компонентный API, основательная документация (в том числе и исполняемая документация в папке `examples`), комплексный набор автоматических тестов, поддержка ведения версий 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 готовую¹), 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.

¹ Например, Gruntwork Infrastructure as Code Library (<https://gruntwork.io/infrastructure-as-code-library/>).

Золотое правило Terraform

Вот как можно быстро проверить работоспособность вашего кода Terraform: откройте репозиторий *текущей* инфраструктуры, выберите наугад несколько папок и выполните команду `terraform plan` 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 `terraform plan` 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 строке 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 буде 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 на

самом деле развернуто, а это непременно приведет к ошибкам и затруднит поддержку. Таким образом, как уже описывалось в подразделе «Изоляция через рабочие области» в главе 3, 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, `main`. Значит, все изменения, которые влияют на промышленную среду, должны сохраняться непосредственно в `main` (вы можете создавать отдельные ветки, но только для создания запросов на включение внесенных изменений с последующим их слиянием с основной веткой) и команду `terraform apply` 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 выполняют `terraform apply` 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 блокировки состояния, они не могут вам помочь с блокировкой 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           = data.aws_ami.ubuntu.id  
  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           = data.aws_ami.ubuntu.id  
  instance_type = "t2.medium"  
}
```

Вот что она увидит при выполнении команды `terraform plan`:

```
$ terraform plan
```

```
(...)
```

Terraform will perform the following actions:

```
# aws_instance.foo will be updated in-place
~ resource "aws_instance" "foo" {
    ami                = "ami-0fb653ca2d3203ac1"
    id                 = "i-096430d595c80cb53"
    instance_state     = "running"
    ~ instance_type    = "t2.micro" -> "t2.medium"
    (...)
}
```

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

Изменения выглядят хорошо, поэтому Анна разворачивает 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            = data.aws_ami.ubuntu.id
    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 `t2.micro` в поле `instance_type`. Вот что 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-0fb653ca2d3203ac1"
    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 в поле `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.

Блокировки, устанавливаемые хранилищами Terraform, здесь не помогут, 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

Следующий шаг после загрузки кода на свой компьютер — 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» в главе 9, 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 `terraform apply`:

```
$ 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 `go test`:

```
$ 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 `terraform apply`, 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 `go test`:

```
$ go test -v -timeout 30m
```

```
(...)
```

```
PASS
```

```
ok  terraform-up-and-running  229.492s
```

Единственное отличие от прикладного кода: тестирование инфраструктурного кода обычно занимает больше времени, поэтому стоит подумать над тем, как сократить этот процесс, чтобы получать результаты как можно быстрее. В подразделе «Стадии тестирования» в главе 9 вы узнали, что можно разбить тест на отдельные стадии и выполнять только те, которые вам нужны, что существенно ускоряет обратную связь.

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 изменений (точно так же, как 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 инфраструктурный инженер 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 создаете¹. 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 определить в 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 8, любой модуль Terraform должен включать примеры его применения. 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 9 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?»

¹ Написание README в первую очередь называется Readme-ориентированной разработкой (<https://bit.ly/1p8QBor>).

Структура каталогов

Ваша команда должна выработать правила, устанавливающие порядок хранения кода Terraform и структуру каталогов. Поскольку структура каталогов определяет, как Terraform хранит свое состояние, нужно особенно тщательно подумать о влиянии вашей структуры каталогов на возможность предоставления гарантий изоляции, чтобы, например, изменения в среде тестирования не вызвали проблем в промышленном окружении. При разборе кода важно следить за соблюдением структуры, описанной в подразделе «Изоляция с помощью размещения файлов» в главе 3, которая обеспечивает изоляцию разных окружений (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 обработчиках событий 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 обработчике события фиксации, чтобы весь код, который сохраняется в системе управления версиями, был написан в одном стиле.

Выполнение автоматических тестов

Репозитории с инфраструктурным кодом, как и с прикладным, должны иметь обработчики событий фиксации, запускающие автоматические тесты в CI-сервере после каждой фиксации и отображающие их результаты на странице запроса на включение изменений. Вы уже видели в главе 9, как пишутся модульные, интеграционные и сквозные тесты для кода Terraform. Но есть еще одна крайне важная проверка, которую вы должны сделать: `terraform plan`. Здесь действует простое правило:

всегда выполняйте команду `plan` перед `apply`.

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 `terraform plan` при фиксации кода и добавляет вывод `plan` в виде комментариев к запросам на включение изменений, как показано на рис. 10.3.



Рис. 10.3. Atlantis умеет автоматически добавлять вывод команды `terraform plan` в виде комментариев к запросам на включение изменений

Terraform Cloud и Terraform Enterprise, платные инструменты, разрабатываемые в HashiCorp, тоже поддерживают автоматический запуск `plan` по запросам на включение.

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 `main` 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
```

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 ведения версий, и именно он будет развертываться.

Развертывание

Получив неизменяемый артефакт со своим номером версии, 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 включение изменений, но и запускать `terraform apply` 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 Cloud* и *Terraform Enterprise*. Продукты для предприятий от компании HashiCorp имеют веб-интерфейс, с помощью которого можно выполнять команды `terraform plan` и `terraform apply`, а также управлять переменными, конфиденциальными данными и правами доступа.

- *Terragrunt*. Это открытая обертка вокруг Terraform, которая заполняет некоторые пробелы в данной системе. Чуть позже в этой главе вы увидите, как с ее помощью в разных средах можно развернуть версионизируемый 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 встроенных стратегий развертывания: например, вы не сможете выполнить сине-зеленое развертывание обновлений в VPC или произвести переключение функций при изменении базы данных. Вам, в сущности, доступна лишь команда `terraform apply`, которая либо сработает, либо нет. Конечно, иногда в самом коде можно реализовать собственную стратегию развертывания; вспомните, скажем, пакетные обновления с нулевым временем простоя в модуле `asg-rolling-deploy`, который вы создали в предыдущих главах, но это скорее исключение, а не норма.

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 удастся пройти проверку работоспособности, балансировщик нагрузки никогда не направит к нему реальный трафик, поэтому пользователи не пострадают. Более того, в случае возникновения ошибок стратегии с плавающими 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!

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 `terraform apply`. 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 (<https://terragrunt.gruntwork.io/docs/features/auto-retry/>).

- *Ошибки состояния Terraform.* Время от времени Terraform не удается сохранить состояние при выполнении `terraform apply`. 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 восстановлении подключения к Интернету вы можете выгрузить его в удаленное хранилище (например, в S3), используя команду `state push`. 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 `terraform apply`, 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 <ID_БЛОКИРОВКИ>
```

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 `terraform apply` из Jenkins, CircleCI, GitHub Actions, Terraform Cloud, 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: а) их сложно сделать безопасными¹; б) они доступны всем разработчикам вашей компании и в) используются для выполнения произвольного кода, поэтому давать им постоянные администраторские полномочия весьма рискованно. По сути, вы предоставите каждому члену вашей команды права администратора и превратите свой CI-сервер в лакомую цель для злоумышленников.

Для минимизации этого риска можно предпринять несколько шагов.

- *Ограничьте доступ к CI-серверу.* Разрешите обращаться к нему только по HTTP, требуйте от пользователей прохождения аутентификации и следуйте рекомендациям по защите серверов (например, предусмотрите строгие правила для брандмауэра, установите fail2ban, включите ведение журнала аудита и т. д.).
- *Не открывайте доступ к CI-серверу из публичного Интернета.* То есть размещайте его в частных 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-обработчики 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 контроля версий.
- *Введите принудительное одобрение.* Настройте конвейер CI/CD так, чтобы каждое развертывание требовало одобрения хотя бы одним человеком (кроме человека, который запросил развертывание). На этом этапе проверяющий должен иметь возможность видеть изменения в коде и вывод команды `plan`, чтобы лишний раз убедиться, что все в порядке. Это гарантирует, что за каждым развертыванием, изменением кода и выводом команды `plan` следят как минимум две пары глаз.
- *Не предоставляйте CI-серверу постоянные учетные данные.* Как было показано в главе 6, вместо постоянных учетных данных, управляемых вручную (например, ключей доступа AWS, скопированных на CI-сервер), лучше при-

¹ Посмотрите десять реальных историй о том, как были взломаны конвейеры CI/CD (<https://research.nccgroup.com/2022/01/13/10-real-world-stories-of-how-weve-compromised-ci-cd-pipelines/>), где найдете некоторые поучительные примеры.

менять механизмы аутентификации, использующие временные учетные данные, например роли IAM и OIDC.

- *Старайтесь не выдавать CI-серверу постоянные администраторские полномочия.* Вместо этого изолируйте учетные данные администратора для совершенно отдельного, изолированного рабочего процесса: например, отдельного сервера, отдельного контейнера и т. д. Этот рабочий процесс должен быть строго заблокирован, чтобы никто из разработчиков не имел к нему доступа и чтобы только CI-сервер мог запускать этот процесс через крайне ограниченный удаленный API. Например, API этого рабочего процесса может позволять вам запускать только определенные команды (например, `terraform plan` и `terraform apply`), в определенных репозиториях (например, в вашем реальном репозитории), в определенных ветвях (например, в ветке `main`) и т. д. В таком случае, даже если злоумышленник получит доступ к вашему CI-серверу, он все равно не будет иметь доступа к учетным данным администратора, и все, что он сможет сделать, — это запросить развертывание некоторого кода, который уже находится в вашей системе управления версиями, что никак не сравнится с такой катастрофой, как утечка учетных данных администратора¹.

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 Prod². 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 `terraform apply` что-то пойдет не так, придется чинить это самостоятельно. Этот процесс будет менее трудоемким и напряженным, если проводить его еще до развертывания в промышленной среде.

¹ В Gruntwork Pipelines (<https://gruntwork.io/pipelines/>) вы найдете практический пример организации такого рабочего процесса.

² Идея продвижения кода Terraform по разным средам принадлежит Кифу Моррису: Using Pipelines to Manage Environments with Infrastructure as Code (bit.ly/2lJmus8).

Процесс продвижения кода Terraform по разным окружениям аналогичен продвижению прикладных артефактов, только у него есть дополнительный этап: выполнение команды `terraform plan` 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 **terraform plan**.
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 **terraform apply**.
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. 10.4.

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

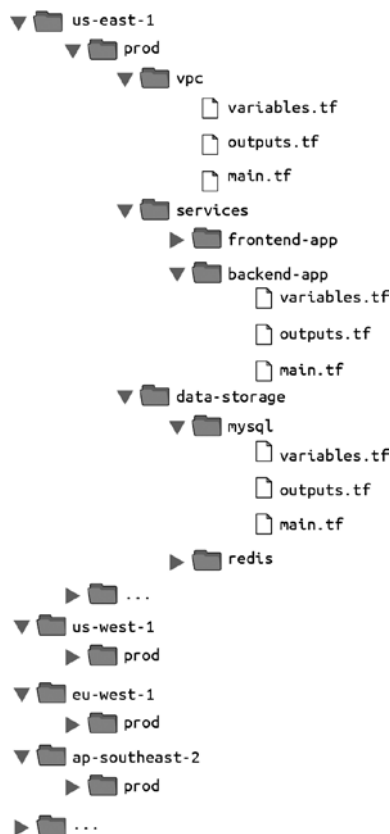


Рис. 10.4. Структура каталогов при активном копировании сред и модулей внутри каждой из них

если вы просто создаете экземпляр одного модуля, вам все равно приходится использовать большой объем шаблонного кода, одинакового для всех окружений:

- конфигурация `provider`;
- конфигурация `backend`;
- задание всех входящих переменных для модуля;
- задание всех исходящих переменных, которые возвращает модуль.

В результате в каждом вашем модуле могут накопиться десятки и сотни строк 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, о котором я уже упоминал ранее. Terragrunt представляет собой тонкую обертку вокруг Terraform, позволяющая выполнять стандартные команды `terraform`, но используя для этого команду `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 *модулей*, 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. 10.5.

Для начала установите Terragrunt, следуя инструкциям на сайте Terragrunt (<https://terragrunt.gruntwork.io/docs/getting-started/install/>). Затем 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"
}
```

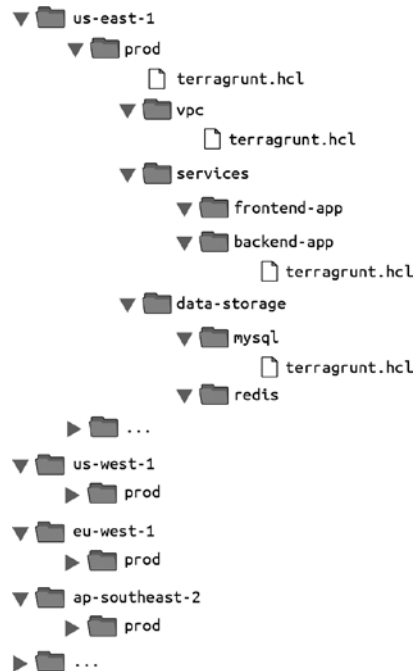


Рис. 10.5. Компоновка файлов при использовании Terragrunt

Зафиксируйте эти изменения и выпустите новую версию своего репозитория *модулей*:

```
$ 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/<ВЛАДЕЛЕЦ>/modules//data-stores/mysql?ref=v0.0.7"
}

inputs = {
  db_name      = "example_stage"
  db_username  = "admin"

  # Установите имя пользователя с помощью переменной среды TF_VAR_db_username
  # 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 `terragrunt apply`, она найдет в файле `terragrunt.hcl` параметр `source` и сделает следующее.

1. Загрузит во временную папку код, находящийся по заданному URL. Здесь поддерживается тот же синтаксис URL, 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 `terraform apply`, 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 входящие 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"

  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }

  config = {
    bucket      = "<ВАША_КОРЗИНА>"
    key         = "${path_relative_to_include()}/terraform.tfstate"
    region     = "us-east-2"
    encrypt     = true
    dynamodb_table = "<ВАША_ТАБЛИЦА>"
  }
}
```

Из этого одного блока `remote_state` Terragrunt может динамически сгенерировать конфигурацию серверной части для каждого из ваших модулей, записывая конфигурацию, определенную в блоке `config`, в файл, указанный в параметре `generate`. Обратите внимание, что значение `key` в `config` использует встроенную функцию Terragrunt с именем `path_relative_to_include()`, возвращающую 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"

  # Установите имя пользователя с помощью переменной среды TF_VAR_db_username
  # Установите пароль с помощью переменной среды TF_VAR_db_password
}
```

Блок `include` находит корневой файл `terragrunt.hcl` с помощью встроенной функции `find_in_parent_folders()` 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 `terragrunt apply`:

```
$ terragrunt apply --terragrunt-log-level debug
DEBU[0001] Reading Terragrunt config file at live/stage/data-stores/mysql/
terragrunt.hcl
DEBU[0001] Included config live/stage/terragrunt.hcl
DEBU[0001] Downloading Terraform configurations from modules into
.terragrunt-cache
DEBU[0001] Generated file backend.tf
DEBU[0013] Running command: terraform init
```

(...)

Initializing the backend...

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

(...)

```
DEBU[0024] Running command: terraform apply
```

(...)

Terraform will perform the following actions:

(...)

Plan: 5 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value: yes

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Обычно Terragrunt показывает только вывод самого Terraform, но, поскольку я добавил флаг `--terragrunt-log-level debug`, приведенный выше вывод показывает, что Terragrunt делает за кулисами.

1. Читает файл `terragrunt.hcl` в папке `mysql`, где вы запустили `apply`.
2. Загружает все настройки из корневого файла `terragrunt.hcl`.

3. Загружает код Terraform, указанный в исходном URL, в рабочую папку `.terragrunt-cache`.
4. Создает файл `backend.tf` с конфигурацией хранилища.
5. Обнаруживает, что команда `init` не запускалась и запускает ее автоматически (Terragrunt даже автоматически создаст корзину S3 и таблицу DynamoDB, если они еще не существуют).
6. Запускает `apply` для развертывания изменений.

Неплохо для пары крохотных файлов `terragrunt.hcl`!

Теперь вы можете развернуть модуль `hello-world-app` в среде Staging, добавив файл `live/stage/services/hello-world-app/terragrunt.hcl` и запустив `terragrunt apply`:

```
terraform {
  source = "github.com/<OWNER>/modules//services/hello-world-app?ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

dependency "mysql" {
  config_path = "../../data-stores/mysql"
}

inputs = {
  environment = "stage"
  ami         = "ami-0fb653ca2d3203ac1"

  min_size = 2
  max_size = 2

  enable_autoscaling = false

  mysql_config = dependency.mysql.outputs
}
```

Этот файл `terragrunt.hcl` использует URL в параметре `source` и блок `inputs` точно так же, как было показано выше, и блок `include` для извлечения настроек из корневого файла `terragrunt.hcl`, поэтому он унаследует те же настройки хранилища, за исключением `key`, который автоматически получит ожидаемое значение `services/hello-world-app/terraform.tfstate`. Единственное новое в этом файле `terragrunt.hcl` — блок `dependency`:

```
dependency "mysql" {
  config_path = "../../data-stores/mysql"
}
```

Эту особенность Terragrunt можно использовать для автоматического чтения выходных переменных другого модуля Terragrunt, поэтому вы можете передать их в качестве входных переменных в текущий модуль следующим образом:

```
mysql_config = dependency.mysql.outputs
```

Другими словами, блоки `dependency` являются альтернативой использованию источников данных `terraform_remote_state` для передачи данных между модулями. Преимущество источников данных `terraform_remote_state` заключается в том, что они встроены в Terraform, но формируют тесные взаимосвязи между модулями, потому что каждый модуль должен знать, как другие модули хранят состояние. Применение блоков `dependency` из Terragrunt позволяет модулям предоставлять общие входные данные, такие как `mysql_config` и `vpc_id`, без использования источников данных, что делает модули менее тесно связанными, и их легче тестировать и повторно использовать.

После того как приложение `hello-world-app` заработает в среде Staging, создайте аналогичные файлы `terragrunt.hcl` в `live/prod` и продвигайте тот же самый артефакт `v0.0.7` в промышленную среду, запустив `terragrunt apply` в каждом модуле.

Собираем все вместе

Теперь вы знаете, какой путь проходят прикладной и инфраструктурный код от стадии разработки до промышленной среды. В табл. 10.1 приводится сравнение этих двух рабочих процессов.

Таблица 10.1. Рабочие процессы для доставки прикладного и инфраструктурного кода

	Прикладной код	Инфраструктурный код
Использование системы управления версиями	<ul style="list-style-type: none">• <code>git clone</code>• По одному репозиторию на приложение.• Используйте ветки	<ul style="list-style-type: none">• <code>git clone</code>• Репозитории <code>live</code> и <code>modules</code>.• Не используйте ветки
Локальное выполнение кода	<ul style="list-style-type: none">• Выполняйте на локальном компьютере.• <code>ruby web-server.rb</code>• <code>ruby web-server-test.rb</code>	<ul style="list-style-type: none">• Выполняйте в изолированной среде.• <code>terraform apply</code>• <code>go test</code>
Внесение изменений в код	<ul style="list-style-type: none">• Отредактируйте код.• <code>ruby web-server.rb</code>• <code>ruby web-server-test.rb</code>	<ul style="list-style-type: none">• Отредактируйте код.• <code>terraform apply</code>• <code>go test</code>• Используйте стадии тестирования

	Прикладной код	Инфраструктурный код
Подача изменений на рассмотрение	<ul style="list-style-type: none"> • Подайте запрос на включение изменений. • Убедитесь, что соблюдаются рекомендации по написанию кода 	<ul style="list-style-type: none"> • Подайте запрос на включение изменений. • Убедитесь, что соблюдаются рекомендации по написанию кода
Выполнение автоматических тестов	<ul style="list-style-type: none"> • Тесты выполняются на CI-сервере. • Модульные тесты. • Интеграционные тесты. • Сквозные тесты. • Статический анализ 	<ul style="list-style-type: none"> • Тесты выполняются на CI-сервере. • Модульные тесты. • Интеграционные тесты. • Сквозные тесты. • Статический анализ. • <code>terraform plan</code>
Слияние и выпуск новой версии	<ul style="list-style-type: none"> • <code>git tag</code> • Создайте неизменяемый артефакт с присвоенной ему версией 	<ul style="list-style-type: none"> • <code>git tag</code> • Используйте репозиторий в качестве неизменяемого артефакта с присвоенной ему версией
Развертывание	<ul style="list-style-type: none"> • Развертывайте с помощью Terraform, средства оркестрации (такого как Kubernetes, Mesos), скриптов. • Множество стратегий развертывания: пакетные, сине-зеленые, канареечные обновления. • Выполняйте развертывание на CI-сервере. • Выделите CI-серверу ограниченные права доступа. • Продвигайте неизменяемые артефакты с присвоенными им версиями по разным средам. • После выполнения запроса на включение изменений развертывание выполнится автоматически 	<ul style="list-style-type: none"> • Развертывайте с помощью Terraform, Atlantis, Terraform Cloud, Terraform Enterprise, Terragrunt, скриптов. • Ограниченный набор стратегий развертывания. Не забудьте об обработке ошибок: повторные попытки, <code>errored.tfstate!</code> • Выполняйте развертывание на CI-сервере. • Выделите CI-серверу временные права исключительно для вызова отдельного и никому больше не доступного рабочего процесса, обладающего правами администратора. • Продвигайте неизменяемые артефакты с присвоенными им версиями по разным средам. • После выполнения запроса на включение изменений пройдите процедуру одобрения, в ходе которой кто-то еще раз проверит вывод команды <code>plan</code>, а затем выполните автоматическое развертывание

Следуя этому процессу, вы сможете выполнить и протестировать свой прикладной и инфраструктурный код, провести его разбор, упаковать его в неизменяемые артефакты с присвоенными номерами версий и затем продвинуть эти артефакты от одного окружения к другому, как показано на рис. 10.6.

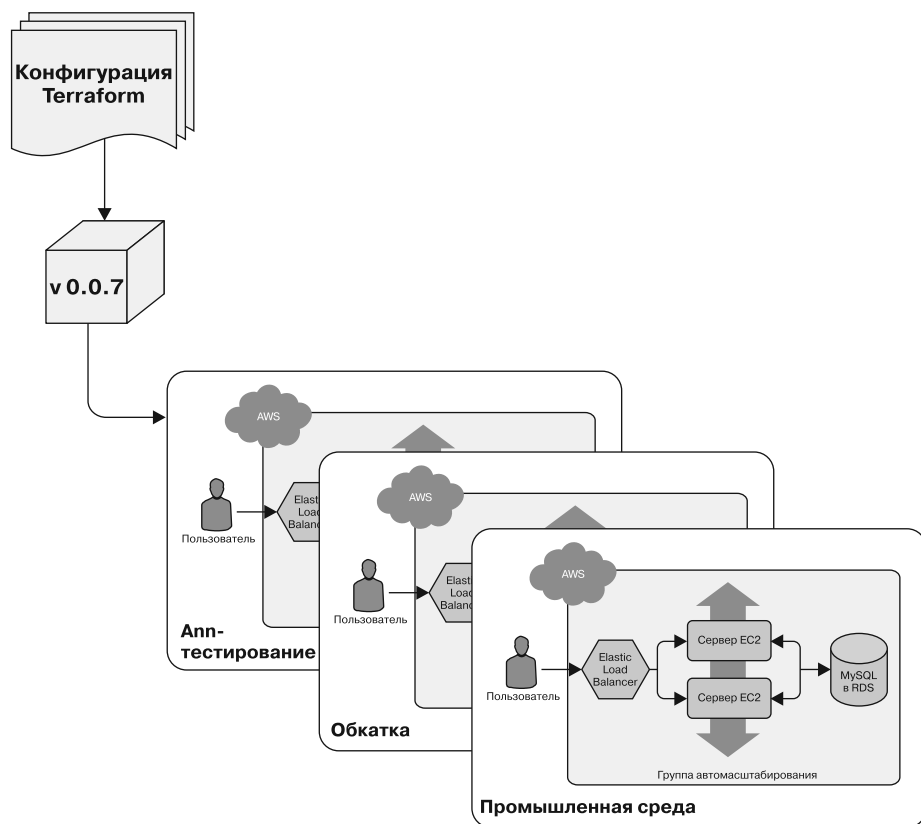


Рис. 10.6. Продвижение неизменяемого, версионизируемого артефакта с кодом Terraform от одной среды к другой

Резюме

Теперь, дочитав до этих строк, вы имеете все необходимые знания для использования Terraform в реальных условиях, включая написание кода, управление состоянием, создание универсальных модулей, эмуляцию циклов и условных выражений, выполнение развертывания, управление секретами, работу с несколькими регионами, учетными записями и облаками, создание кода промышленного уровня, тестирование и применение Terraform в командной работе.

Мы рассмотрели примеры развертывания 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 `terraform destroy` для каждого модуля.

Уникальность Terraform и IaC в целом состоит в возможности управления всеми операционными аспектами приложения с помощью тех же программистских методик, которые используются в самом приложении. Это позволяет применить к инфраструктуре всю мощь индустрии создания ПО, включая модули, разбор кода, управление версиями и автоматическое тестирование.

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 делаете свою работу как следует, а не тратите все свое время на ручное управление ресурсами, ваша команда сможет уделять все больше и 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, листайте дальше и переходите 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!

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., Мерфи 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.
- *Лукуша 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., Spafford G.* Visible Ops Handbook. — Information Technology Process Institute, 2004.
- *Дэвис Д., Дэниелс К.* Философия DevOps. Искусство управления ИТ. — 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 (<https://highscalability.com/>).
- Code as Craft (<https://codeascraft.com/>).
- Блог AWS (<https://aws.amazon.com/blogs/aws/>).
- Kitchen Soap (<https://www.kitchensoap.com/>).
- Блог Пола Хамманта (<https://paulhammant.com/>).
- Блог Мартина Фаулера (<https://martinfowler.com/bliki/>).
- Блог Gruntwork (<https://blog.gruntwork.io/>).
- Блог Евгения Брикмана (<https://www.ybrikman.com/writing/>).

Лекции

- Евгений Брикман. *Reusable, composable, battle-tested Terraform modules* (<https://bit.ly/32b28JD>).
- Евгений Брикман. *5 Lessons Learned From Writing Over 300,000 Lines of Infrastructure Code* (<https://bit.ly/2ZCcEfi>).
- Евгений Брикман. *Infrastructure as code: running microservices on AWS using Docker, Terraform and ECS* (<https://www.infoq.com/presentations/automated-testing-terraform-docker-packer/>).
- Евгений Брикман. *Agility Requires Safety* (<https://bit.ly/2YJuqJb>).
- Джез Хамбл. *Adopting Continuous Delivery* (<https://youtu.be/ZLBhVEo1OG4>).
- Майкл Рембетси и Патрик Макдоннел. *Continuously Deploying Culture* (<https://vimeo.com/51310058>).
- Джон Оллспой и Пол Хаммонд. *10+ Deploys Per Day: Dev and Ops Cooperation at Flickr* (<https://youtu.be/LdOe18KhtT4>).
- Рэйчел Потвин. *Why Google Stores Billions of Lines of Code in a Single Repository* (<https://youtu.be/W71BTKUbdqE>).
- Рич Хикки. *The Language of the System* (https://youtu.be/ROor6_NGIWU).
- Бен Кристенсен. *Don't Build a Distributed Monolith* (<https://youtu.be/-czp0Y4Z36Y>).
- Глен Вандербург. *Real Software Engineering* (<https://youtu.be/NP9AIUT9nos>).

Информационные рассылки

- DevOps Weekly (<https://www.devopsweekly.com/>).
- Gruntwork Newsletter (<https://www.gruntwork.io/newsletter/>).
- Terraform: Up & Running Newsletter (<https://bit.ly/32dnRAW>).
- Terraform Weekly Newsletter (<https://weekly.tf/>).

Онлайн-форумы

- Подфорум Terraform на форуме HashiCorp (<https://discuss.hashicorp.com/c/terraform-core>).
- Terraform subreddit (<https://www.reddit.com/r/Terraform>).
- DevOps subreddit (<https://www.reddit.com/r/devops/>).

Об авторе

Евгений (Джим) Брикман обожает программировать, писать, выступать, путешествовать и заниматься тяжелой атлетикой. Он соучредитель компании Gruntwork, которая предоставляет DevOps как услугу, 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 <http://www.ybrikman.com/>.

Иллюстрация на обложке

На обложке этой книги изображен летучий дракон (*Draco volans*) — небольшая рептилия, получившая имя за способность планировать в воздухе за счет крыло-видных кожных складок, известных как «летательная перепонка». Эта перепонка имеет яркий окрас и позволяет животному планировать в воздухе на расстояние до восьми метров. 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 в земле норах. 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. Сегодня они не являются вымирающим или близким к исчезновению видом.

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 <http://animals.oreilly.com/>.

Титульная иллюстрация взята из книги *Johnson's Natural History*.

Евгений Брикман

TERRAFORM: ИНФРАСТРУКТУРА НА УРОВНЕ КОДА

3-е международное издание

Перевела с английского Л. Киселева

Изготовлено в Казахстане. Изготовитель: ТОО «Спринт Бук».

Место нахождения и фактический адрес:

010000 Казахстан, город Астана, район Алматы,

Проспект Рахымжан Кошкарбаев, дом 10/1, н. п. 18.

Дата изготовления: 02.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 10.01.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 37,410. Заказ 0000.

Отпечатано в ТОО «ФАРОС Графикс». 100004, РК, г. Караганда, ул. Молокова, [REDACTED].

