Aula 15 – Data Sources: Consultando Recursos Existentes

Imagine que você está construindo uma nova ala em um prédio já existente. Você não começaria do zero, certo? Primeiro, você precisaria saber onde estão as paredes mestras, a tubulação, a fiação elétrica. Você consultaria as plantas originais, verificaria a estrutura atual para garantir que sua nova construção se encaixe perfeitamente e não cause problemas. No mundo da Infraestrutura como Código (IaC), especificamente com ferramentas como o Terraform, essa necessidade de "consultar as plantas existentes" é fundamental.

Muitas vezes, ao automatizar a criação de infraestrutura, nos deparamos com um cenário híbrido: parte do ambiente já existe, gerenciada manualmente, por outro time, ou até mesmo por uma configuração anterior do Terraform. Como podemos, então, fazer com que nossa nova automação "enxergue" e utilize esses recursos já presentes, sem tentar recriá-los ou causar conflitos? É exatamente para resolver esse desafio que os **Data Sources** foram criados. Eles são a ponte entre o que você está construindo e o que já está lá.

Nesta aula, vamos mergulhar no universo dos Data Sources, entendendo como eles nos permitem buscar informações de infraestrutura já existente e integrá-las de forma fluida em nossos projetos. Ao final, você será capaz de utilizar Data Sources para consultar recursos, conectar infraestrutura gerenciada pelo Terraform com aquela que não é, e aplicar esses conceitos em cenários práticos, como encontrar a última imagem de sistema operacional ou detalhes de uma rede virtual já configurada. Prepare-se para otimizar seus fluxos de trabalho e construir infraestruturas mais robustas e interconectadas.

A Necessidade de "Olhar para Fora": Por Que Data Sources São Essenciais?

No dia a dia da gestão de infraestrutura, é raro começar um projeto em um ambiente completamente vazio. Quase sempre, existem redes, máquinas virtuais, bancos de dados ou outros serviços que já estão em funcionamento. Se você está usando o Terraform para provisionar novos recursos, como um novo servidor web ou um cluster de containers, e precisa que esses novos recursos se conectem a uma rede existente ou usem uma imagem de sistema operacional específica que já foi criada, como você faria?

O desafio aqui é que o Terraform, por padrão, gerencia o estado dos recursos que ele *cria*. Ele sabe o que ele fez. Mas ele não tem conhecimento intrínseco sobre recursos que foram criados fora do seu controle, seja por outro time, manualmente, ou por uma configuração anterior que não está mais ativa. Tentar criar um recurso que já existe pode gerar erros, e não conseguir referenciar um recurso existente pode inviabilizar a implantação de novos serviços. É nesse ponto que a capacidade de "consultar" se torna um superpoder.

> 💡 **Pense nos Data Sources como...**
>
> Um detetive que você contrata para encontrar informações específicas sobre o ambiente. Você dá a ele algumas pistas (filtros, nomes, IDs) e ele retorna os detalhes completos do recurso que corresponde a essas pistas.

Essa funcionalidade é crucial para cenários de integração, migração e para manter a metodologia GitOps, onde o estado desejado da sua infraestrutura é sempre a "única fonte da verdade" no Git, e essa verdade precisa considerar o que já está em produção.

Entendendo a Sintaxe e o Funcionamento Básico

Para começar a usar Data Sources, precisamos entender sua estrutura básica. Eles se assemelham aos blocos resource que você já conhece, mas com uma diferença fundamental: em vez de **criar** um recurso, eles **buscam** informações sobre um recurso existente.

1

Sintaxe Básica

A estrutura segue o padrão:

`data "tipo_do_provedor_recurso" "nome_local"`

2

Argumentos de Busca

Dentro do bloco, você define os critérios necessários para identificar o recurso desejado (filtros, nomes, IDs).

3

Retorno de Atributos

Uma vez executado, retorna os atributos do recurso encontrado para uso em outros blocos.

Exemplo Prático: Buscando a Última AMI do Ubuntu

```
# Exemplo básico: Buscando a última AMI do Ubuntu para uma região específica
data "aws_ami" "ubuntu_latest" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}

# Agora, podemos usar o ID dessa AMI para criar uma instância EC2
resource "aws_instance" "servidor_web" {
  ami           = data.aws_ami.ubuntu_latest.id
  instance_type = "t2.micro"

  tags = {
    Name = "ServidorWebUbuntu"
  }
}
```

Neste exemplo, o `data "aws_ami" "ubuntu_latest"` não cria uma AMI, mas sim consulta a AWS para encontrar a AMI mais recente do Ubuntu 22.04. O ID dessa AMI é então usado pelo `resource "aws_instance" "servidor_web"` para provisionar uma nova máquina virtual. Essa é a essência da integração: usar informações existentes para construir algo novo.

Cenário Prático: Encontrando a Última AMI do Ubuntu

Vamos aprofundar no exemplo da busca por uma AMI. Imagine que sua equipe precisa sempre implantar servidores com a versão mais recente e segura do Ubuntu. Manter o ID da AMI atualizado manualmente em seu código Terraform seria um trabalho tedioso e propenso a erros, além de ir contra o princípio de automação.

01

Definir most_recent = true

Garante que, dentre todas as AMIs que correspondem aos seus filtros, a mais nova será selecionada.

02

Especificar owners

Define quem publicou a AMI (no caso do Ubuntu, a Canonical tem um ID de conta AWS específico: 099720109477).

03

Aplicar filtros precisos

Os blocos filter refinam a busca por nome, tipo de virtualização, arquitetura, etc. É como usar um motor de busca avançado.

04

Referenciar o resultado

Use o ID retornado em seus recursos para garantir sempre a versão mais atualizada.

Código Completo com Detalhamento

```
# Detalhando a busca pela AMI do Ubuntu
data "aws_ami" "ubuntu_latest" {
  most_recent = true  # Garante que a AMI mais recente seja selecionada
  owners      = ["099720109477"]  # ID da conta AWS da Canonical, publicadora oficial do Ubuntu

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]  # Padrão de nome para Ubuntu 22.04 LTS
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]  # Tipo de virtualização
  }

  filter {
    name   = "architecture"
    values = ["x86_64"]  # Arquitetura do processador
  }
}

output "ubuntu_ami_id" {
  description = "O ID da AMI mais recente do Ubuntu 22.04 LTS."
  value       = data.aws_ami.ubuntu_latest.id
}
```

🔒 Benefício DevSecOps

Ao invés de codificar um ID de AMI que pode se tornar obsoleto rapidamente, você está instruindo o Terraform a *sempre* buscar a AMI mais atualizada que atenda aos seus requisitos. Isso não só economiza tempo, mas também aumenta a segurança, garantindo que suas novas instâncias estejam sempre rodando em uma base de sistema operacional com as últimas correções e patches.

Integrando Recursos Gerenciados e Não Gerenciados: A Ponte Essencial

O Desafio da Integração

Um dos cenários mais comuns e desafiadores na gestão de infraestrutura é a necessidade de integrar recursos que são gerenciados pelo Terraform com aqueles que não são. Talvez sua equipe de rede configure as VPCs (Virtual Private Clouds) manualmente ou através de um script legado, e sua equipe de desenvolvimento precise implantar aplicações dentro dessas VPCs usando Terraform.

Como garantir que os novos recursos sejam provisionados no ambiente correto sem que o Terraform tente recriar a VPC?

A Solução: Data Sources

Os Data Sources atuam como essa ponte crucial. Eles permitem que o Terraform "leia" o estado de recursos externos, trazendo suas informações para o escopo do seu plano de execução.

Isso significa que você pode buscar:

- O ID de uma VPC existente
- Os IDs de suas subnets
- O ID de um grupo de segurança
- Qualquer outro recurso necessário

Consistência

Sua configuração no Git reflete a realidade do ambiente, mesmo que parte dessa realidade tenha sido estabelecida por outros meios.

Auditabilidade

O código sempre sabe onde está se encaixando, facilitando rastreamento e conformidade.

Redução de Erros

Menor chance de conflitos ao provisionar novos recursos em ambientes existentes.

> "Essa capacidade é vital para ambientes complexos e para a adoção de práticas como o GitOps. Em um modelo GitOps, o Git é a fonte única da verdade."

Cenário Prático: Referenciando uma VPC Existente

Vamos aplicar o conceito de integração de recursos gerenciados e não gerenciados a um exemplo concreto: implantar um novo servidor web em uma VPC que já existe.

1

1. Buscar a VPC

Encontrar a VPC correta filtrando pelo nome (tag Name) ou por um ID específico.

2

2. Acessar Atributos

Uma vez encontrada, acessar seus atributos, como o ID.

3

3. Buscar Subnets

Usar o ID da VPC para encontrar as subnets associadas a ela.

4

4. Provisionar Recursos

Criar instâncias EC2 ou outros recursos nas subnets encontradas.

Implementação Completa

```
# Buscando uma VPC existente pelo nome
data "aws_vpc" "minha_vpc_existente" {
  filter {
    name   = "tag:Name"
    values = ["minha-vpc-de-producao"]
  }
}

# Buscando as subnets públicas dentro da VPC encontrada
data "aws_subnets" "subnets_publicas" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.minha_vpc_existente.id]
  }

  filter {
    name   = "tag:Tier"
    values = ["public"]
  }
}

# Agora, podemos criar uma instância EC2 em uma dessas subnets
resource "aws_instance" "servidor_app" {
  ami           = data.aws_ami.ubuntu_latest.id  # Reutilizando o Data Source da AMI
  instance_type = "t2.micro"
  subnet_id     = data.aws_subnets.subnets_publicas.ids[0]  # Pega a primeira subnet pública

  tags = {
    Name = "ServidorAplicacao"
  }
}
```

🎯 Ponto-Chave

Neste exemplo, o Terraform **não cria** a VPC nem as subnets. Ele apenas as consulta, obtém seus IDs e, em seguida, usa essas informações para provisionar o `aws_instance` no local correto. Isso é incrivelmente poderoso, pois permite que diferentes equipes ou processos gerenciem diferentes partes da infraestrutura, enquanto o Terraform atua como um orquestrador inteligente que respeita e se integra ao ambiente existente.

Data Sources Avançados: Encadeamento e for_each

Levando Data Sources ao Próximo Nível

A verdadeira força dos Data Sources se revela quando começamos a combiná-los e a usá-los com recursos mais avançados do Terraform, como o encadeamento e o `for_each`.

Encadeamento de Data Sources

Encadear Data Sources significa usar a saída de um Data Source como entrada para outro, criando uma sequência lógica de consultas. Isso é como um detetive que, após encontrar uma pista, usa essa pista para encontrar a próxima, até chegar à informação final desejada.

Exemplo de fluxo:

1. Buscar uma VPC
2. Usar o ID da VPC para buscar todas as subnets
3. Usar os IDs das subnets para provisionar múltiplos recursos

Uso de for_each

O uso de `for_each` com Data Sources é extremamente útil quando você precisa buscar múltiplos recursos de um mesmo tipo e iterar sobre eles.

Casos de uso:

- Criar um grupo de segurança para cada subnet
- Provisionar balanceadores de carga distribuídos
- Configurar recursos de forma escalável e consistente

Exemplo Prático: Encadeamento + for_each

```
# Exemplo de encadeamento: Buscando uma VPC e depois suas subnets
data "aws_vpc" "main_vpc" {
  filter {
    name   = "tag:Name"
    values = ["prod-vpc"]
  }
}

data "aws_subnets" "all_subnets_in_vpc" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.main_vpc.id]
  }
}

# Exemplo de for_each com Data Sources: Criando um Security Group para cada subnet
resource "aws_security_group" "subnet_sgs" {
  for_each = toset(data.aws_subnets.all_subnets_in_vpc.ids)  # Itera sobre os IDs das subnets

  name        = "sg-${each.key}"
  description = "Security group for subnet ${each.key}"
  vpc_id      = data.aws_vpc.main_vpc.id

  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"]
  }
}
```

> ⚡ **Poder da Automação**
>
> Neste exemplo, primeiro localizamos a VPC principal. Em seguida, usamos o ID dessa VPC para listar todas as subnets associadas. Finalmente, com `for_each`, criamos um Security Group único para cada uma dessas subnets, utilizando seus IDs como chave. Isso demonstra como Data Sources podem ser combinados para criar automações poderosas e dinâmicas, que se adaptam à complexidade do ambiente.

Data Sources vs. terraform import vs. terraform state: Qual Usar?

É comum haver confusão entre Data Sources e outras funcionalidades do Terraform que também lidam com recursos existentes. Entender essas diferenças é crucial para evitar dores de cabeça e manter a integridade do seu estado.

Data Sources

Propósito: Leitura

Consultam informações sobre recursos existentes que *não* são gerenciados pelo seu arquivo de estado atual do Terraform.

Analogia: Função de "lookup" ou "get"

terraform import

Propósito: Assumir gerenciamento

Permite trazer um recurso existente (criado manualmente ou por outro processo) para o controle do seu estado Terraform.

Analogia: "Adotar" um recurso órfão

terraform state

Propósito: Gerenciar o estado

Manipulação direta do arquivo de estado (mv, rm, pull, push) para refatoração, migração ou correção de inconsistências.

Analogia: Mexer no "cérebro" do Terraform

Tabela Comparativa Detalhada

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
Data Source	Ler informações de recursos existentes (não gerenciados)	API do provedor de nuvem	Buscar ID da última AMI do Ubuntu; ID de uma VPC existente
terraform import	Trazer recurso existente para gerenciamento do Terraform	Recurso já provisionado na nuvem	Importar uma instância EC2 criada manualmente para o estado do Terraform
terraform state	Gerenciar o arquivo de estado do Terraform	Arquivo terraform.tfstate (local ou remoto)	Mover um recurso de um módulo para outro; remover um recurso do estado

Boas Práticas e Armadilhas Comuns ao Usar Data Sources

Embora os Data Sources sejam ferramentas poderosas, seu uso inadequado pode levar a problemas. Adotar boas práticas é fundamental para garantir que sua infraestrutura permaneça estável, segura e fácil de manter.

1

Especificação Precisa dos Filtros

Quanto mais específicos forem seus filtros, menor a chance de o Data Source retornar o recurso errado ou múltiplos recursos quando você esperava apenas um.

Analogia: "Procure o carro vermelho, modelo X, placa Y, estacionado na rua Z" vs. "Procure um carro vermelho"

2

Evite Dependência Excessiva

Embora Data Sources sejam ótimos para integração, depender de muitos recursos não gerenciados pode tornar sua infraestrutura mais frágil.

Recomendação: Sempre que possível, tente gerenciar seus recursos com o Terraform. Quando não for possível, documente claramente as dependências externas.

3

Segurança dos Dados Consultados

Data Sources podem expor informações sensíveis, como IDs de contas, nomes de buckets S3 ou detalhes de redes.

Ação: Certifique-se de que as permissões sejam as mínimas necessárias e que as saídas (outputs) não exponham informações confidenciais desnecessariamente.

🔒 DevSecOps

No contexto de DevSecOps, é crucial considerar a segurança dos dados consultados. Sempre aplique o princípio do menor privilégio ao configurar permissões para execução do Terraform.

🤖 AIOps

AIOps pode se beneficiar de Data Sources para coletar métricas de infraestrutura existente, mas a segurança desses dados é primordial. Implemente monitoramento e alertas para mudanças inesperadas.

Consolidação e Próximos Passos

Chegamos ao fim da nossa jornada sobre Data Sources, uma funcionalidade que se revela indispensável para qualquer profissional que lida com Infraestrutura como Código em ambientes reais.

Integração	Leitura	Construção
Data Sources são a chave para integrar o novo com o existente	Permitem que o Terraform "leia" o ambiente ao seu redor	Possibilitam construir sobre infraestrutura existente sem recriá-la

---

Em Prática: Checklist de Uso

- ✓ Sempre que precisar referenciar um recurso que não será criado pelo seu Terraform atual, pense em um Data Source
- ✓ Use filtros específicos para garantir que o Data Source retorne exatamente o que você precisa
- ✓ Considere o encadeamento de Data Sources para construir lógicas de busca mais complexas
- ✓ Lembre-se da diferença entre Data Sources (leitura), terraform import (assumir gerenciamento) e terraform state (gerenciar o estado)
- ✓ Priorize a segurança ao consultar dados, garantindo que apenas informações necessárias sejam expostas

---

Próxima Aula

Aula 16 – Workspaces: Gerenciando Múltiplos Ambientes

Exploraremos como o Terraform nos permite gerenciar diferentes ambientes (desenvolvimento, homologação, produção) a partir de uma única base de código, utilizando Workspaces. Essa será uma continuação natural da nossa discussão sobre como manter a organização e a consistência em projetos de IaC.

Recursos Adicionais

- **Documentação oficial do Terraform sobre Data Sources:** Para aprofundar nos detalhes e exemplos específicos de cada provedor
- **Artigos sobre GitOps com Terraform:** Para entender como Data Sources se encaixam na filosofia de "infraestrutura como código, gerenciada via Git"
- **Guias de DevSecOps para IaC:** Para explorar as implicações de segurança ao usar Data Sources e gerenciar segredos

Autoavaliação

Teste Seus Conhecimentos

1

Questão 1

Qual é o principal objetivo de um Data Source no Terraform?

a) Criar novos recursos de infraestrutura.
b) Assumir o gerenciamento de recursos existentes.
c) Consultar informações sobre recursos de infraestrutura já existentes.
d) Remover recursos de infraestrutura do estado do Terraform.

2

Questão 2

Você precisa implantar um novo servidor em uma VPC que foi criada manualmente por outra equipe. Qual funcionalidade do Terraform você utilizaria para obter o ID dessa VPC e suas subnets sem tentar recriá-las?

a) terraform import
b) terraform state mv
c) Data Sources
d) Blocos resource

3

Questão 3

Ao usar um Data Source para buscar a última AMI do Ubuntu, qual argumento é crucial para garantir que a versão mais recente seja selecionada entre as que atendem aos filtros?

a) owners
b) filter
c) most_recent
d) architecture

4

Questão 4

Qual das seguintes afirmações descreve corretamente a relação entre Data Sources e a metodologia GitOps?

a) Data Sources são irrelevantes para GitOps, que foca apenas na criação de novos recursos.
b) Data Sources ajudam a manter o Git como a única fonte da verdade, permitindo que o código reflita o estado existente da infraestrutura.
c) GitOps proíbe o uso de Data Sources, pois eles introduzem dependências externas.
d) Data Sources são usados para reverter o estado da infraestrutura para uma versão anterior do Git.

📝 Gabarito

1. c) | **2.** c) | **3.** c) | **4.** b)

Questão Discursiva

> **Explique como a utilização de Data Sources contribui para a segurança (DevSecOps) e a automação inteligente (AIOps) em um ambiente de Infraestrutura como Código.**

NOTA IMPORTANTE: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.