Praxishandbuch Terraform

Infrastructure as Code für DevOps, Administration und Entwicklung

KAPITEL 4

Wie man wiederverwendbare Infrastruktur mit Terraform-Modulen erzeugt

Am Ende von Kapitel 3 haben Sie die Architektur aus Abbildung 4-1 deployt.

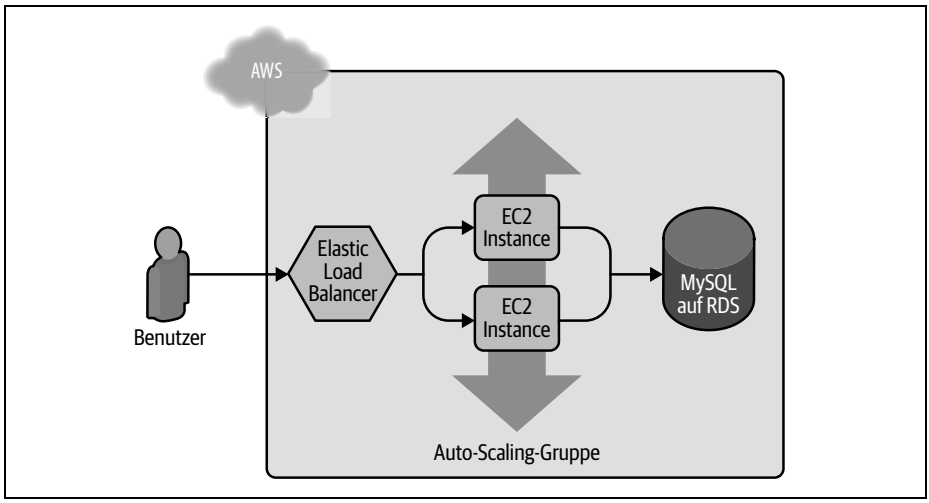

Abbildung 4-1: Die in den vorherigen Kapiteln deployte Architektur enthält einen Load Balancer, einen Webserver-Cluster und eine Datenbank.

Das funktioniert als eine erste Umgebung super, aber Sie brauchen typischerweise mindestens zwei Umgebungen: eine für die internen Tests des Teams (»Staging«) und eine, auf die die echten Anwender und Anwenderinnen zugreifen können (»Production«), wie in Abbildung 4-2 zu sehen ist. Idealerweise sind die beiden Umgebungen nahezu identisch, auch wenn Sie in Staging eventuell ein paar weniger und/oder kleinere Server laufen lassen, um Geld zu sparen.

Wie ergänzen Sie diese Produktivumgebung, ohne all den Code aus Staging kopieren zu müssen? Wie vermeiden Sie es beispielsweise, all den Code aus *stage/services/webserver-cluster* nach *prod/services/webserver-cluster* und allen Code aus *stage/data-stores/mysql* nach *prod/data-stores/mysql* zu kopieren?

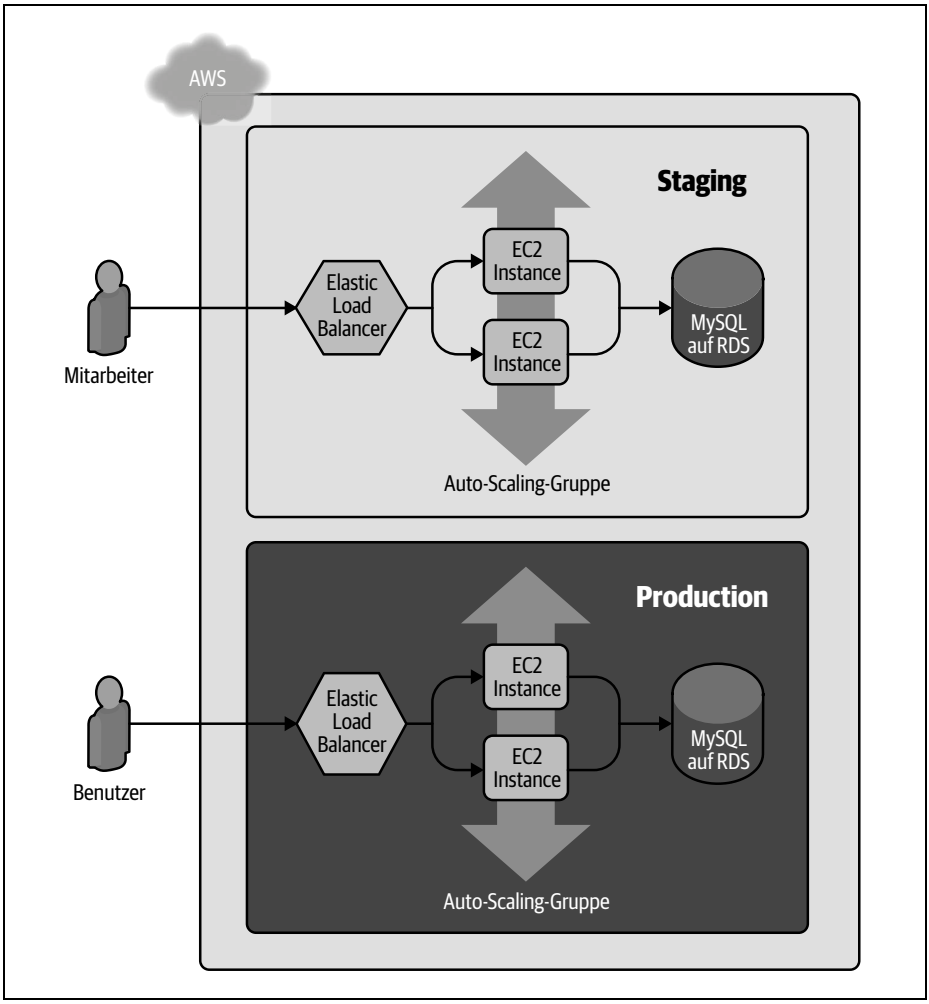

Abbildung 4-2: Die in diesem Kapitel zu deployende Architektur enthält zwei Umgebungen mit jeweils einem eigenen Load Balancer, einem Webserver-Cluster und einer eigenen Datenbank.

In einer Allzweckprogrammiersprache wie Ruby könnten Sie Code, der an mehreren Stellen benötigt wird, in ein Funktion stecken und diese dann überall einsetzen:

```
# Funktion an einem Ort definieren
def example_function()
  puts "Hallo Welt"
end

# Funktion an vielen anderen Stellen nutzen
example_function()
```

Bei Terraform können Sie Ihren Code in ein Terraform-Modul stecken und dieses Modul an mehreren Stellen in Ihrem Code verwenden. Statt den gleichen Code zwischen Staging- und Produktivumgebung hin- und herzukopieren, können beide Umgebungen Code aus dem gleichen Modul wiederverwenden (siehe Abbildung 4-3).

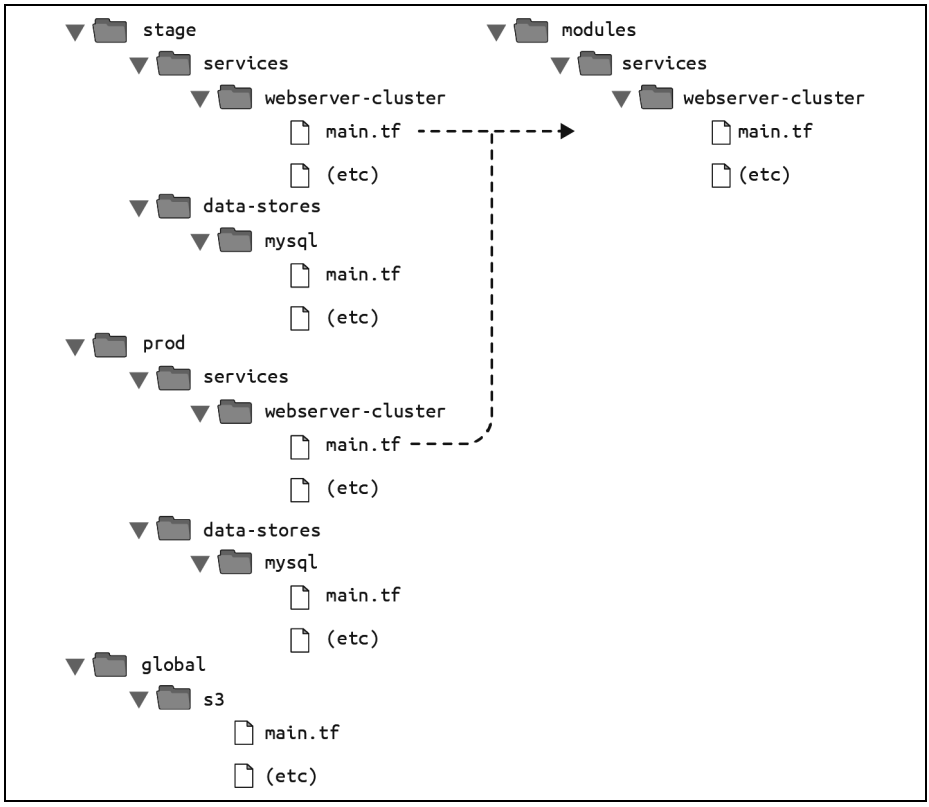

Abbildung 4-3: Wenn Sie Code in Module stecken, können Sie diesen Code in mehreren Umgebungen einsetzen.

Das ist ziemlich wichtig. Module sind der zentrale Bestandteil beim Schreiben von wiederverwendbarem, wartbarem und testbarem Terraform-Code. Sobald Sie einmal damit angefangen haben, sie einzusetzen, gibt es kein Zurück mehr. Sie werden alles als Modul erstellen, eine Bibliothek mit Modulen aufbauen, die Sie innerhalb Ihrer Firma gemeinsam nutzen, auf Module zurückgreifen, die Sie online gefunden haben, und Ihre gesamte Infrastruktur als Sammlung wiederverwendbarer Module betrachten.

In diesem Kapitel werde ich Ihnen zeigen, wie Sie Terraform-Module erstellen und verwenden, indem ich folgende Themen abdecke:

- Modulgrundlagen.
- Moduleingaben.
- Lokale Werte in Modulen.
- Modulausgaben.
- Fallstricke bei Modulen.
- Versionierung von Modulen.

Beispielcode

Nicht vergessen: Sie finden alle Codebeispiele aus diesem Buch unter *https://github.com/brikis98/terraform-up-and-running-code*.

Modulgrundlagen

Ein Terraform-Modul ist sehr einfach – jeder Satz an Terraform-Konfigurationsdateien in einem Ordner ist ein Modul. Alle Konfigurationen, die Sie bisher geschrieben haben, sind technisch gesehen Module, wenn auch vielleicht keine besonders interessanten, da Sie sie direkt deployt haben. Führen Sie `apply` direkt für ein Modul aus, wird dieses als *Root-Modul* bezeichnet. Um zu sehen, wozu Module wirklich in der Lage sind, müssen Sie ein *wiederverwendbares Modul* erstellen – also eines, das dafür gedacht ist, innerhalb von anderen Modulen genutzt zu werden.

Schauen wir uns als Beispiel den Code in *stage/services/webserver-cluster* an, der eine Auto-Scaling-Gruppe (ASG), einen Application Load Balancer (ALB), Sicherheitsgruppen und viele andere Ressourcen enthält, und verwandeln wir ihn in ein wiederverwendbares Modul.

Als ersten Schritt rufen Sie `terraform destroy` in *stage/services/webserver-cluster* auf, um alle Ressourcen abzuräumen, die Sie zuvor erstellt haben. Als Nächstes erzeugen Sie einen neuen Ordner *modules* auf oberster Ebene und verschieben alle Dateien aus *stage/services/webserver-cluster* nach *modules/services/webserver-cluster*. Sie sollten dann eine Ordnerstruktur wie die in Abbildung 4-4 haben.

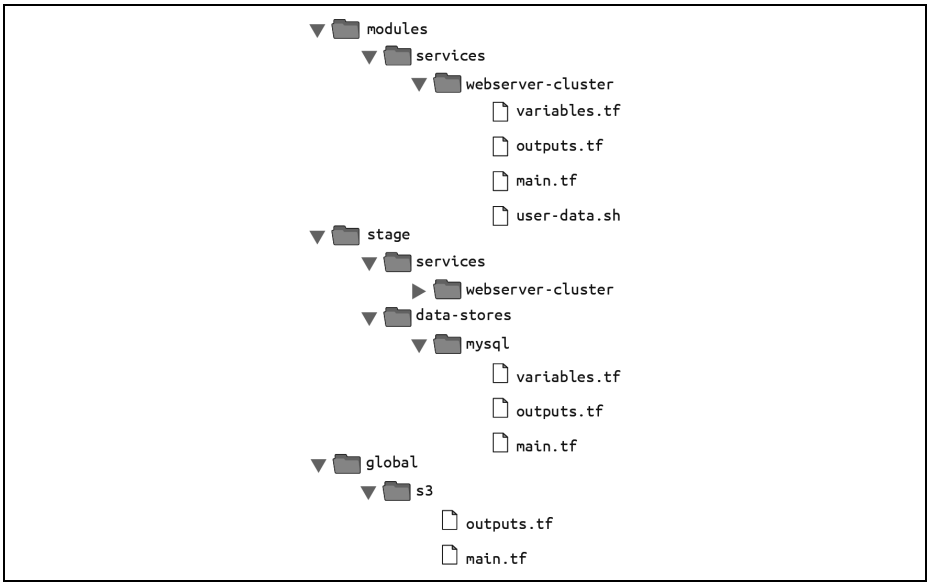

Abbildung 4-4: Verschieben Sie Ihren Webserver-Cluster-Code in einen Ordner modules/services/webserver-cluster.

Öffnen Sie die Datei *main.tf* in *modules/services/webserver-cluster* und entfernen Sie die `provider`-Definition. Provider sollten nur in Root-Modulen und nicht in wiederverwendbaren Modulen konfiguriert werden (Sie werden noch viel mehr über die Arbeit mit Providern in Kapitel 7 erfahren).

Nun können Sie dieses Modul in der Staging-Umgebung nutzen. Dies ist die Syntax für das Verwenden eines Moduls:

```
module "<NAME>" {
  source = "<SOURCE>"

  [CONFIG ...]
}
```

Dabei ist `NAME` ein Kennung, die Sie im Terraform-Code nutzen können, um sich auf dieses Modul zu beziehen (zum Beispiel `webserver_cluster`), `SOURCE` ist der Pfad, in dem der Modulcode gefunden werden kann (zum Beispiel *modules/services/webserver-cluster*), und `CONFIG` besteht aus Argumenten, die für dieses Modul spezifisch sind. Sie können beispielsweise eine neue Datei *stage/services/webserver-cluster/main.tf* erstellen und das Modul `webserver-cluster` wie folgt nutzen:

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

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

Und da ist es schon passiert: wiederverwendeter Code in mehreren Umgebungen, für den nur ganz wenig Kopieren erforderlich ist. Beachten Sie: Immer dann, wenn Sie ein Modul zu Ihrer Terraform-Konfiguration hinzunehmen oder den `source`-Parameter eines Moduls verändern, müssen Sie den `init`-Befehl ausführen, bevor Sie `plan` oder `apply` anwenden:

```
$ terraform init
Initializing modules...
- webserver_cluster in ../../../modules/services/webserver-cluster

Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!
```

Jetzt kennen Sie alle Tricks, die `init` beherrscht: Es installiert Provider, konfiguriert Ihre Backends und lädt Module herunter – alles mit einem praktischen Befehl.

Bevor Sie den `apply`-Befehl für diesen Code laufen lassen, sollte Ihnen bewusst sein, dass es ein Problem mit dem `webserver-cluster`-Modul gibt: Alle Namen dort sind hartcodiert, die Namen der Sicherheitsgruppen, ALBs und anderer Ressourcen sind also alle festgeschrieben – nutzen Sie dieses Modul mehr als einmal im gleichen AWS-Konto, werden Sie Namenskonflikte bekommen. Selbst die Details zum

Lesen des Datenbankstatus sind hartcodiert, weil die Datei *main.tf*, die Sie nach *modules/services/webserver-cluster* kopiert haben, eine Data Source `terraform_remote_state` nutzt, um die Adresse und den Port der Datenbank herauszubekommen – und dieser `terraform_remote_state` wird so genutzt, dass er in die Staging-Umgebung schaut.

Um diese Probleme zu beheben, müssen Sie das `webserver-cluster`-Modul um konfigurierbare Eingaben erweitern, sodass es sich in unterschiedlichen Umgebungen auch unterschiedlich verhalten kann.

Moduleingaben

Um eine Funktion in einer Allzweckprogrammiersprache wie Ruby konfigurierbar zu machen, können Sie diese Funktion um Eingabeparameter erweitern:

```
# Eine Funktion mit zwei Eingabeparametern
def example_function(param1, param2)
  puts "Hallo #{param1} #{param2}"
end

# Zwei Eingabeparameter an die Funktion übergeben
example_function("foo", "bar")
```

In Terraform können Module ebenfalls Eingabeparameter besitzen. Um sie zu definieren, nutzen Sie einen Mechanismus, den Sie schon kennen: Eingabevariablen. Öffnen Sie *modules/services/webserver-cluster/variables.tf* und fügen Sie drei neue Eingabevariablen hinzu:

```
variable "cluster_name" {
  description = "Name für alle Cluster-Ressourcen"
  type = string
}

variable "db_remote_state_bucket" {
  description = "Name des S3 Bucket für den Remote-Status der DB"
  type = string
}

variable "db_remote_state_key" {
  description = "Pfad für Remote-Status der DB in S3"
  type = string
}
```

Als Nächstes gehen Sie *modules/services/webserver-cluster/main.tf* durch und nutzen `var.cluster_name` statt der hartcodierten Namen (zum Beispiel statt `"Terraform-ASG-Beispiel"`). So sieht das beispielsweise für die ALB-Sicherheitsgruppe aus:

```
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"]
  }
}
```

Hier wird der `name`-Parameter auf `"${var.cluster_name}-alb"` gesetzt. Ähnliche Änderungen müssen Sie für die andere `aws_service_group`-Ressource vornehmen (zum Beispiel den Namen auf `"${var.cluster_name}-instance"` setzen), für die `aws_alb`-Ressource und für den `tag`-Abschnitt der Ressource `aws_autoscaling_group`.

Sie sollten auch die Data Source `terraform_remote_state` so anpassen, dass sie `db_remote_state_bucket` und `db_remote_state_key` für ihre Parameter `bucket` und `key` nutzt, um sicherzustellen, dass Sie die Statusdatei aus der richtigen Umgebung lesen:

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}
```

Jetzt können Sie in der Staging-Umgebung in *stage/services/webserver-cluster/main.tf* diese neuen Eingabevariablen entsprechend setzen:

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

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"
}
```

Sie sollten diese Variablen auch in der Produktivumgebung in *prod/services/webserver-cluster/main.tf* setzen, hier aber auf andere Werte, die zu dieser Umgebung passen:

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

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

Hinweis

Die Produktivdatenbank gibt es bisher noch nicht. Ich überlasse das zur Übung Ihnen: Erstellen Sie eine Produktivdatenbank ähnlich der aus dem Staging-Bereich.

Wie Sie sehen können, setzen Sie die Eingabevariablen für ein Modul über die gleiche Syntax wie beim Setzen von Argumenten für eine Ressource. Die Eingabevariablen sind die API des Moduls – sie steuern, wie es sich in unterschiedlichen Umgebungen verhält.

Bisher haben Sie Eingabevariablen für den Namen und den Remote-Status der Datenbank hinzugefügt, aber Sie wollen vielleicht auch andere Parameter in Ihrem Modul konfigurierbar machen. So möchten Sie eventuell in Staging ein kleines Webserver-Cluster betreiben, um Geld zu sparen, während Sie in der Produktivumgebung ein größeres Cluster laufen lassen möchten, um mehr Traffic handhaben zu können. Dazu können Sie drei weitere Eingabevariablen in *modules/services/webserver-cluster/variables.tf* aufnehmen:

```
variable "instance_type" {
  description = "Art der auszuführenden EC2 Instances (z.B. t2.micro)"
  type = string
}

variable "min_size" {
  description = "Minimale Zahl von EC2 Instances in der ASG"
  type = number
}

variable "max_size" {
  description = "Maximale Zahl von EC2 Instances in der ASG"
  type = number
}
```

Dann aktualisieren Sie die Startkonfiguration in *modules/services/webserver-cluster/main.tf*, sodass diese den Paramater `instance_type` auf die neue Eingabevariable `var.instance_type` setzt:

```
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
  })

  # Erforderlich beim Einsatz einer Startkonfiguration
  # zusammen mit einer Auto-Scaling-Gruppe.
  lifecycle {
    create_before_destroy = true
  }
}
```

Genauso sollten Sie in der ASG-Definition in der gleichen Datei die Parameter `min_size` und `max_size` auf die neuen Eingabevariablen `var.min_size` und `var.max_size` setzen:

```
resource "aws_autoscaling_group" "example" {
  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

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
}
```

In der Staging-Umgebung (*stage/services/webserver-cluster/main.tf*) können Sie das Cluster nun klein und günstig halten, indem Sie `instance_type` auf `"t2.micro"` und `min_size` sowie `max_size` auf `2` setzen:

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

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

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

Demgegenüber können Sie in der Produktivumgebung einen größeren `instance_type` mit mehr CPU und Speicher nutzen, wie zum Beispiel `m4.large` (beachten Sie, dass dieser Instanztyp nicht Teil des AWS Free Tier ist – wenn Sie nur etwas lernen und keine Kosten verursachen wollen, bleiben Sie bei `"t2.micro"`), und `max_size` auf `10` setzen, damit das Cluster abhängig von der Last wachsen und wieder schrumpfen kann (keine Sorge, das Cluster wird zunächst mit zwei Instanzen starten):

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

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

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

Lokale Werte in Modulen

Es ist sehr praktisch, Eingabevariablen für die Definition der Eingangswerte Ihres Moduls zu nutzen. Aber was können Sie tun, wenn Sie eine Variable in Ihrem Modul für Hilfsberechnungen definieren wollen oder Sie in Ihrem Code einfach weniger Wiederholungen haben wollen, diese Variable aber nicht von außen konfigurierbar sein soll? So lauscht beispielsweise der Load Balancer im Modul `webserver-cluster` in *modules/services/webserver-cluster/main.tf* an Port 80 – dem Standardport für HTTP. Diese Portnummer wird aktuell an mehreren Stellen eingesetzt, unter anderem beim Load Balancer Listener:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"

  # Als Standard eine einfache 404-Seite zurückgeben
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

Und in der Sicherheitsgruppe zum Load Balancer:

```
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"]
  }
}
```

Die Werte in der Sicherheitsgruppe, einschließlich des CIDR-Blocks `0.0.0.0/0` für »alle IPs«, des Werts `0` für »jeden Port« und des Werts `"-1"` für »jedes Protokoll«, werden ebenfalls an mehreren Stellen im Modul eingesetzt. Dadurch, dass diese magischen Werte an mehreren Stellen hartcodiert sind, wird es schwerer, den Code zu lesen und zu warten. Sie können die Werte in Eingabevariablen extrahieren, aber dann könnten die Anwenderinnen und Anwender Ihres Moduls (unab-

sichtlich) diese Werte überschreiben, was Sie vermutlich nicht wollen. Statt als Eingabevariablen können Sie diese aber als *lokale Werte* in einem `locals`-Block definieren:

```
locals {
  http_port    = 80
  any_port     = 0
  any_protocol = "-1"
  tcp_protocol = "tcp"
  all_ips      = ["0.0.0.0/0"]
}
```

Lokale Werte erlauben es Ihnen, einem beliebigen Terraform-Ausdruck einen Namen zuzuweisen und diesen im Modul zu verwenden. Diese Namen sind nur innerhalb des Moduls sichtbar, daher haben sie keinen Einfluss auf andere Module, und Sie können diese Werte auch nicht von außen überschreiben. Um einen lokalen Wert zu lesen, müssen Sie eine lokale Referenz verwenden, die folgende Syntax nutzt:

```
local.<NAME>
```

Mit dieser Syntax passen Sie den `port`-Parameter Ihres Load Balancer Listeners an:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port = local.http_port
  protocol = "HTTP"

  # Als Standard eine einfache 404-Seite zurückgeben
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

Und so gut wie alle Parameter in den Sicherheitsgruppen des Moduls können Sie auch anpassen, einschließlich des Load Balancers:

```
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
  }
}
```

Mit lokalen Werten lässt sich Ihr Code leichter lesen und warten, daher sollten Sie sie möglichst oft verwenden.

Modulausgaben

Ein leistungsfähiges Feature von ASGs ist, dass sie die Anzahl an laufenden Servern abhängig von der Last erhöhen oder verringern können. Eine Möglichkeit besteht darin, eine *geplante Aktion* zu nutzen, die die Größe des Clusters zu einem bestimmten Zeitpunkt am Tag ändern kann. Wenn beispielsweise der Traffic auf Ihrem Cluster während der normalen Arbeitszeiten deutlich höher ist, können Sie eine geplante Aktion verwenden, um die Anzahl der Server um 9 Uhr zu erhöhen und um 17 Uhr wieder zu verringern.

Definieren Sie die geplante Aktion im Modul `webserver-cluster`, würde sie für die Staging- und Produktivumgebung gelten. Da Sie diese Form der Skalierung in Ihrer Staging-Umgebung nicht brauchen, können Sie diese geplante Aktion direkt in den Produktivkonfigurationen definieren (in Kapitel 5 werden Sie sehen, wie Sie Ressourcen bedingt definieren können, sodass die geplante Aktion doch in das Modul `webserver-cluster` verschoben werden kann).

Um eine geplante Aktion zu definieren, fügen Sie die folgenden beiden `aws_autoscaling_schedule`-Ressourcen zu *prod/services/webserver-cluster/main.tf* hinzu:

```
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      = 10
  recurrence            = "0 17 * * *"
}
```

Dieser Code nutzt eine `aws_autoscaling_schedule`-Ressource, um die Zahl der Server am Morgen auf zehn zu erhöhen (der Parameter `recurrence` greift auf die Cron-Syntax zurück, daher hat `"0 9 * * *"` die Bedeutung »jeden Tag um 9 Uhr«), und eine zweite `aws_autoscaling_schedule`-Ressource, um die Anzahl der Server am Abend zu verringern (`"0 17 * * *"` bedeutet »jeden Tag um 17 Uhr«). Aber beiden Instanzen von `aws_autoscaling_schedule` fehlt ein wichtiger Parameter `autoscaling_group_name`, der den Namen der ASG angibt. Die ASG selbst ist im Modul `webserver-`

cluster definiert – wie können Sie dann auf ihren Namen zugreifen? In einer Allzweckprogrammiersprache wie Ruby können Funktionen Werte zurückgeben:

```
# Eine Funktion, die einen Wert zurückgibt
def example_function(param1, param2)
  return "Hello, #{param1} #{param2}"
end

# Die Funktion aufrufen und den Rückgabewert erhalten
return_value = example_function("foo", "bar")
```

In Terraform kann ein Modul ebenfalls Werte zurückgeben. Auch hier nutzen Sie einen Mechanismus, den Sie schon kennen: Ausgabevariablen. Sie können den Namen der ASG als Ausgabevariable in *modules/services/webserver-cluster/outputs.tf* wie folgt ergänzen:

```
output "asg_name" {
  value       = aws_autoscaling_group.example.name
  description = "Name der Auto-Scaling-Gruppe"
}
```

Auf die Ausgabevariablen von Modulen können Sie mit folgender Syntax zugreifen:

```
module.<MODULE_NAME>.<OUTPUT_NAME>
```

beispielsweise so:

```
module.frontend.asg_name
```

In *prod/services/webserver-cluster/main.tf* können Sie diese Syntax verwenden, um den Parameter autoscaling_group_name für jede der aws_autoscaling_schedule-Ressourcen zu setzen:

```
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
}
```

Vielleicht wollen Sie im webserver-cluster-Modul noch eine weitere Ausgabe bereitstellen: den DNS-Namen des ALB, sodass Sie wissen, welche URL Sie zum Tes-

ten nutzen können, wenn das Cluster deployt ist. Dazu fügen Sie noch eine Ausgabevariable in *modules/services/webserver-cluster/outputs.tf* hinzu:

```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "Domainname des Load Balancers"
}
```

Dann können Sie diese Ausgabe in *stage/services/webserver-cluster/outputs.tf* und *prod/services/webserver-cluster/outputs.tf* einfach durchreichen:

```
output "alb_dns_name" {
  value       = module.webserver_cluster.alb_dns_name
  description = "Domainname des Load Balancers"
}
```

Ihr Webserver-Cluster ist jetzt schon fast zum Deployen bereit. Das Einzige, was noch übrig bleibt, sind ein paar Fallstricke, auf die Sie achten sollten.

Fallstricke bei Modulen

Beim Erstellen von Modulen müssen Sie auf folgende Dinge achten:

- Dateipfade
- Inline-Blöcke

Dateipfade

In Kapitel 3 haben Sie das Benutzerdatenskript für das Webserver-Cluster in eine externe Datei *user-data.sh* verschoben und die eingebaute Funktion `templatefile` verwendet, um diese Datei von der Festplatte zu lesen. Das Problem bei der `templatefile`-Funktion ist, dass der verwendete Dateipfad ein relativer Pfad sein muss (Sie wollen keine absoluten Pfade verwenden, damit Ihr Terraform-Code auf vielen verschiedenen Computern mit jeweils anderem Festplattenlayout laufen kann) – aber relativ zu was?

Standardmäßig interpretiert Terraform den Pfad relativ zum aktuellen Arbeitsverzeichnis. Das funktioniert, wenn Sie die `templatefile`-Funktion in einer Terraform-Konfigurationsdatei verwenden, die sich im selben Verzeichnis befindet, in dem Sie auch `terraform apply` aufrufen (also wenn Sie die `templatefile`-Funktion im Root-Modul nutzen), nicht aber, wenn Sie `templatefile` in einem Modul einsetzen, das in einem anderen Ordner definiert ist (ein wiederverwendbares Modul).

Um dieses Problem zu lösen, können Sie einen Ausdruck namens *Pfadreferenz* verwenden, der die Form `path.<TYPE>` hat. Terraform unterstützt die folgenden Arten von Pfadreferenzen:

`path.module`
: Gibt den Dateisystempfad des Moduls zurück, in dem der Ausdruck definiert ist.

`path.root`
: Gibt den Dateisystempfad des Root-Moduls zurück.

`path.cwd`
: Gibt den Dateisystempfad des aktuellen Arbeitsverzeichnisses zurück. Bei einer normalen Verwendung von Terraform ist das das Gleiche wie `path.root`, aber einige ausgefeiltere Techniken von Terraform führen es in einem Verzeichnis als dem Root-Modul-Verzeichnis aus, was dafür sorgt, dass sich diese Pfade unterscheiden.

Für das Benutzerdatenskript brauchen Sie einen Pfad, der relativ zum Modul selbst ist, daher sollten Sie beim Aufruf der `templatefile`-Funktion in *modules/services/webserver-cluster/main.tf* `path.module` verwenden:

```
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
})
```

Inline-Blöcke

Die Konfiguration einiger Terraform-Ressourcen kann entweder als Inline-Block oder als eigene Ressource geschehen. Ein *Inline-Block* ist ein Argument, das Sie in einer Ressource in folgendem Format setzen:

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

Dabei ist `NAME` der Name des Inline-Blocks (zum Beispiel `ingress`), und `CONFIG` besteht aus einem oder mehreren Argumenten, die für diesen Inline-Block spezifisch sind (zum Beispiel `from_port` und `to_port`). Bei der Ressource `aws_security_group` können Sie beispielsweise Ingress- und Egress-Regeln entweder über Inline-Blöcke (zum Beispiel `ingress { … }`) oder als eigene `aws_security_group_rule`-Ressourcen definieren.

Nutzen Sie einen Mix aus Inline-Blöcken und eigenen Ressourcen, werden Sie aufgrund des Designs von Terraform Fehler erhalten, wenn die Konfigurationen zu Konflikten führen und sich gegenseitig überschreiben. Daher müssen Sie entweder das eine oder das andere verwenden. Mein Ratschlag: Beim Erstellen eines Moduls sollten Sie immer eigenen Ressourcen den Vorzug geben.

Der Vorteil eigener Ressourcen ist, dass sie irgendwo hinzugefügt werden können, während ein Inline-Block lediglich innerhalb des Moduls stehen kann, das eine Ressource erzeugt. Verwenden Sie nur eigene Ressourcen, werden Ihre Module flexibler und besser konfigurierbar.

Sie haben beispielsweise im Modul `webserver-cluster` (*modules/services/webserver-cluster/main.tf*) Inline-Blöcke genutzt, um Ingress- und Egress-Regeln zu definieren:

```
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
  }
}
```

Mit diesen Inline-Blöcken hat ein Benutzer oder eine Benutzerin dieses Moduls keine Möglichkeit, zusätzliche Ingress- oder Egress-Regeln außerhalb des Moduls hinzuzufügen. Um Ihr Modul flexibler zu machen, sollten Sie es so ändern, dass die gleichen Ingress- und Egress-Regeln über eigene `aws_security_group_rule`-Ressourcen definiert werden (achten Sie darauf, das für beide Sicherheitsgruppen im Modul umzusetzen):

```
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
}
```

Sie sollten auch die ID der `aws_security_group` als Ausgabevariable in *modules/services/webserver-cluster/outputs.tf* exportieren:

```
output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "ID der mit dem Load Balancer verbundenen Sicherheitsgruppe"
}
```

Müssen Sie nun einen zusätzlichen Port nur in der Staging-Umgebung freigeben (zum Beispiel zum Testen), können Sie eine `aws_security_group_rule`-Ressource zu *stage/services/webserver-cluster/main.tf* hinzufügen:

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

  # (Parameter zur besseren Übersicht verborgen)
}

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"]
}
```

Hätten Sie auch nur eine Ingress- oder Egress-Regel als Inline-Block definiert, würde dieser Code nicht funktionieren. Beachten Sie, dass diese Art von Problem eine Reihe von Terraform-Ressourcen betrifft, wie zum Beispiel:

- `aws_security_group` und `aws_security_group_role`
- `aws_route_table` und `aws_route`
- `aws_network_acl` und `aws_network_acl_rule`

Jetzt sind Sie endlich dazu bereit, Ihr Webserver-Cluster in der Staging- und der Produktivumgebung zu deployen. Führen Sie wie üblich `terraform apply` aus und freuen Sie sich an zwei getrennten Kopien Ihrer Infrastruktur.

Netzwerk-Isolation

Die Beispiele in diesem Kapitel haben zwei Umgebungen erzeugt, die in Ihrem Terraform-Code isoliert sind. Sie haben auch getrennte Load Balancer, Server und Datenbanken, aber sie sind nicht auf Netzwerkebene isoliert. Um alle Beispiele in diesem Buch einfach zu halten, deployen alle Ressourcen in die gleiche *Virtual Private Cloud* (VPC). Das heißt, dass ein Server in der Staging-Umgebung mit einem Server in der Produktivumgebung kommunizieren kann – und umgekehrt.

Bei einem realen Einsatz handeln Sie sich mit dem Betreiben beider Umgebungen in einer VPC zwei Gefahren ein. Zum einen kann ein Fehler in einer Umgebung auch die andere betreffen. Nehmen Sie beispielsweise Änderungen an Staging vor und bringen unabsichtlich die Konfiguration der Routing-Tabellen durcheinander, kann auch das Produktiv-Routing betroffen sein. Zum anderen kann ein Angriff auf die eine Umgebung dafür sorgen, dass auch Zugriff auf die andere besteht. Nehmen Sie kurzfristig Änderungen an Staging vor und lassen ungeplant einen Port offen, kann jeder Hacker und jede Hackerin, der oder die darüber einbricht, nicht nur auf Ihre Staging-Daten, sondern auch auf Ihre Produktivdaten zugreifen.

Daher sollten Sie außerhalb einfacher Beispiele und Experimente jede Umgebung in einer eigenen VPC laufen lassen. Tatsächlich können Sie für zusätzliche Sicherheit sogar jede Umgebung in einem eigenen AWS-Konto betreiben.

Versionierung von Modulen

Wenn sowohl Ihre Staging- als auch Ihre Produktivumgebung auf denselben Modulordner verweisen, werden auch beide beim nächsten Deployment betroffen sein, wenn Sie eine Änderung an diesem Ordner vornehmen. Diese Art von Verkopplung macht es schwieriger, eine Änderung in Staging zu testen, ohne die Produktivumgebung zu beeinflussen. Besser ist es, *versionierte Module* zu erzeugen, sodass Sie eine Version im Staging (zum Beispiel v0.0.2) und eine andere produktiv (zum Beispiel v0.0.1) nutzen – wie in Abbildung 4-5 dargestellt.

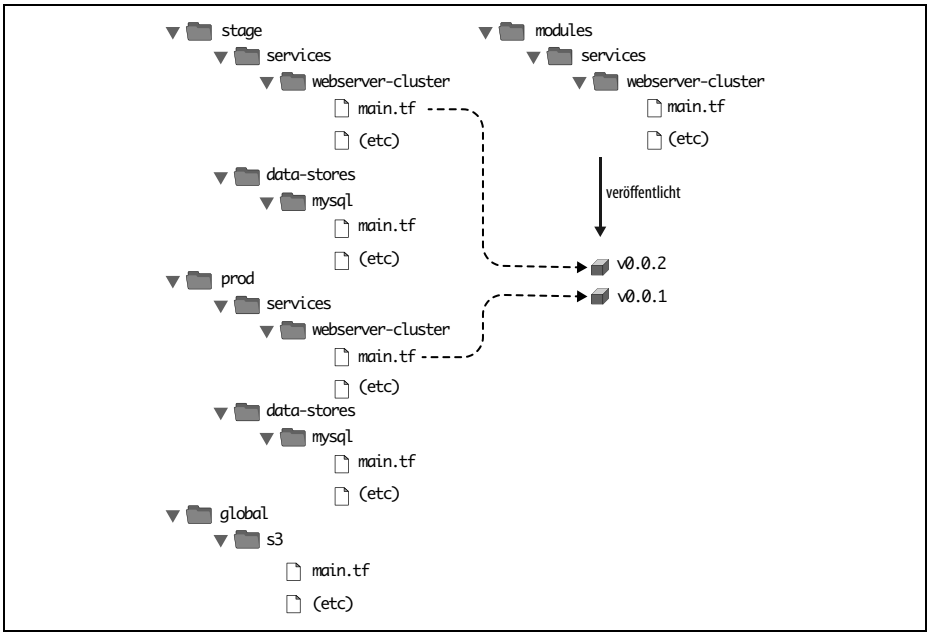

Abbildung 4-5: Durch Versionieren Ihrer Module können Sie unterschiedliche Versionen in verschiedenen Umgebungen einsetzen: zum Beispiel v0.0.01 in prod und v0.0.2 in stage.

In all den bisher vorgestellten Modulbeispielen haben Sie beim Einsatz eines Moduls immer den Parameter `source` des Moduls auf einen lokalen Pfad gesetzt. Neben einem Dateipfad unterstützt Terraform auch andere Arten von Modulquellen, zum Beispiel Git-URLs, Mercurial-URLs und beliebige HTTL-URLs.[1]

Der einfachste Weg, ein versioniertes Modul zu erstellen, ist, den Code dafür in ein eigenes Git-Repository zu stecken und den `source`-Parameter auf die URL dieses Repository zu setzen. Das bedeutet, dass Ihr Terraform-Code (mindestens) auf zwei Repositories verteilt sein wird:

modules
: Dieses Repo definiert wiederverwendbare Module. Stellen Sie sich jedes Modul als »Blaupause« vor, die einen bestimmten Teil Ihrer Infrastruktur definiert.

1 Mehr zu Source-URLs finden Sie unter *https://www.terraform.io/language/modules/sources*.

live

Dieses Repo definiert die Live-Infrastruktur, die Sie in jeder Umgebung betreiben (*stage*, *prod*, *mgmt* und so weiter). Stellen Sie sich dieses als die »Häuser« vor, die Sie anhand der »Blaupausen« im *modules*-Repo gebaut haben.

Die angepasste Ordnerstruktur für Ihren Terraform-Code sieht nun wie in Abbildung 4-6 aus.

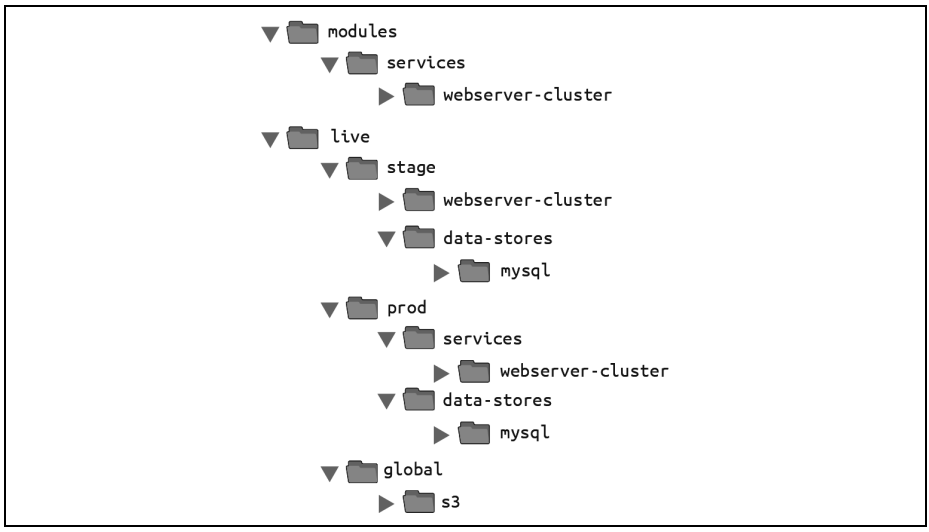

Abbildung 4-6: Sie sollten wiederverwendbare, versionierte Module im einen Repo (modules) und die Konfiguration für Ihre Live-Umgebungen in einem anderen Repo (live) ablegen.

Um diese Ordnerstruktur einzurichten, müssen Sie zuerst die Ordner *stage*, *prod* und *global* in einen Ordner namens *live* verschieben. Dann konfigurieren Sie die Ordner *live* und *modules* als eigene Git-Repositories. Hier ein Beispiel für den *modules*-Ordner:

```
$ cd modules
$ git init
$ git add .
$ git commit -m "Erster Commit des modules-Repo"
$ git remote add origin "(URL_OF_REMOTE_GIT_REPO)"
$ git push origin main
```

Sie können dem *modules*-Repo auch noch ein Tag spendieren, um dies als Versionsnummer zu verwenden. Nutzen Sie GitHub, können Sie mit dem GitHub-UI ein Release erstellen (*https://bit.ly/2Yv8kPg*), das dann hinter den Kulissen ein Tag erzeugt.

Nutzen Sie GitHub nicht, können Sie die Git CLI nutzen:

```
$ git tag -a "v0.0.1" -m "Erstes Release des webserver-cluster-Moduls"
$ git push --follow-tags
```

Jetzt können Sie dieses versionierte Modul sowohl in Staging wie auch produktiv verwenden, indem Sie eine Git-URL im `source`-Parameter angeben. So würde das in

live/stage/services/webserver-cluster/main.tf aussehen, wenn sich Ihr *modules*-Repo im GitHub-Repo *github.com/foo/modules* befindet (beachten Sie, dass der doppelte Schrägstrich in der Git-URL notwendig ist):

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/webserver-cluster?ref=v0.0.1"

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

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

Wollen Sie versionierte Module ausprobieren, ohne sich mit Git-Repos herumschlagen zu müssen, können Sie ein Modul aus dem GitHub-Repo mit den Codebeispielen zu diesem Buch (*https://github.com/brikis98/terraform-up-and-running-code*) ausprobieren (ich musste die URL aufteilen, damit sie in das Buch passt, aber es sollte alles in einer Zeile stehen):

```
source = "github.com/brikis98/terraform-up-and-running-code//
  code/terraform/04-terraform-module/module-example/modules/
  services/webserver-cluster?ref=v0.3.0"
```

Der `ref`-Parameter ermöglicht Ihnen, einen bestimmten Git-Commit über dessen SHA1-Hash, einen Branch-Namen oder – wie in diesem Beispiel – ein Git-Tag festzulegen. Ich empfehle im Allgemeinen, Git-Tags als Versionsnummern für Module zu nutzen. Branch-Namen sind nicht stabil, da Sie immer den letzten Commit eines Branchs erhalten – der sich bei jedem Ausführen von `init` ändern kann –, und die SHA1-Hashs sind nicht sehr gut lesbar. Git-Tags sind so stabil wie ein Commit (tatsächlich ist ein Tag nur ein Verweis auf ein Commit), aber sie erlauben Ihnen, einen gut lesbaren Namen zu verwenden.

Ein besonders nützliches Namensschema für Tags ist *semantisches Versionieren*. Das ist ein Versionierungsschema der Form `MAJOR.MINOR.PATCH` (zum Beispiel `1.0.4`) mit bestimmten Regeln zum Inkrementieren der verschiedenen Teile der Versionsnummer. Insbesondere sollten Sie:

- `MAJOR` hochzählen, wenn Sie inkompatible API-Änderungen vornehmen,
- `MINOR` hochzählen, wenn Sie Funktionalität auf eine abwärtskompatible Art und Weise ergänzen, und
- `PATCH` hochzählen, wenn Sie (abwärtskompatible) Fehlerkorrekturen vornehmen.

Durch semantisches Versionieren können Sie den Benutzerinnen und Benutzern Ihres Moduls kommunizieren, welche Art von Änderungen Sie vorgenommen haben und welche Auswirkungen ein Upgrade hat. Weil Sie Ihren Terraform-Code so aktualisiert haben, dass er eine versionierte Modul-URL verwendet, müssen Sie

Terraform anweisen, den Modulcode herunterzuladen, indem Sie `terraform init` erneut ausführen:

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

(...)
```

Dieses Mal können Sie sehen, dass Terraform den Modulcode von Git und nicht aus Ihrem lokalen Dateisystem lädt. Danach können Sie wie üblich den `apply`-Befehl ausführen.

Private Git-Repos

Befindet sich Ihr Terraform-Modul in einem privaten Git-Repository, müssen Sie Terraform die Möglichkeit geben, sich an diesem Repository zu authentifizieren, damit Sie dieses Repo als Modul-`source` nutzen können. Ich empfehle eine Authentifizierung per SSH, sodass Sie die Credentials nicht hartcodiert im Code stehen haben müssen. Mit einer SSH-Authentifizierung kann jeder Entwickler und jede Entwicklerin einen SSH-Schlüssel erstellen, ihn mit seinem/ihrem Git-Benutzer verknüpfen und ihn zu ssh-agent hinzufügen. Terraform wird diesen Schlüssel automatisch zur Authentifizierung verwenden, wenn Sie eine SSH-`source`-URL verwenden.[2]

Die `source`-URL sollte folgende Form haben:

```
git@github.com:<OWNER>/<REPO>.git//<PATH>?ref=<VERSION>
```

Zum Beispiel:

```
git@github.com:acme/modules.git//example?ref=v0.1.2
```

Um zu prüfen, ob Sie die URL korrekt formatiert haben, können ein `git clone` mit der Basis-URL im Terminal ausführen:

```
$ git clone git@github.com:acme/modules.git
```

Wenn dieser Befehl erfolgreich ist, sollte auch Terraform dazu in der Lage sein, auf das private Repo zuzugreifen.

Nachdem Sie nun versionierte Module verwenden, wollen wir uns anschauen, was bei Änderungen zu tun ist. Nehmen wir an, Sie änderten etwas am `webserver-cluster`-Modul und wollten dies in Staging austesten. Als Erstes würden Sie diese Änderungen in das *modules*-Repo committen:

```
$ cd modules
$ git add .
$ git commit -m "Änderungen an webserver-cluster"
$ git push origin main
```

Dann würden Sie ein neues Tag im *modules*-Repo erzeugen:

```
$ git tag -a "v0.0.2" -m "Zweites Release von webserver-cluster"
$ git push --follow-tags
```

2 Unter *https://bit.ly/2ZFLJwe* finden Sie eine nette Anleitung zur Arbeit mit SSH-Schlüsseln.

Nun können Sie *nur* die source-URL, die Sie in der Staging-Umgebung nutzen (*live/stage/services/webserver-cluster/main.tf*), aktualisieren, damit sie die neue Version nutzt:

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

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

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

In der Produktivumgebung (*live/prod/services/webserver-cluster/main.tf*) können Sie weiter problemlos mit v0.0.1 arbeiten:

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/webserver-cluster?ref=v0.0.1"

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

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

Nachdem v0.0.2 ausführlich getestet wurde und sich in Staging bewährt hat, können Sie auch die Produktivumgebung anpassen. Aber wenn sich in v0.0.2 ein Fehler zeigt, ist das kein großes Problem, weil es keinen Einfluss auf die echten Benutzer und Benutzerinnen Ihrer Produktivumgebung hat. Beheben Sie den Fehler, releasen Sie eine neue Version und wiederholen Sie den gesamten Prozess, bis Sie etwas haben, das stabil genug ist, um es produktiv einzusetzen.

Module entwickeln

Versionierte Module sind wunderbar, wenn Sie in eine gemeinsam genutzte Umgebung deployen (zum Beispiel Staging oder Production), aber wenn Sie nur Tests auf Ihrem eigenen Computer durchführen, werden Sie lokale Dateipfade nutzen wollen. Damit können Sie schneller iterieren, weil Sie eine Änderung in den Modulordnern vornehmen und den `plan`- oder `apply`-Befehl direkt in den Live-Ordnern durchführen können, statt jedes Mal Ihren Code committen, eine neue Version veröffentlichen und `init` erneut laufen lassen zu müssen.

Da das Ziel dieses Buchs darin besteht, Ihnen beim Lernen und Experimentieren mit Terraform zu helfen, werden die restlichen Codebeispiele lokale Dateipfade für Module nutzen.

Zusammenfassung

Durch das Definieren von Infrastructure as Code in Modulen können Sie eine Vielzahl von Best Practices aus der Softwareentwicklung für Ihre Infrastruktur anwenden. Sie können jede Änderung an einem Modul durch Code Reviews und automatisierte Tests validieren, Sie können semantisch versionierte Releases jedes Moduls erstellen, und Sie können gefahrlos unterschiedliche Versionen eines Moduls in unterschiedlichen Umgebungen austesten und zu älteren Versionen zurückrollen, wenn Sie auf ein Problem stoßen.

All das kann Ihre Fähigkeit, Infrastruktur schnell und zuverlässig zu bauen, dramatisch verbessern, weil Entwicklerinnen und Entwickler ganze Blöcke bewährter, getesteter und dokumentierter Infrastruktur wiederverwenden können. Sie könnten beispielsweise ein kanonisches Modul erstellen, das definiert, wie ein einzelner Microservice zu deployen ist – einschließlich des Aufsetzens des Clusters, des Skalierens des Clusters als Reaktion auf Last und des Verteilens von Traffic-Requests über das Cluster –, und jedes Team könnte dieses Modul verwenden, um seinen eigenen Microservice mit nur ein paar Zeilen Code zu managen.

Damit solch ein Modul für mehrere Teams funktioniert, muss der Terraform-Code flexibel und konfigurierbar sein. So will beispielsweise ein Team vielleicht Ihr Modul zum Deployen einer einzelnen Instanz ihres Microservice ohne Load Balancer einsetzen, während ein anderes ein Dutzend Instanzen ihres Microservice benötigt und einen Load Balancer, der den Traffic über diese Instanzen verteilt. Wie nutzen Sie bedingte Anweisungen in Terraform? Gibt es eine Möglichkeit für eine `for`-Schleife? Können Sie mit Terraform Änderungen am Microservice ausrollen, ohne dass es zu einer Downtime kommt? Diese fortgeschritteneren Aspekte der Terraform-Syntax sind Thema von Kapitel 5.