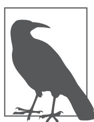
3 Как управлять состоянием Terraform

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 структуру файлов и каталогов, изоляцию и блокирование в проекте Terraform. Вот ключевые темы, по которым мы пройдемся.

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



Примеры кода

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

Что собой представляет состояние Terraform

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

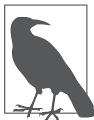
```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

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

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafe1f0",
            "availability_zone": "us-east-2c",
            "id": "i-00d689a0acc43af0f",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

Благодаря формату JSON Terraform знает, что ресурс типа `aws_instance` с именем `example` соответствует серверу EC2 с идентификатором `i-00d689a0acc43af0f`

в вашей учетной записи AWS. При каждом запуске Terraform может запросить у AWS текущее состояние этого сервера и сравнить его с вашей конфигурацией, чтобы определить, какие изменения следует внести. Иными словами, вывод команды `plan` — это расхождение между кодом на вашем компьютере и инфраструктурой, развернутой в реальном мире (согласно идентификаторам в файле состояния).



Файл состояния является приватным API

Файл состояния — это приватный API, который меняется с каждым новым выпуском и предназначен сугубо для внутреннего использования в Terraform. Вы никогда не должны редактировать его вручную или считывать его напрямую.

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

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

- *Общее хранилище для файлов состояния.* Чтобы обновлять инфраструктуру с помощью Terraform, у каждого члена команды должен быть доступ к одним и тем же файлам состояния. Это означает, что вам нужно хранить эти файлы в общедоступном месте.
- *Блокирование файлов состояния.* Разделение данных сразу же создает новую проблему: блокирование. Если два члена команды запускают Terraform одновременно, может возникнуть состояние гонки, так как обновление файлов состояния происходит параллельно со стороны двух разных процессов. Без блокирования это может привести к конфликтам, потере данных и повреждению файлов состояния.
- *Изоляция файлов состояния.* При изменении инфраструктуры рекомендуется изолировать разные окружения. Например, при правке состояния в среде предварительного или финального тестирования следует убедиться, что это никак не навредит промышленной системе. Но как изолировать изменения, если вся инфраструктура описана в одном и том же файле состояния Terraform?

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

Общее хранилище для файлов состояния

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

- *Человеческий фактор*. Вы можете легко забыть загрузить последние изменения из системы управления версиями перед запуском Terraform или сохранить свои собственные обновления постфактум. Рано или поздно кто-то в вашей команде случайно запустит Terraform с устаревшими файлами состояния, что приведет к откату или дублированию уже развернутых ресурсов.
- *Блокирование*. Большинство систем управления версиями не предоставляют никаких средств блокирования, которые могли бы предотвратить одновременное выполнение `terraform apply` двумя разными членами команды.
- *Наличие конфиденциальных данных*. Все данные в файлах состояния Terraform хранятся в виде обычного текста. Это чревато проблемами, поскольку некоторым ресурсам Terraform необходимо хранить чувствительные данные. Например, если вы создаете базу данных с помощью ресурса `aws_db_instance`, Terraform сохранит имя пользователя и пароль к ней в файле состояния в открытом виде. Открытое хранение конфиденциальных данных *где бы то ни было*, включая систему управления версиями, — плохая идея. По состоянию на май 2019 года сообщество Terraform обсуждает открытую заявку (http://bit.ly/33gqaVe), созданную по этому поводу, хотя для решения данной проблемы есть обходные пути, которые мы вскоре обсудим.

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

Удаленные хранилища решают все три проблемы, перечисленные выше.

- *Человеческий фактор*. После конфигурации удаленного хранилища Terraform будет автоматически загружать из него файл состояния при каждом выполнении

команд `plan` и `apply` и аналогично сохранять его туда после выполнения `apply`. Таким образом, возможность ручной ошибки исключается.

- *Блокирование*. Большинство удаленных хранилищ имеют встроенную поддержку блокирования. При выполнении `terraform apply` Terraform автоматически устанавливает блокировку. Если в этот момент данную команду выполняет кто-то другой, блокировка уже установлена и вам придется подождать. Команду `apply` можно ввести с параметром `-lock-timeout=<TIME>`. Так Terraform будет знать, сколько времени нужно ждать снятия блокировки (например, если указать `-lock-timeout=10m`, ожидание будет продолжаться десять минут).
- *Конфиденциальные данные*. Большинство удаленных хранилищ имеют встроенную поддержку активного и пассивного шифрования файлов состояния. Более того, они обычно позволяют настраивать права доступа (например, при использовании политик IAM в сочетании с бакетом Amazon S3), чтобы вы могли управлять тем, кто может обращаться к вашим файлам состояния и конфиденциальным данным, которые могут в них находиться. Конечно, было бы лучше, если бы в Terraform поддерживалось шифрование конфиденциальных данных прямо в файлах состояния, но эти удаленные хранилища минимизируют большинство рисков безопасности (файл состояния не хранится в открытом виде где-нибудь на вашем диске).

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

- Это управляемый сервис, поэтому для его использования не нужно развертывать и обслуживать дополнительную инфраструктуру.
- Он рассчитан на 99,999999999%-ную устойчивость и 99,99%-ную доступность. Это означает, что вам не стоит сильно волноваться о потере данных и перебоях в работе[1].
- Он поддерживает шифрование, что снижает риск хранения чувствительных данных в файлах состояния. Хотя это лишь частичное решение, так как любой член вашей команды с доступом к бакету S3 сможет просматривать эти файлы в открытом виде, но так данные будут шифроваться при сохранении (Amazon S3 поддерживает шифрование на серверной стороне с помощью AES-256) и передаче (Terraform использует SSL для чтения и записи данных в Amazon S3).
- Он поддерживает блокирование с помощью DynamoDB (подробнее об этом — чуть позже).

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

- Он поддерживает *управление версиями*, поэтому вы сможете хранить каждую ревизию своего состояния и в случае возникновения проблем откатываться на более старую версию.
- Он недорогой, поэтому большинство сценариев применения Terraform легко вписываются в бесплатный тарифный план[1].

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

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

Затем создайте бакет S3, используя ресурс `aws_s3_bucket`:

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

  # Предотвращаем случайное удаление этого бакета S3
  lifecycle {
    prevent_destroy = true
  }

  # Включаем управление версиями, чтобы вы могли просматривать
  # всю историю ваших файлов состояния
  versioning {
    enabled = true
  }

  # Включаем шифрование по умолчанию на стороне сервера
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

Этот код устанавливает четыре аргумента.

- `bucket`. Это имя бакета S3. Имейте в виду, что имена бакетов должны быть уникальными на *глобальном* уровне среди всех клиентов AWS. Поэтому вместо `"terraform-up-and-running-state"` вы должны подобрать свое собственное

[1] Ознакомьтесь с тарифами для S3 по адресу amzn.to/2yTtnw1.

название (так как бакет с этим именем я уже создал[1]). 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 бакет S3 со всем вашим состоянием Terraform. Конечно, если вы действительно хотите его удалить, просто закомментируйте этот параметр.
- `versioning`. В данном разделе включается управления версиями в бакете S3, в результате чего каждое обновление хранящегося в нем файла будет создавать его новую версию. Это позволяет просматривать старые версии и откатываться к ним в любой момент.
- `server_side_encryption_configuration`. В этом разделе включается шифрование по умолчанию на стороне сервера для всех данных, которые записываются в бакет S3. Благодаря этому ваши файлы состояния и любые конфиденциальные данные, которые могут в них содержаться, всегда будут шифроваться при сохранении в S3.

Далее нужно создать таблицу DynamoDB, которая будет использоваться для блокирования. DynamoDB — это распределенное хранилище типа «ключ — значение» от Amazon. Оно поддерживает строго согласованное чтение и условную запись — все, что необходимо для распределенной системы блокирования. Более того, оно полностью управляемое, поэтому вам не нужно заниматься никакой дополнительной инфраструктурой, а большинство сценариев применения Terraform легко впишутся в бесплатный тарифный план[2].

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

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

  attribute {
    name = "LockID"
```

[1] Подробнее об именах бакетов S3 можно почитать по адресу bit.ly/2b1s7eh.

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

```
    type = "S"
  }
}
```

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

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

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

```
terraform {
  backend "s3" {
    # Поменяйте это на имя своего бакета!
    bucket         = "terraform-up-and-running-state"
    key            = "global/s3/terraform.tfstate"
    region         = "us-east-2"

    # Замените это именем своей таблицы DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Пройдемся по этим параметрам.

- `bucket`. Имя нужного бакета S3. Не забудьте поменять его на название созданного ранее бакета.
- `key`. Файловый путь внутри бакета S3, по которому Terraform будет записывать файл состояния. Позже вы увидите, почему в предыдущем примере этому параметру присвоено `global/s3/terraform.tfstate`.

- `region`. Регион AWS, в котором находится бакет S3. Не забудьте указать регион того бакета, который вы создали ранее.
- `dynamodb_table`. Таблица DynamoDB, которая будет использоваться для блокирования. Не забудьте указать имя той таблицы, которую вы создали ранее.
- `encrypt`. Если указать `true`, состояние Terraform будет шифроваться при сохранении в S3. Это дополнительная мера, которая гарантирует шифрование данных во всех ситуациях, так как мы уже включили шифрование по умолчанию для S3.

Чтобы состояние Terraform сохранялось в этом бакете, нужно опять выполнить `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 повторное выполнение безопасно:

```
$ terraform init

Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the previous "local" backend
  to the newly configured "s3" backend. No existing state was found in the
  newly configured "s3" backend. Do you want to copy this state to the new
  "s3" backend? Enter "yes" to copy and "no" to start with an empty state.

  Enter a value:
```

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

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

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

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

```
output "s3_bucket_arn" {
  value       = aws_s3_bucket.terraform_state.arn
  description = "The ARN of the S3 bucket"
}
```

```
output "dynamodb_table_name" {
  value       = aws_dynamodb_table.terraform_locks.name
  description = "The name of the DynamoDB table"
}
```

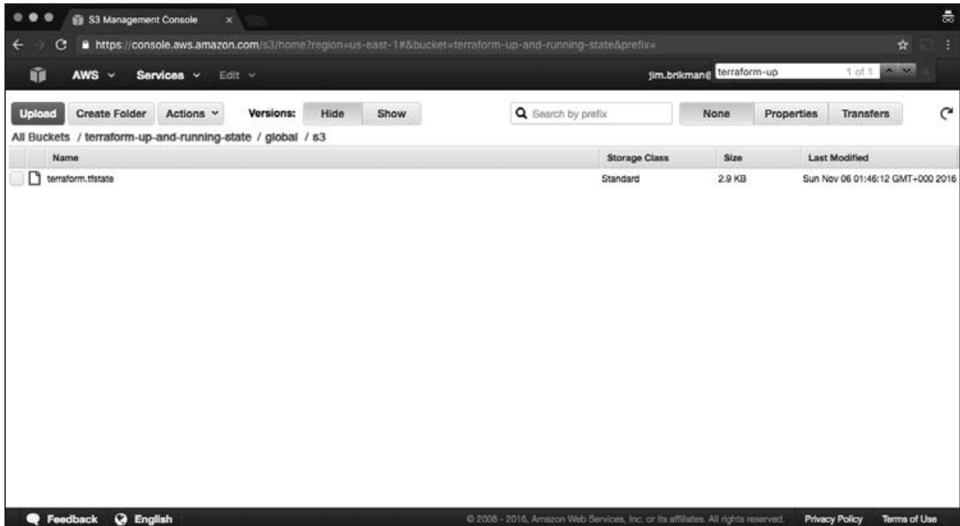


Рис. 3.1. Файл состояния Terraform, хранящийся в S3

Эти переменные выведут на экран ARN (Amazon Resource Name) вашего бакета S3 и имя вашей таблицы DynamoDB. Чтобы в этом убедиться, выполните `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 по адресу https://amzn.to/2Kw5qAc, обновите страницу и нажмите серую кнопку Show (Показать) рядом с надписью Versions (Версии). На экране должно появиться несколько версий вашего файла `terraform.tfstate`, хранящегося в бакете S3 (рис. 3.2).

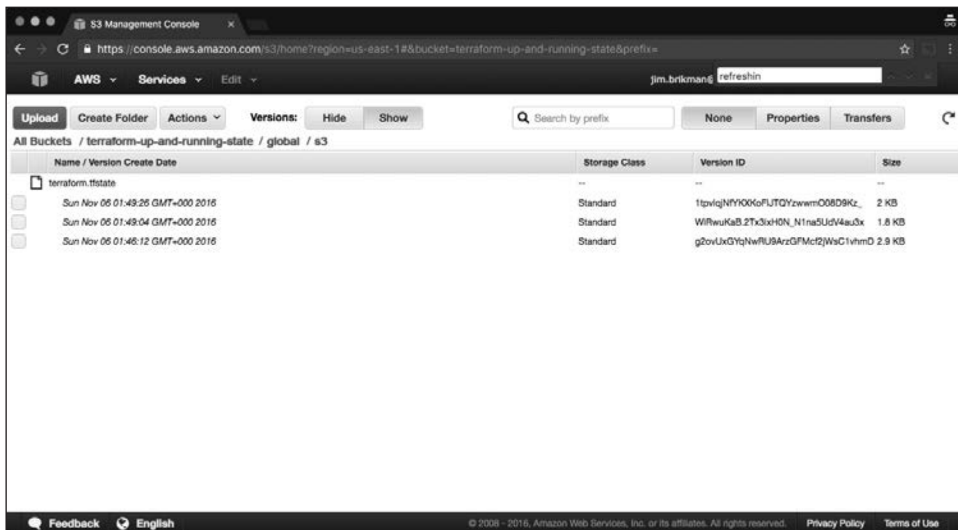


Рис. 3.2. Несколько версий состояния Terraform в S3

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

Ограничения хранилищ Terraform

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

1. Написать код Terraform, чтобы создать бакет S3 и таблицу DynamoDB, а затем развернуть этот код с использованием локального хранилища.
2. Вернуться к коду Terraform, добавить в него конфигурацию для удаленного хранилища, чтобы применить свежесозданные бакет S3 и таблицу DynamoDB,

и выполнить команду `terraform init`, чтобы скопировать ваше локальное состояние в S3.

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

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

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

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

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

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

По состоянию на май 2019 года единственным решением является использование *частичной конфигурации*, в которой можно опустить определенные параметры раздела `backend` и передавать их вместо этого в аргументе командной строки `-backend-`

`config` при вызове `terraform init`. Например, вы можете вынести повторяющиеся параметры хранилища, такие как `bucket` и `region`, в отдельный файл под названием `backend.hcl`:

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

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

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

Чтобы собрать воедино все фрагменты вашей конфигурации, выполните команду `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.

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

Изоляция файлов состояния

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