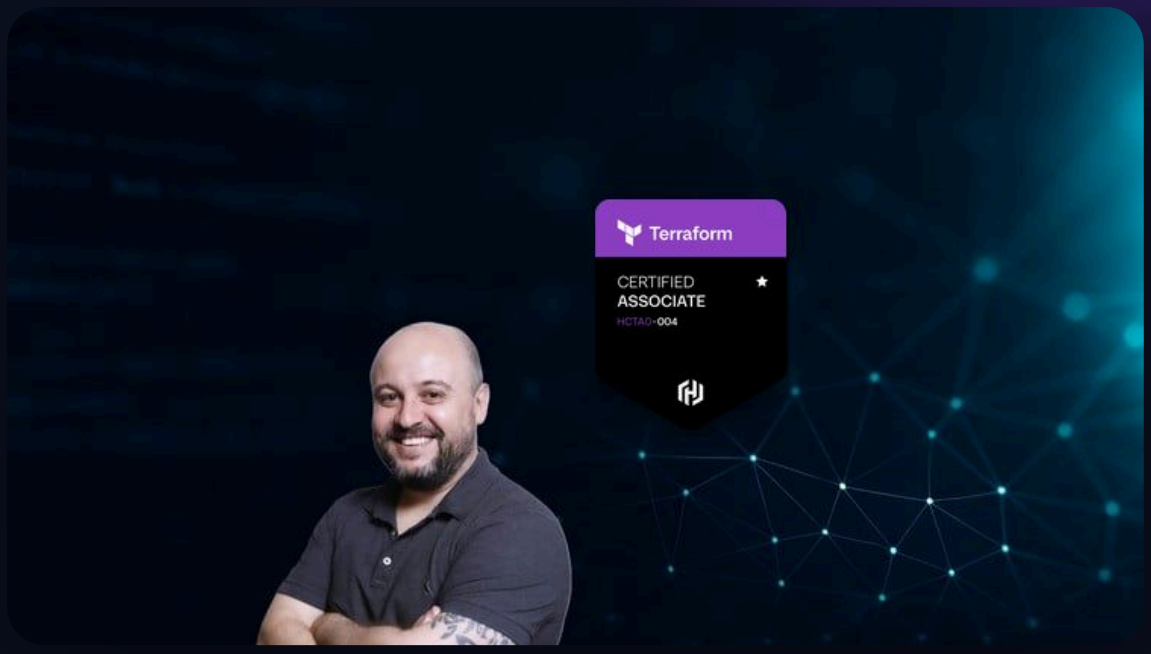


Guia de preparação Terraform Associate

Tudo o que a prova cobra, organizado por domínio: conceitos-chave, comparações que confundem, pegadinhas reais, gatilhos do enunciado, questões comentadas e um plano para chegar pronto ao exame.

8
capítulos por domínio

44
questões comentadas

70%
nota de corte

6
provas completas no curso

ANTES DE COMEÇAR

Como usar este guia

Este guia foi feito para ser estudado junto com as 6 provas completas do curso Terraform Associate: o guia explica o que cai e onde estão as armadilhas; as provas treinam o ritmo e mostram o que você ainda não domina.

Quando	O que fazer
Semana 1	Faça o teste diagnóstico (página seguinte) e leia os capítulos 1 e 2. No fim da semana, faça a Prova 1 do curso no tempo do exame.
Semana 2	Leia os demais capítulos, começando pelos domínios em que você errou mais no diagnóstico. Faça as provas 2 e 3.
Semana 3	Releia os alertas de pegadinha e as tabelas de comparação. Faça as provas 4 e 5 e revise cada erro pela explicação.
Semana 4	Estude a folha de cola, faça a Prova 6 e refaça as provas abaixo de 80%. Agende o exame quando estiver acima de 80% em todas (a nota de corte é 70%).

Mapa do exame

Cada prova do curso tem 57 questões e 60 minutos, no formato do exame. Referência de aprovação: 70%.

Domínio (capítulo)	Peso	Neste guia
1. Configuração do Terraform	—	18 conceitos · 15 pegadinhas
2. Fluxo de Trabalho Principal	—	19 conceitos · 14 pegadinhas
3. Fundamentos do Terraform	—	18 conceitos · 13 pegadinhas
4. Gerenciamento de State	—	18 conceitos · 13 pegadinhas
5. HCP Terraform	—	18 conceitos · 13 pegadinhas
6. Infraestrutura como Código com Terraform	—	17 conceitos · 12 pegadinhas
7. Manutenção da Infraestrutura	—	16 conceitos · 13 pegadinhas
8. Módulos do Terraform	—	17 conceitos · 13 pegadinhas

> **Como ler os capítulos:** os **tópicos-chave** são o que mais cai; a **tabela de memorização** separa o que a prova costuma confundir; os **alertas de prova** mostram as armadilhas das alternativas erradas; os **gatilhos** ligam palavras do enunciado à resposta; e o **checklist** diz se você está pronto para o domínio.

Estude um pouco todo dia, de graça

Instagram @strecht.simulados

Um post de prova por dia: questão comentada, pegadinhas e o “não confunda” de cada certificação.

Seguir no Instagram →

YouTube Strecht Simulados

Shorts diários com pegadinhas de prova e, aos sábados, um simulado comentado.

Inscrever-se no canal →

DIAGNÓSTICO

Você já sabe isso?

Responda às 12 questões sem consultar nada. As respostas, com a explicação e o capítulo para revisar, estão no fim do guia. Comece o estudo pelos capítulos em que errou.

D1 · Configuração do Terraform

Uma lista contém outras listas de sub-redes por zona. Qual função transforma tudo em uma única lista simples?

A. concat
B. distinct
C. flatten
D. compact

D2 · Configuração do Terraform

Um recurso usa count com uma lista de nomes. Qual problema surge ao remover um nome do meio da lista?

A. O Terraform bloqueia a remoção por padrão
B. O estado é apagado e precisa ser importado
C. As instâncias seguintes mudam de índice
D. O item removido continua existindo para sempre

D3 · Configuração do Terraform

Nas versões atuais do Terraform, a condição de validation de uma variável pode se referir a outras variáveis?

A. Não, só a data sources
B. Sim, desde o Terraform 1.9
C. Não, só ao próprio valor
D. Sim, mas só a outputs

D4 · Fluxo de Trabalho Principal

Por que é boa prática revisar o terraform plan em um pull request antes do apply?

A. Dispensa o arquivo de estado
B. Mostra o impacto antes de mudar algo
C. Aplica as mudanças mais rápido na nuvem
D. Atualiza os providers sozinho

D5 · Fluxo de Trabalho Principal

Ao rodar terraform apply sem um plano salvo, o que acontece?

A. Os recursos são recriados todos
B. Um novo plano é gerado e exibido
C. O último plano salvo é usado
D. O apply falha sem um plano salvo

D6 · Fluxo de Trabalho Principal

Qual é o comportamento do terraform fmt -check quando há arquivos fora do padrão?

A. Abre um editor para cada arquivo
B. Reescreve os arquivos e retorna zero
C. Apaga os arquivos fora do padrão
D. Retorna código de saída não zero

D7 · Fundamentos do Terraform

Qual restrição fixa um provider exatamente na versão 5.31.0?

A. version = "5.31.0"
B. version = "!= 5.31.0"
C. version = "~> 5.31"
D. version = ">= 5.31.0"

D8 · Fundamentos do Terraform

Um recurso aws_instance é usado sem declarar o provider em required_providers. De onde o Terraform tenta obtê-lo?

A. De um espelho local configurado por padrão
B. Do arquivo de estado de outra configuração
C. Do namespace hashicorp no registry público
D. Do diretório de módulos do projeto atual

D9 · Fundamentos do Terraform

Em qual bloco da configuração se declara a origem e a restrição de versão de um provider, como hashicorp/aws ~> 5.0?

A. provider dentro de um bloco resource
B. variable com o nome do fornecedor
C. required_providers dentro de terraform
D. output com o endereço do registry

D10 · Gerenciamento de State

Um recurso foi retirado do código e do estado, mas continua existindo na nuvem. Como o Terraform o trata depois disso?

A. Destrói o recurso no próximo plan
B. Não o vê mais nem o gerencia
C. Recria o recurso no próximo apply
D. Importa o recurso automaticamente

D11 · Gerenciamento de State

Para que serve o bloqueio (locking) do estado?

A. Criptografar o estado com uma chave própria
B. Guardar versões antigas do estado local
C. Impedir execuções simultâneas no mesmo estado
D. Impedir a leitura do estado por terceiros

D12 · Gerenciamento de State

Depois de revisar o resultado de um plan -refresh-only, como gravar no estado as diferenças aceitas?

A. terraform init -refresh-only
B. terraform state push -refresh
C. terraform refresh -approve-all
D. terraform apply -refresh-only

01

CAPÍTULO 1

Configuração do Terraform

É o domínio de maior peso de referência (22,9%) e o mais cheio de sintaxe. Eu preciso dominar referências (tipo.nome.atributo, data., var., local., module.), variáveis e a precedência dos valores, tipos, funções, expressões for, splat, count x for_each, dynamic, dependências, lifecycle, validação (validation, precondition, postcondition, check) e segredos (sensitive, ephemeral, write-only, Vault). As questões mostram trechos de código com uma vírgula ou prefixo errado, e várias são de múltipla resposta. Onde mais erro: precedência de variáveis, sensitive achando que tira do estado, e escolher o argumento certo do lifecycle.

Neste capítulo

18 tópicos-chave · 15 pegadinhas · 8 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Referências

Resource: aws_vpc.principal.id; data source: data.aws_ami.ubuntu.id; variável: var.regiao; local: local.prefixo (o bloco é locals, a referência é local no singular); output de módulo: module.rede.subnet_id. Referências criam dependências implícitas.

resource x data

resource gerencia (cria, altera, destrói); data só lê objetos existentes, como uma VPC de outra equipe buscada pela tag. Atributos de data source não podem ser alterados; para gerenciar, importe como resource.

Precedência de variáveis

Ordem crescente: default, TF_VAR_, terraform.tfvars, *.auto.tfvars e, por fim, -var e -var-file na CLI. Arquivos *.auto.tfvars são lidos em ordem léxica e o último vence.

Arquivos carregados automaticamente

terraform.tfvars, terraform.tfvars.json e *.auto.tfvars. Outros, como producao.tfvars, exigem -var-file. variables.tf declara, não define valores.

Tipos

Primitivos: string, number, bool. Coleções: list, map, set (mesmo tipo de elemento). Estruturais: object (atributos nomeados com tipos diferentes) e tuple (posições com tipos diferentes). any deixa o Terraform inferir.

Conversão de tipos

"5" para number é convertido; "true" para bool também. "abc" para number gera erro de tipo, sem conversão silenciosa.

optional e default = null

optional(number, 80) declara atributo opcional de object com padrão (desde o Terraform 1.3). default = null torna a variável opcional; sem default ela é obrigatória.

count x for_each

count endereça por índice (web[0], count.index); for_each por chave (app["api"], each.key, each.value). Remover item do meio com count muda índices seguintes; com for_each só a instância removida é afetada. Não se usa os dois no mesmo bloco.

for_each com lista

for_each aceita map ou set de strings; converta com toset(var.nomes). Com set, each.key e each.value são iguais.

Expressões for e splat

[for n in var.nomes : upper(n)] gera lista; com chaves e => gera mapa; filtro com if no final. Splat aws_instance.web[*].id vale para count; para for_each use values(aws_instance.web)[*].id. one(module.app[*].id) retorna o único elemento ou null.

Bloco dynamic

Gera blocos aninhados repetidos (como ingress para cada porta) com for_each. count não vale em blocos aninhados. Use com moderação.

Funções cobradas

flatten (listas aninhadas), lookup (mapa com padrão), zipmap (listas para mapa), merge, concat, coalesce (primeiro não nulo nem vazio), try (primeira expressão sem erro), can (verdadeiro/falso), templatefile, join, replace, timestamp (muda a cada run). Não existem funções definidas pelo usuário em HCL.

Dependências

Implícitas por referência (forma preferida) ou explícitas com depends_on, quando não há referência. Ordem no arquivo ou nomes parecidos não criam dependência. Data source que referencia resource é lido depois dele, no apply se depender de valor desconhecido.

lifecycle

create_before_destroy (cria o novo antes de destruir), prevent_destroy (bloqueia destruição), ignore_changes (lista ou all; aceita mudanças externas), replace_triggered_by (recria quando outro recurso muda), precondition e postcondition.

Validação

validation na variável restringe entradas com condition e error_message; desde o 1.9 pode referenciar outras variáveis. precondition valida antes e postcondition valida depois de criar ou ler o objeto, ambas no bloco lifecycle; precondition também pode ficar em output. check com assert gera aviso, não erro, e pode ter um data source com escopo.

sensitive

sensitive = true em variável ou output oculta o valor na saída, mas ele continua no estado. Output que usa valor sensível sem sensitive = true gera erro. A função sensitive() marca um valor; nonsensitive() faz o oposto.

ephemeral e write-only

Valores efêmeros (Terraform 1.10) e argumentos write-only (1.11) evitam gravar o segredo no estado e no plano. Recurso efêmero do Vault lê sem ir ao estado. Para trocar um write-only (password_wo), altere o argumento de versão associado (password_wo_version).

Vault

Ler o segredo do Vault na execução tira credenciais do código, e o provider gera credenciais dinâmicas e curtas. Data source comum leva o valor ao estado; use ephemeral para evitar.

Macetes para memorizar

Precedência

D-E-T-A-C: Default, Env (TF_VAR_), Tfvars, Auto.tfvars, CLI. Quem vem por último manda.

Primitivos

SNB: String, Number, Bool. O resto é coleção ou estrutura.

Referências

Bloco locals, chamada local; bloco variable, chamada var.

lifecycle

Proteger = prevent; Sem queda = create_before; Fingir que não viu = ignore; Efeito dominó = replace_triggered_by.

Validações

Pré antes, pós depois, check só avisa.

Segredos

sensitive esconde da tela, ephemeral esconde do estado.

count x for_each

count conta [0]; for_each chama pelo nome ["api"].

Funções

flatten achata, zipmap casa, lookup procura com plano B, try tenta, coalesce pega o primeiro que existe.

Tabela de memorização: não confunda

Quando é...	...e quando é
count Número de instâncias ou condicional 0/1 (cond ? 1 : 0); endereço por índice.	**for_each** Instâncias por nome/chave estável; remover item afeta só ele.
object Atributos nomeados com tipos diferentes.	**map** Todos os valores do mesmo tipo, como tags map(string).
tuple Posições com tipos diferentes, tamanho fixo.	**list** Sequência de valores do mesmo tipo.
precondition Validar antes de criar (imagem é x86_64?).	**postcondition** Validar o resultado depois de criar (recebeu IP público?).
validation / precondition / postcondition Falha gera erro e interrompe.	**check** Falha gera aviso, sem interromper o apply.
sensitive Oculta na saída, mas grava no estado.	**ephemeral / write-only** Não grava no estado nem no plano.
try Primeira expressão que não gera erro (can retorna bool).	**coalesce** Primeiro argumento não nulo e não string vazia.
prevent_destroy Impedir destruição acidental.	**create_before_destroy** Evitar indisponibilidade na troca: cria o novo antes.

Números que caem

Item	Valor
validation referenciando outras variáveis	**Desde o Terraform 1.9**
Valores efêmeros (ephemeral)	**Terraform 1.10**
Argumentos write-only	**Terraform 1.11**
optional(tipo, padrão) em object	**Desde o Terraform 1.3**
Índice da primeira instância com count	**0**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que TF_VAR_ vence terraform.tfvars ou que default vence arquivos.
Como não cair: Crescente: default < TF_VAR_ < terraform.tfvars < *.auto.tfvars < -var/-var-file.

⚠ **Alerta de prova:** Achar que sensitive = true impede gravar o segredo no estado.
Como não cair: sensitive só oculta na saída; para tirar do estado use ephemeral ou write-only.

⚠ **Alerta de prova:** Usar locals.prefixo ou variable.regiao como referência.
Como não cair: Referências válidas: local.prefixo e var.regiao.

⚠ **Alerta de prova:** Usar aws_instance.web[*].id em recurso com for_each.
Como não cair: Splat direto não vale para mapa: values(aws_instance.web)[*].id.

⚠ **Alerta de prova:** Usar count dentro de bloco aninhado ingress.
Como não cair: count não vale em blocos aninhados; use dynamic com for_each.

⚠ **Alerta de prova:** Passar tolist(var.nomes) ou var.nomes[*] ao for_each.
Como não cair: Continua lista; use toset(var.nomes).

⚠ **Alerta de prova:** Achar que declarar o bucket antes no arquivo garante a ordem.
Como não cair: A ordem no arquivo não importa; sem referência, use depends_on.

⚠ **Alerta de prova:** Usar ignore_changes para impedir destruição ou prevent_destroy para aceitar tags externas.
Como não cair: prevent_destroy protege contra destruição; ignore_changes = [tags] aceita mudanças externas.

⚠ **Alerta de prova:** Escrever ignore_changes = any, true ou [*].
Como não cair: Para ignorar tudo o valor é all.

⚠ **Alerta de prova:** Achar que falha de check desfaz o apply.
Como não cair: check gera aviso; o Terraform não desfaz applies.

⚠ **Alerta de prova:** Colocar precondition no bloco variable ou validation no provider.
Como não cair: validation fica na variável; precondition e postcondition ficam no lifecycle (precondition também pode ficar em output).

⚠ **Alerta de prova:** Achar que validation só pode olhar a própria variável nas versões atuais.
Como não cair: Desde o Terraform 1.9 pode referenciar outras variáveis; a regra antiga era só a própria.

⚠ **Alerta de prova:** Usar hidden = true ou secret = true para esconder output.
Como não cair: O argumento é sensitive = true; o valor segue acessível com terraform output NOME.

⚠ **Alerta de prova:** Achar que timestamp() força recriar recursos.
Como não cair: Retorna valor novo a cada execução e só altera o atributo; use valor fixo ou ignore_changes.

⚠ **Alerta de prova:** Usar coalesce para capturar erro ou can para valor alternativo.
Como não cair: try retorna a primeira expressão sem erro; can retorna verdadeiro ou falso; coalesce ignora nulos.

Como cai na prova

📝 **Como cai:** Trecho de código com quatro sintaxes parecidas (referência, for, splat, interpolação).
Como resolver: Confira prefixo (var., local., data., module.), ordem tipo.nome.atributo e indexação ([0] para count, ["chave"] para for_each). Em for: dois-pontos para lista, => só entre chaves, if no final.

📝 **Como cai:** Variável definida em vários lugares: qual valor vale.
Como resolver: Aplique a ordem crescente e lembre que entre *.auto.tfvars o último em ordem alfabética vence.

📝 **Como cai:** Requisito de ciclo de vida (proteger, não ficar fora do ar, aceitar mudança externa, recriar por gatilho).
Como resolver: Mapeie para prevent_destroy, create_before_destroy, ignore_changes ou replace_triggered_by.

📝 **Como cai:** Múltipla resposta (Escolha DUAS) sobre tipos, funções ou referências.
Como resolver: Elimine nomes inexistentes (hash_map, unique_all, input., hidden) e itens de outra categoria (map e list não são primitivos).

📝 **Como cai:** Segredo: onde fica e como proteger.
Como resolver: sensitive só esconde; ephemeral/write-only não gravam; Vault tira do código e dá credenciais dinâmicas.

📝 **Como cai:** Validar entrada, recurso antes, recurso depois ou saúde contínua.
Como resolver: validation, precondition, postcondition ou check, nessa ordem de uso.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
valor com -var, tfvars e default ao mesmo tempo	**-var vence**
a.auto.tfvars e b.auto.tfvars com a mesma variável	**b vence (ordem léxica)**
aceitar só dev, hml ou prd com mensagem	**validation com contains e error_message**
falhar se não recebeu IP público após criar	**postcondition no lifecycle**
conferir imagem antes de criar	**precondition no lifecycle**
site não responde após apply, só alertar	**check com assert (aviso)**
banco não pode ser destruído por engano	**prevent_destroy = true**
trocar sem indisponibilidade	**create_before_destroy = true**
ferramenta externa altera tags	**ignore_changes = [tags]**
recriar quando outro recurso muda	**replace_triggered_by**
bloco ingress para cada porta	**dynamic com for_each**

segredo fora do estado e do plano	**ephemeral e write-only**
módulo/recurso opcional com count 0 ou 1	**one(module.app[*].id)**
VPC de outra equipe pela tag	**Data source**

🎯 Se você lembrar de só uma coisa deste capítulo...

- Precedência crescente: default, TF_VAR_, terraform.tfvars, *.auto.tfvars, -var/-var-file.
- sensitive oculta na saída, mas o valor continua no estado; ephemeral e write-only não gravam.
- for_each precisa de map ou set: toset() para listas; splat de for_each usa values().
- check gera aviso; validation, precondition e postcondition geram erro.
- Dependência: referência primeiro, depends_on só quando não há referência.

Essenciais para a prova (checklist)

- ☐ Sei escrever referências a resource, data source, var, local e output de módulo.
- ☐ Sei a ordem de precedência dos valores de variáveis e quais arquivos carregam sozinhos.
- ☐ Sei os tipos primitivos, de coleção e estruturais e quando usar object, tuple e map(string).
- ☐ Sei quando usar count e quando usar for_each, e como endereçar cada instância.
- ☐ Sei escrever expressões for (lista, mapa, filtro) e splat, inclusive com values() e one().
- ☐ Sei usar dynamic para blocos aninhados repetidos.
- ☐ Sei cada argumento do lifecycle e o cenário de cada um.
- ☐ Sei diferenciar validation, precondition, postcondition e check.
- ☐ Sei o que sensitive, ephemeral e write-only fazem com estado e saída.
- ☐ Sei as funções flatten, lookup, zipmap, merge, coalesce, try, can, templatefile e timestamp.

Questões de revisão do capítulo

R1.1 · Configuração do Terraform

Um data source consulta um recurso criado na mesma configuração, referenciando seu id. Como fica a ordem?

A. O data source é lido antes de tudo
B. O Terraform acusa ciclo de dependência
C. O recurso é criado antes da leitura
D. Os dois ocorrem sem ordem definida

R1.2 · Configuração do Terraform

Qual função lê um arquivo de modelo e substitui variáveis nele, útil para gerar scripts de inicialização?

A. filebase64
B. formatlist
C. templatefile
D. jsondecode

R1.3 · Configuração do Terraform

Um security group precisa de um bloco ingress para cada porta de uma lista. Qual recurso da linguagem gera esses blocos aninhados?

A. Bloco locals dentro do ingress
B. Função templatefile no ingress
C. Bloco dynamic com for_each
D. Meta-argumento count no ingress

R1.4 · Configuração do Terraform

Uma sub-rede usa vpc_id = aws_vpc.principal.id. O que o Terraform conclui dessa referência?

A. A VPC precisa de depends_on explícito
B. A VPC deve ser criada antes da sub-rede
C. Os dois recursos são criados em paralelo
D. A sub-rede deve ser criada antes da VPC

Respostas comentadas

R1.1. C) O recurso é criado antes da leitura

Por quê: Referências funcionam da mesma forma para resources e data sources.

✗ A) Incorreta. Depende do recurso.
✗ B) Incorreta. Não há ciclo.
✗ D) Incorreta. A referência define ordem.

> **Dica de prova:** Data source que referencia resource: depois dele.

R1.2. C) templatefile

Por quê: O modelo usa a sintaxe ${nome} e diretivas como %{ for }.

✗ A) Incorreta. Lê e codifica em base64.

✗ B) Incorreta. Formata listas de strings.

✗ D) Incorreta. Converte JSON em valor.

> **Dica de prova:** Modelo com variáveis: templatefile.

R1.3. C) Bloco dynamic com for_each

Por quê: Use dynamic com moderação para não prejudicar a leitura do código.

✗ A) Incorreta. locals fica no nível do módulo.

✗ B) Incorreta. Gera texto de arquivos.

✗ D) Incorreta. count não vale em blocos aninhados.

> **Dica de prova:** Blocos aninhados repetidos: dynamic.

R1.4. B) A VPC deve ser criada antes da sub-rede

Por quê: Referências entre recursos são a forma preferida de definir dependências.

✗ A) Incorreta. Não é necessário.

✗ C) Incorreta. Há dependência.

✗ D) Incorreta. É o inverso.

> **Dica de prova:** Referência cria dependência implícita.

02

CAPÍTULO 2

Fluxo de Trabalho Principal

Esse é o segundo domínio de maior peso de referência (17,5%) e eu preciso saber de cor o fluxo write, plan, apply com init antes, e as opções de cada comando: init (-upgrade, -reconfigure, -migrate-state, -backend=false, -backend-config, -plugin-dir), validate, fmt (-check, -diff, -recursive), plan (-out, -destroy, -refresh=false, -detailed-exitcode, -var, -var-file, -input=false) e apply (-auto-approve, -replace, plano salvo). A prova adora opções inventadas (-yes, -force, -save, -only) e verdadeiro ou falso sobre confirmação e init. Onde mais erro: validate versus fmt -check versus plan, e o comportamento do plano salvo (aplica sem pedir yes e é recusado se ficou obsoleto).

Neste capítulo

19 tópicos-chave · 14 pegadinhas · 6 macetes · 13 gatilhos · 4 questões comentadas

Tópicos-chave

Fluxo principal

Write, plan e apply. Na prática, o init prepara o diretório antes do primeiro plan. Na etapa Write em equipe, versione o código em repositório e use branches e pull requests.

terraform init

Prepara o diretório: baixa providers e módulos e configura o backend. Não cria recursos e é seguro repetir; repita ao mudar providers, módulos ou backend. Em diretório sem .tf, informa que está vazio.

init -upgrade

Baixa a versão mais nova permitida pelas restrições e atualiza o lock. Sem -upgrade, o init respeita as versões já gravadas no lock.

init -reconfigure x -migrate-state

-reconfigure aponta para o novo backend ignorando o estado anterior; -migrate-state leva o estado existente para o novo backend.

init -backend-config e -backend=false

-backend-config=backend.hcl completa uma configuração parcial de backend (o bloco backend não aceita variáveis). -backend=false prepara o diretório sem acessar o backend, comum antes do validate em CI.

init -plugin-dir

Usa providers de um diretório local, para servidores sem internet. Espelhos de providers também podem ser configurados no arquivo de configuração da CLI.

terraform validate

Verifica sintaxe e consistência interna (tipos, referências a variáveis) sem acessar serviços remotos. Precisa de diretório inicializado (providers instalados). Não testa credenciais nem valores que dependem das APIs, como tipo de instância inexistente.

terraform fmt

Ajusta arquivos .tf e .tfvars ao estilo canônico, sem mudar lógica. Por padrão reescreve o diretório atual; -recursive inclui subdiretórios; -check não reescreve e retorna código não zero se houver arquivo fora do padrão; -diff mostra as diferenças.

terraform plan

Mostra as ações propostas sem alterar recursos reais; faz leituras às APIs no refresh. Símbolos: + criar, ~ alterar no local, -/+ destruir e recriar. O resumo 'Plan: 2 to add, 1 to change, 1 to destroy' conta substituições como criação e destruição.

Plano salvo (-out)

terraform plan -out=tfplan salva o plano; terraform apply tfplan aplica exatamente ele, sem pedir confirmação. Se o código ou o estado mudaram depois, o Terraform recusa o plano obsoleto. O arquivo pode conter valores sensíveis.

terraform apply

Sem plano salvo, gera um novo plano, exibe e pede confirmação (yes). -auto-approve dispensa o prompt. Em falha parcial, o que foi criado fica no estado; não há rollback automático.

apply -replace

Forma recomendada de recriar um recurso sem mudar o código; mostra a substituição no plano. taint está obsoleto. Para for_each: -replace='aws_instance.web["api"]' com aspas simples para o shell.

destroy

terraform destroy remove todos os recursos da configuração pedindo confirmação, na ordem inversa das dependências. Equivale a terraform apply -destroy; plan -destroy só mostra a prévia. prevent_destroy faz o comando falhar sem destruir nada.

-target

Para exceções, como recuperar erros; uso rotineiro é desaconselhado porque pode deixar a configuração inconsistente com o estado. Ex.: terraform destroy -target=ENDEREÇO.

Opções de automação

-input=false faz o comando falhar em vez de esperar valor de variável; -detailed-exitcode faz o plan retornar 2 quando há mudanças; -auto-approve aplica sem prompt.

Valores na CLI

-var="regiao=us-east-1" passa um valor (maior precedência) e -var-file=producao.tfvars carrega arquivo que não é carregado sozinho.

known after apply

O valor só existirá depois da criação; referências a ele também ficam desconhecidas até o apply. Não indica erro nem segredo.

Plano em JSON e console

Para ferramentas: plan -out=tfplan e depois show -json tfplan. terraform console abre um console interativo para testar expressões e funções, como cidrsubnet.

Paralelismo

-parallelism=N ajusta o paralelismo das operações. Não serve para esperar o bloqueio do estado (isso é -lock-timeout) nem para pular a consulta aos objetos reais (isso é -refresh=false).

Macetes para memorizar

💡 Ordem antes da revisão

F-V-P: fmt, validate, plan. Depois do aprovado: apply do arquivo.

💡 Plano salvo

Plano salvo não pergunta e não aceita velho: sem yes, e recusado se obsoleto.

💡 Backend no init

Migrate leva a mudança; Reconfigure recomeça do zero.

💡 Símbolos do plan

+ nasce, ~ muda, -/+ morre e renasce.

💡 fmt

Sem flag reescreve; -check só acusa; -diff mostra; -recursive desce pastas.

💡 Destroy

destroy = apply -destroy; plan -destroy é só trailer.

Tabela de memorização: não confunda

Quando é...	...e quando é
terraform validate Sintaxe, tipos e referências (ex.: var.porta não declarada).	**terraform fmt -check** Estilo canônico dos arquivos, sem reescrever; falha com código não zero.
init -migrate-state Levar o estado existente para o novo backend.	**init -reconfigure** Apontar para o novo backend sem migrar o estado anterior.
plan -destroy Só mostrar o que seria destruído.	**apply -destroy / destroy** Destruir de fato (pede confirmação).
apply (sem arquivo) Gera um plano na hora e pede yes.	**apply tfplan** Aplica exatamente o plano salvo, sem prompt; recusa se obsoleto.
fmt -check Só indicar arquivos fora do padrão (CI).	**fmt -diff** Mostrar as diferenças (ainda reescreve sem -check).
apply -replace Forma atual de recriar um recurso, visível no plano.	**taint** Obsoleto.
-input=false Não esperar valores de variáveis: falhar em automação.	**-auto-approve** Não esperar o yes de confirmação do apply/destroy.

Números que caem

Item	Valor
Código de saída do plan -detailed-exitcode quando há mudanças	**2**
Código de saída de fmt -check com arquivos fora do padrão	**Não zero**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que validate checa credenciais ou compara recursos reais com o estado.
Como não cair: validate não acessa a nuvem; credenciais inválidas aparecem no plan/apply e comparar com a infraestrutura real é papel do plan.

⚠ **Alerta de prova:** Marcar 'falso' para 'validate precisa de init'.
Como não cair: Verdadeiro: sem init não há providers para validar; em CI use init -backend=false.

⚠ **Alerta de prova:** Achar que apply tfplan pede confirmação.
Como não cair: Falso: o plano salvo é aplicado sem prompt, por isso revise antes.

⚠ **Alerta de prova:** Achar que plan -destroy executa a destruição.
Como não cair: Só mostra a prévia; o equivalente de destroy é apply -destroy.

⚠ **Alerta de prova:** Usar terraform state rm para destruir um recurso.
Como não cair: state rm só tira do estado; o recurso continua na nuvem.

⚠ **Alerta de prova:** Achar que fmt -diff serve para CI sem alterar arquivos.
Como não cair: -diff mostra diferenças mas ainda reescreve; para só verificar use -check.

⚠ **Alerta de prova:** Achar que fmt atualiza providers, remove código não usado ou reordena blocos.
Como não cair: fmt só ajusta estilo; atualizar providers é init -upgrade.

⚠ **Alerta de prova:** Achar que após falha parcial no apply o Terraform desfaz o que criou.
Como não cair: Não há rollback: os recursos criados ficam no estado; corrija e rode apply de novo.

⚠ **Alerta de prova:** Achar que o init repetido apaga o estado ou recria o lock.
Como não cair: init é seguro para repetir: mantém o lock e não apaga estado nem cria recursos.

⚠ **Alerta de prova:** Escolher opções inventadas: -yes, -force, -no-prompt, -save, -file, -only, -fail-on-diff, -silent.
Como não cair: As reais: -auto-approve, -out, -target, -detailed-exitcode.

⚠ **Alerta de prova:** Usar -replace=aws_instance.web[0] para instância de for_each.
Como não cair: Índice é de count; para for_each use a chave: aws_instance.web["api"].

⚠ **Alerta de prova:** Achar que ~ indica destruição e recriação.
Como não cair: ~ é alteração no local; -/+ é destruir e recriar; + é só criar.

⚠ **Alerta de prova:** Usar -target como rotina para ganhar tempo.
Como não cair: A recomendação é reservar -target para exceções, como recuperar erros.

⚠ **Alerta de prova:** Passar arquivo de backend com -var-file ou no plan.
Como não cair: Configuração parcial de backend: terraform init -backend-config=backend.hcl.

Como cai na prova

📝 **Como cai:** Cenário de pipeline com uma necessidade (sem prompt, falhar sem variável, código de saída, checar formato).
Como resolver: Mapeie a necessidade para a opção: -auto-approve, -input=false, -detailed-exitcode, fmt -check. Descarte opções que não existem.

📝 **Como cai:** O que um comando NÃO faz ou qual o limite dele.
Como resolver: init não cria recursos; plan não altera recursos; validate não acessa a nuvem; fmt não muda lógica.

📝 **Como cai:** Verdadeiro ou falso sobre confirmação, init ou fmt.
Como resolver: apply tfplan não pede yes; validate precisa de init; fmt por padrão reescreve.

📝 **Como cai:** Resultado de plan ou situação após apply/destroy.
Como resolver: Após destroy sem mudar código, o plan propõe criar tudo; falha parcial deixa o criado no estado; plano obsoleto é recusado.

📝 **Como cai:** Qual opção do init usar após mudar backend, provider ou rede.
Como resolver: Migrar estado: -migrate-state; não migrar: -reconfigure; atualizar versões: -upgrade; sem internet: -plugin-dir; completar backend: -backend-config.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
pipeline sem digitar yes	**-auto-approve**
pipeline falhar em vez de esperar variável	**-input=false**
código de saída 2 quando há mudanças	**plan -detailed-exitcode**
pipeline falhar se .tf fora do padrão sem alterar	**terraform fmt -check (com -recursive)**
aplicar exatamente o que foi revisado	**plan -out=tfplan e apply tfplan**
recriar instância com problema sem mudar código	**terraform apply -replace=ENDEREÇO**
novo backend sem migrar estado	**init -reconfigure**
servidor sem internet no init	**init -plugin-dir=CAMINHO**
validate sem acessar backend remoto	**init -backend=false**

atualizar provider dentro da restrição e o lock	**init -upgrade**
plano para ferramenta de análise	**plan -out=tfplan + show -json tfplan**
testar cidrsubnet ou outra função	**terraform console**
pular consulta aos objetos reais em infra grande	**-refresh=false (risco de ignorar desvio)**

🎯 **Se você lembrar de só uma coisa deste capítulo...**

- Plano salvo: apply sem prompt e recusado se o código ou o estado mudaram.
- validate precisa de init, não acessa a nuvem e não testa credenciais.
- fmt -check não reescreve; fmt sem flag reescreve .tf e .tfvars do diretório atual.
- apply -replace substitui o taint obsoleto; -target é só para exceções.
- Sem rollback: falha parcial deixa no estado o que já foi criado.

Essenciais para a prova (checklist)

☐ Sei a sequência write, plan, apply e onde entra o init.

☐ Sei todas as opções de init cobradas: -upgrade, -reconfigure, -migrate-state, -backend=false, -backend-config, -plugin-dir.

☐ Sei o que validate verifica e o que não verifica.

☐ Sei o comportamento padrão de fmt e das opções -check, -diff e -recursive.

☐ Sei salvar e aplicar um plano e o que acontece se ele ficar obsoleto.

☐ Sei ler os símbolos +, ~, -/+ e o resumo add/change/destroy.

☐ Sei recriar recursos com -replace, inclusive instâncias de for_each.

☐ Sei diferenciar destroy, apply -destroy, plan -destroy e state rm.

☐ Sei as opções de automação -auto-approve, -input=false e -detailed-exitcode.

☐ Sei o que significa known after apply e por que não indica erro nem segredo.

Questões de revisão do capítulo

R2.1 · Fluxo de Trabalho Principal

O backend foi alterado e a equipe quer apontar para o novo local sem migrar o estado anterior. Qual opção do init usar?

A. -get=false
B. -migrate-state
C. -reconfigure
D. -upgrade

R2.2 · Fluxo de Trabalho Principal

Qual opção do terraform plan faz o comando retornar código 2 quando há mudanças, útil em pipelines?

A. -fail-on-diff
B. -detailed-exitcode
C. -exit-on-change
D. -strict-exit

R2.3 · Fluxo de Trabalho Principal

Um revisor aprovou o plano salvo de uma mudança. Qual é o próximo passo do fluxo?

A. Apagar o estado para gerar um novo plano
B. Rodar terraform fmt no plano salvo
C. Aplicar o plano salvo com apply
D. Rodar terraform init -upgrade outra vez

R2.4 · Fluxo de Trabalho Principal

Qual comando verifica se a configuração é sintaticamente válida e internamente consistente, sem acessar serviços remotos?

A. terraform import
B. terraform apply
C. terraform validate
D. terraform login

Respostas comentadas

R2.1. C) -reconfigure

Por quê: Use -migrate-state quando quiser levar o estado para o novo backend.

✗ A) Incorreta. Pula módulos.
✗ B) Incorreta. Copiaria o estado.
✗ D) Incorreta. Atualiza providers e módulos.

> **Dica de prova:** Novo backend sem migrar: -reconfigure.

R2.2. B) -detailed-exitcode

Por quê: Assim o pipeline pode decidir se precisa de aprovação para aplicar.

✗ A) Incorreta. Opção inexistente.
✗ C) Incorreta. Opção inexistente.
✗ D) Incorreta. Opção inexistente.

> **Dica de prova:** Código de saída por mudanças: -detailed-exitcode.

R2.3. C) Aplicar o plano salvo com apply

Por quê: Aplicar o arquivo garante que nada diferente do revisado será executado.

✗ A) Incorreta. Perderia o mapeamento.
✗ B) Incorreta. Não formata planos.
✗ D) Incorreta. Pode mudar providers.

> **Dica de prova:** Plano aprovado: apply do arquivo.

R2.4. C) terraform validate

Por quê: O validate é útil em pipelines de CI antes do plan.

✗ A) Incorreta. Traz recursos ao estado.
✗ B) Incorreta. Altera a infraestrutura.
✗ D) Incorreta. Obtém token do HCP Terraform.

> **Dica de prova:** Checar a configuração sem nuvem: terraform validate.

03

CAPÍTULO 3

Fundamentos do Terraform

Neste domínio eu preciso dominar providers (instalação, versão, alias, origem, lock) e o propósito do estado. A prova cobra muito required_providers, os operadores de versão (~>, >=, =), o arquivo .terraform.lock.hcl e o diretório .terraform, além do papel de Core x provider. Também aparecem cenários de 'apaguei o estado' e 'removi o recurso do código'. Onde mais erro: confundir o que vai e o que não vai para o Git (lock sim, .terraform e tfstate não) e ler errado ~> 5.1 versus ~> 5.1.0.

Neste capítulo
18 tópicos-chave · 13 pegadinhas · 7 macetes · 13 gatilhos · 4 questões comentadas

Tópicos-chave

Terraform Core x provider

O Core lê a configuração e o estado, calcula o grafo e monta o plano; o provider é um plugin que chama as APIs e implementa os tipos de resource e data source. Providers recentes também oferecem funções próprias.

required_providers

Fica dentro do bloco terraform e declara origem (source) e restrição de versão de cada provider, como hashicorp/aws ~> 5.0. O bloco provider configura (região, por exemplo); required_providers diz de onde baixar e qual versão aceitar.

Endereço de origem do provider

Formato HOSTNAME/NAMESPACE/TIPO. Quando o hostname é omitido, usa-se registry.terraform.io, como em hashicorp/aws. Provider não declarado é buscado no namespace hashicorp do registry público.

Nome local

Em aws = { source = "hashicorp/aws" }, aws é o nome local: como o módulo referencia o provider. Por convenção é igual ao tipo do provider.

Operador ~> (pessimista)

Permite variar só o componente mais à direita informado. ~> 5.1 aceita de 5.1 até antes de 6.0; ~> 5.1.0 aceita só patches, abaixo de 5.2.0.

Versão exata

version = "5.31.0" fixa exatamente; o operador = explícito tem o mesmo efeito. >= permite qualquer versão maior e != exclui a versão.

required_version

Fica no bloco terraform e restringe a versão da CLI, por exemplo required_version = ">= 1.12". version no bloco provider não restringe a CLI.

.terraform.lock.hcl

Criado pelo terraform init para registrar as versões exatas e hashes dos providers selecionados. Deve ir ao controle de versão para todos usarem as mesmas versões; não contém senhas e não é recriado pelo plan.

Diretório .terraform

Onde o init instala os plugins de providers (e os módulos baixados). Não deve ser versionado.

Provider com alias

Para usar o mesmo provider em duas regiões, declare um segundo bloco provider com alias. No resource, selecione com provider = aws.oeste; sem isso, o recurso usa a configuração padrão.

Herança de provider em módulos

Módulo filho sem bloco provider herda a configuração padrão do raiz. Configurações com alias precisam ser repassadas com o argumento providers no bloco module.

Provider em módulo reutilizável

O módulo declara seus providers em required_providers no próprio módulo, com só uma versão mínima (>=). O raiz combina as restrições e o lock define a versão exata; versão exata no filho gera conflitos.

Funções de provider

Disponíveis desde o Terraform 1.8, chamadas com provider::NOME::FUNÇÃO(args), como provider::aws::arn_parse(var.arn).

Estado: finalidade

Mapear a configuração aos objetos reais. Também guarda atributos, valores de outputs e metadados (dependências, provider de cada recurso) e melhora o desempenho. É gravado em JSON.

lineage e serial

lineage identifica o estado e evita misturar estados de configurações diferentes; serial aumenta a cada gravação do estado.

Refresh no plan

Por padrão, antes de comparar configuração e estado, o plan atualiza o estado com os objetos reais. -refresh=false pula essa leitura, com risco de plano desatualizado.

Workspaces da CLI

Mesma configuração, cada workspace com seu próprio estado; o lock é compartilhado e o provider é o mesmo. terraform.workspace pode variar nomes por ambiente.

Instalação de provider interno

Provider fora do registry público é instalado por espelho ou registry privado. A configuração provider_installation da CLI define espelhos de rede ou de sistema de arquivos.

Macetes para memorizar

💡 Operador ~>

O último número informado é o único que anda: ~> 5.1 anda o 1 (até 6.0); ~> 5.1.0 anda o 0 (até 5.2.0).

💡 O que vai para o Git

Código e lock SIM; .terraform e tfstate NÃO.

💡 Core x provider

Core pensa (grafo e plano), provider fala (API).

💡 Alias

alias no provider, provider no resource, providers no module.

💡 Origem do provider

H-N-T: Hostname/Namespace/Tipo; sem hostname = registry.terraform.io.

💡 Versão em módulo reutilizável

Filho pede o mínimo (>=), raiz e lock decidem o exato.

💡 Estado

Estado é o mapa código ↔ mundo real; sem mapa, o Terraform acha que nada existe.

Tabela de memorização: não confunda

Quando é...	...e quando é
required_providers Declarar origem (source) e restrição de versão.	**bloco provider** Configurar argumentos como região padrão.
required_version Restringir a versão da CLI do Terraform.	**version em required_providers** Restringir a versão de um provider.
~> 5.1 Aceita 5.1 ou superior, abaixo de 6.0.	**~> 5.1.0** Só patches: 5.1.x, abaixo de 5.2.0.
.terraform.lock.hcl Versionar no Git: fixa versões e hashes dos providers.	**diretório .terraform** Não versionar: cache local de plugins e módulos.
terraform version Mostrar a versão da CLI e dos providers instalados.	**terraform providers** Listar providers exigidos pela configuração e pelos módulos, com restrições.
provider = aws.oeste (no resource) Escolher a configuração com alias para um recurso.	**providers (no bloco module)** Repassar configuração com alias para um módulo filho.
>= no módulo reutilizável Módulo filho declara só versão mínima.	**versão exata no raiz/lock** O raiz e o arquivo de lock definem a versão exata usada.

Números que caem

Item	Valor
Funções de provider (provider::NOME::FUNÇÃO)	**Desde o Terraform 1.8**
Hostname padrão do registry	**registry.terraform.io**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que ~> 5.1 aceita 6.x e 7.x.
Como não cair: ~> trava o componente à esquerda do último informado: ~> 5.1 fica abaixo de 6.0.

⚠ **Alerta de prova:** Achar que o arquivo de lock não deve ir ao Git porque 'contém senhas' ou 'é recriado a cada plan'.
Como não cair: O lock não contém senhas, não é recriado pelo plan e deve ser versionado.

⚠ **Alerta de prova:** Marcar 'verdadeiro' para enviar terraform.tfstate ao Git.
Como não cair: Falso: o estado pode conter segredos e causar conflitos; use backend remoto com bloqueio e criptografia.

⚠ **Alerta de prova:** Usar alias = aws.oeste ou source = aws.oeste no bloco resource.
Como não cair: alias fica no bloco provider; no resource o argumento é provider = aws.oeste.

⚠ **Alerta de prova:** Criar uma variável region com dois valores ou um segundo bloco terraform para outra região.
Como não cair: Segunda região do mesmo provider = segundo bloco provider com alias.

⚠ **Alerta de prova:** Achar que o estado apagado é recuperado da nuvem ou que o plan não muda nada.
Como não cair: Sem estado não há mapeamento: o plan propõe criar tudo de novo; seria preciso importar para evitar duplicatas.

⚠ **Alerta de prova:** Achar que o Git ou o provider avisam o Terraform que um recurso saiu do código.
Como não cair: O recurso segue no estado e não está no código: por isso o plano propõe destruí-lo.

⚠ **Alerta de prova:** Achar que a ordem de destruição vem da ordem alfabética ou da data de criação.
Como não cair: Para recurso removido do código, a ordem vem das dependências gravadas no estado.

⚠ **Alerta de prova:** Fixar versão exata do provider dentro de módulo reutilizável.
Como não cair: Gera conflitos entre módulos; no módulo, declare só versão mínima com >=.

⚠ **Alerta de prova:** Usar terraform init -upgrade para gerar hashes de outra plataforma.
Como não cair: Lock multiplataforma: terraform providers lock -platform=linux_amd64 -platform=darwin_arm64.

⚠ **Alerta de prova:** Achar que provider não declarado vem de espelho local padrão ou do diretório de módulos.
Como não cair: Vem do namespace hashicorp no registry público; espelhos exigem configuração.

⚠ **Alerta de prova:** Achar que o estado guarda código dos módulos, binários de providers ou histórico de commits.
Como não cair: Estado guarda mapeamento, atributos, outputs e metadados; binários ficam em .terraform e o histórico no Git.

⚠ **Alerta de prova:** Achar que cada workspace da CLI usa outro diretório de código ou outro lock.
Como não cair: O código e o lock são os mesmos; só o estado é separado.

Como cai na prova

📝 **Como cai:** Restrição de versão (~> 5.1, ~> 5.1.0, = 5.31.0, >= 5.31.0) e pergunta qual faixa ela aceita.
Como resolver: Conte os números informados: o último é o que varia com ~>; sem operador ou com = é exato; >= não tem teto.

📝 **Como cai:** Onde declarar algo: origem/versão, região, versão da CLI.
Como resolver: Origem e versão do provider: required_providers; região: bloco provider; versão da CLI: required_version. required_providers e required_version ficam dentro do bloco terraform; o bloco provider fica fora dele.

📝 **Como cai:** Cenário 'o que acontece no próximo plan' após apagar estado ou remover recurso do código.
Como resolver: Pergunte: o Terraform tem o mapeamento? Sem estado: criar tudo. No estado e fora do código: destruir.

📝 **Como cai:** Verdadeiro ou falso sobre versionar tfstate ou lock.
Como resolver: tfstate fora do Git (segredos e conflitos); lock dentro (mesmas versões para todos).

📝 **Como cai:** Módulo filho e provider: qual configuração ele usa.
Como resolver: Sem bloco provider, herda o padrão do raiz; com alias, repasse via providers no bloco module.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
origem e versão de provider, hashicorp/aws ~> 5.0	**required_providers dentro de terraform**
região padrão do provider	**Bloco provider**
duas regiões da AWS	**Segundo bloco provider com alias + provider = aws.ALIAS**
versões exatas registradas pelo init	**.terraform.lock.hcl (versionar)**
onde ficam os plugins baixados	**Subdiretório .terraform (não versionar)**
exigir CLI 1.12 ou superior	**required_version no bloco terraform**
macOS na equipe e Linux no pipeline	**terraform providers lock -platform**
provider interno fora do registry público	**Espelho ou registry privado (provider_installation)**
provider::aws::arn_parse	**Função de provider, Terraform 1.8+**
sufixo aleatório para nomes	**Provider random ao lado do aws**

lineage	**Evitar misturar estados de configurações diferentes**
dev e prd com a mesma configuração na CLI	**Workspaces da CLI: um estado por workspace**
listar providers exigidos com restrições	**terraform providers**

🎯 Se você lembrar de só uma coisa deste capítulo...

- ~> 5.1 = de 5.1 até antes de 6.0; ~> 5.1.0 = só 5.1.x.
- Lock vai para o Git; .terraform e terraform.tfstate não vão.
- Segunda região do mesmo provider: bloco provider com alias e provider = aws.ALIAS no recurso.
- Sem estado o Terraform não conhece os recursos: propõe criar tudo de novo.
- Estado é JSON e guarda mapeamento, atributos, outputs e metadados de dependência.

Essenciais para a prova (checklist)

- ☐ Sei ler as restrições ~> 5.1, ~> 5.1.0, = 5.31.0 e >= 5.31.0.
- ☐ Sei a diferença entre required_providers, bloco provider e required_version.
- ☐ Sei o formato HOSTNAME/NAMESPACE/TIPO e o padrão registry.terraform.io.
- ☐ Sei o que é o .terraform.lock.hcl, quem o cria e que deve ser versionado.
- ☐ Sei usar alias e selecionar com provider = aws.ALIAS.
- ☐ Sei que módulo filho herda o provider padrão e recebe alias via providers.
- ☐ Sei a finalidade do estado e o que ele guarda (incluindo lineage e serial).
- ☐ Sei prever o plan após apagar o estado ou remover um recurso do código.
- ☐ Sei gerar lock multiplataforma com terraform providers lock -platform.
- ☐ Sei a sintaxe de funções de provider (provider::NOME::FUNÇÃO).

Questões de revisão do capítulo

R3.1 · Fundamentos do Terraform

Uma configuração precisa gerar um sufixo aleatório para nomes de buckets, além de criar recursos na AWS. O que usar?

A. Um bloco locals com a função now
B. Uma variável com sensitive = true
C. O provider random ao lado do aws
D. O comando terraform fmt -random

R3.2 · Fundamentos do Terraform

A empresa usa um provider interno que não está no registry público. Como o Terraform pode instalá-lo?

A. Por um output com o endereço do provider
B. Pela variável TF_LOG apontando o arquivo
C. Pelo bloco backend com o caminho do binário
D. Por espelho ou registry privado

R3.3 · Fundamentos do Terraform

Com workspaces da CLI, como os ambientes dev e prd usam a mesma configuração sem misturar recursos?

A. Cada workspace usa outro diretório de código
B. Cada workspace grava outro arquivo de lock
C. Cada workspace tem o seu próprio estado
D. Cada workspace tem um provider diferente

R3.4 · Fundamentos do Terraform

Além do mapeamento dos recursos, quais informações o estado do Terraform guarda?

A. Atributos e valores de outputs
B. O código-fonte completo dos módulos
C. Os binários dos providers instalados
D. O histórico de commits do repositório

Respostas comentadas

R3.1. C) O provider random ao lado do aws

Por quê: Os valores do random ficam no estado e só mudam quando seus argumentos mudam.

✗ A) Incorreta. Função inexistente.
✗ B) Incorreta. Não gera valores.
✗ D) Incorreta. Opção inexistente.

> **Dica de prova:** Valores aleatórios: provider random.

R3.2. D) Por espelho ou registry privado

Por quê: A configuração provider_installation da CLI define espelhos de rede ou de sistema de arquivos.

✗ A) Incorreta. Outputs não instalam nada.

✗ B) Incorreta. TF_LOG trata de logs.

✗ C) Incorreta. backend trata de estado.

> **Dica de prova:** Provider interno: espelho ou registry privado.

R3.3. C) Cada workspace tem o seu próprio estado

Por quê: terraform.workspace pode ser usado no código para variar nomes por ambiente.

✗ A) Incorreta. O código é o mesmo.

✗ B) Incorreta. O lock é compartilhado.

✗ D) Incorreta. O provider é o mesmo.

> **Dica de prova:** Workspaces da CLI: estados separados.

R3.4. A) Atributos e valores de outputs

Por quê: Também registra metadados como dependências e o provider de cada recurso.

✗ B) Incorreta. O código fica nos arquivos .tf.

✗ C) Incorreta. Ficam em .terraform.

✗ D) Incorreta. Fica no Git.

> **Dica de prova:** Estado: atributos, outputs e metadados.

04

CAPÍTULO 4

Gerenciamento de State

Aqui eu preciso saber onde o estado mora (backend local por padrão, backends remotos S3, azurerm, gcs, HCP Terraform), como ele é protegido (locking, criptografia, versionamento, dado sensível) e como lidar com desvios (drift) com plan e apply -refresh-only. A prova cobra muitos argumentos concretos: key, encrypt, use_lockfile, path, -lock-timeout, force-unlock e init -migrate-state. Onde mais erro: achar que dá para usar variável no backend, confundir state pull com push e achar que o apply aceita automaticamente a mudança manual.

Neste capítulo
18 tópicos-chave · 13 pegadinhas · 6 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Backend local

Padrão quando nenhum backend é configurado: arquivo terraform.tfstate local, com terraform.tfstate.backup guardando a versão anterior. O argumento path muda o caminho. Usa bloqueio do sistema de arquivos, que não protege outras máquinas.

Backend remoto

Configurado no bloco backend dentro de terraform. Principal vantagem: estado compartilhado com segurança, em geral com criptografia, versionamento e bloqueio. O init continua necessário.

Backend S3

key define o caminho do objeto de estado no bucket (uma por configuração ou ambiente); encrypt = true ativa criptografia no servidor; use_lockfile = true ativa bloqueio nativo sem DynamoDB, que está obsoleto para lock a partir do Terraform 1.11.

Outros backends

azurerm guarda o estado em contêiner do Azure Blob Storage e bloqueia por lease do blob; gcs guarda em bucket do Google Cloud Storage com bloqueio.

Locking

Impede execuções simultâneas no mesmo estado; adquirido por operações que podem escrever no estado (validate, fmt e login não usam estado). -lock=false desativa e é arriscado; -lock-timeout=5m espera o bloqueio ser liberado.

force-unlock

terraform force-unlock ID_DO_LOCK libera um bloqueio preso após execução interrompida. A mensagem de erro mostra o ID, quem criou e quando. Use só com certeza de que nada está rodando.

Estado como dado sensível

Pode conter senhas e chaves em texto claro. Use backend com criptografia e controle de acesso e não envie ao Git.

Configuração parcial de backend

O backend é lido antes das variáveis, por isso var.bucket não funciona ali. Valores por ambiente vêm de -backend-config no init.

Migração de backend

terraform init -migrate-state move o estado existente para o novo backend; -reconfigure ignora o estado anterior.

Drift (desvio)

Mudança feita fora do Terraform; o plan mostra 'Objects have changed outside of Terraform'. Sem mudar o código, o próximo apply reverte para o valor do código; VM apagada no console é recriada pelo plan.

-refresh-only

terraform plan -refresh-only mostra diferenças para atualizar o estado sem propor mudanças nos recursos; terraform apply -refresh-only grava as diferenças aceitas. O antigo terraform refresh faz o mesmo sem revisão e não é recomendado.

Manter mudança manual

Atualize o código para refletir a mudança (ou use ignore_changes); depois o plan volta a ficar limpo. Editar o estado à mão é arriscado.

state pull e state push

state pull mostra o estado atual, local ou remoto, na saída padrão. state push envia um estado ao backend e pode sobrescrever um estado mais novo: é para recuperações excepcionais.

Não editar o tfstate

O arquivo é JSON legível, mas editá-lo à mão pode corromper o estado e perder o mapeamento. Use terraform state, moved, import e removed.

terraform_remote_state

Data source que lê os outputs do estado de outra configuração; só os outputs do módulo raiz dela ficam acessíveis.

Workspaces locais

Com backend local, o workspace default usa terraform.tfstate na raiz e os demais ficam em terraform.tfstate.d.

Recurso órfão

Retirado do código e do estado, mas existindo na nuvem: o Terraform não o vê mais nem o gerencia; continua gerando custo e deve ser tratado manualmente ou importado.

HCP Terraform como backend

Oferece histórico de versões do estado, bloqueio automático durante cada run e criptografia em repouso; também é possível bloquear o workspace manualmente pela interface.

Macetes para memorizar

💡 Refresh-only

Plan olha, apply assina: plan -refresh-only mostra, apply -refresh-only grava.

💡 Pull x push

Pull puxa para ver; push empurra e pode atropelar.

💡 Backend S3

KEY onde mora, ENCRYPT tranca o cofre, USE_LOCKFILE tranca a porta.

💡 Backends de nuvem

AWS = s3, Azure = azurerm, Google = gcs.

💡 Drift

Mudou na mão e não no código? O apply desfaz; quer manter, leve ao código.

💡 Variáveis no backend

Backend acorda antes das variáveis: complete com -backend-config.

Tabela de memorização: não confunda

Quando é...	...e quando é
plan -refresh-only Ver o desvio sem propor mudanças nos recursos.	**apply -refresh-only** Gravar no estado as diferenças aceitas.
state pull Exibir o estado atual na saída padrão.	**state push** Enviar um estado ao backend (excepcional; pode sobrescrever o mais novo).
-lock-timeout=5m Esperar até 5 minutos pelo bloqueio.	**-lock=false** Desativar o bloqueio: risco de escrita concorrente.
Atualizar o código Manter a mudança manual.	**Rodar apply sem mudar código** Reverter o recurso para o valor do código.
terraform.tfstate.backup Versão anterior do estado local.	**terraform.tfstate.d** Estados dos workspaces não default no backend local.
use_lockfile (S3) Bloqueio nativo sem DynamoDB.	**versionamento do bucket** Recuperar versão anterior de estado corrompido.

Números que caem

Item	Valor
DynamoDB para lock no backend S3	**Obsoleto a partir do Terraform 1.11**
Exemplo de espera por bloqueio	**-lock-timeout=5m**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Usar var.bucket no bloco backend.
Como não cair: O backend é lido antes das variáveis; use configuração parcial com -backend-config.

⚠ **Alerta de prova:** Achar que o backend local não tem bloqueio ou que bloqueia outras máquinas.
Como não cair: Tem bloqueio do sistema de arquivos, que só vale no disco local.

⚠ **Alerta de prova:** Achar que backend remoto tira os dados sensíveis do estado.
Como não cair: O estado continua podendo conter segredos; protegê-lo é papel de criptografia e acesso.

⚠ **Alerta de prova:** Achar que o apply reescreve o código com a tag alterada à mão.
Como não cair: O código não é alterado: o apply devolve a tag ao valor do código.

⚠ **Alerta de prova:** Achar que a VM apagada no console some do plano porque 'o estado já sabe'.
Como não cair: O refresh detecta a falta e o plan propõe criá-la de novo.

⚠ **Alerta de prova:** Usar state rm para liberar o lock.
Como não cair: state rm remove recursos do estado; para lock preso é terraform force-unlock ID.

⚠ **Alerta de prova:** Achar que criptografia ou bloqueio de acesso público recuperam versões antigas.
Como não cair: Só o versionamento de objetos no bucket permite recuperar uma versão anterior.

⚠ **Alerta de prova:** Achar que state push é rotina segura.
Como não cair: Pode sobrescrever estado mais novo; é para recuperações excepcionais.

⚠ **Alerta de prova:** Achar que o arquivo de estado é binário ou que editar apaga recursos na nuvem.
Como não cair: É JSON; editar não mexe na nuvem, mas pode corromper o mapeamento.

⚠ **Alerta de prova:** Usar path no backend S3.
Como não cair: path é do backend local; no S3 o caminho é key.

⚠ **Alerta de prova:** Escolher lock_table ou force_lock para bloqueio no S3.
Como não cair: O argumento é use_lockfile = true; DynamoDB para lock está obsoleto a partir do 1.11.

⚠ **Alerta de prova:** Achar que terraform validate ou fmt adquirem lock.
Como não cair: Só operações que podem escrever no estado adquirem bloqueio.

⚠ **Alerta de prova:** Achar que recurso fora do código e do estado será destruído ou importado sozinho.
Como não cair: Vira órfão: o Terraform não o vê mais.

Como cai na prova

📝 **Como cai:** Erro de estado bloqueado ou execução concorrente.
Como resolver: Esperar: -lock-timeout; lock preso: force-unlock com o ID; nunca -lock=false em backend compartilhado.

📝 **Como cai:** Mudança manual detectada: o que acontece ou o que fazer.
Como resolver: Sem mudar o código, o apply reverte; para manter, atualize o código; para só sincronizar o estado, use -refresh-only.

📝 **Como cai:** Argumento de backend específico (S3, local, Azure, GCS).
Como resolver: S3: key, encrypt, use_lockfile; local: path; Azure: azurerm; Google: gcs. Descarte nomes inventados.

📝 **Como cai:** Onde fica o estado e o arquivo de backup.
Como resolver: Sem backend: terraform.tfstate local; anterior: terraform.tfstate.backup; workspaces: terraform.tfstate.d.

📝 **Como cai:** Por que o estado é sensível ou por que não editar.
Como resolver: Pode ter segredos em texto claro; editar à mão pode corromper o mapeamento. Use comandos e blocos próprios.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
estado bloqueado após execução interrompida	**terraform force-unlock ID_DO_LOCK**
segunda pessoa esperar até 5 minutos	**-lock-timeout=5m**
trocar backend local por remoto levando o estado	**terraform init -migrate-state**
Objects have changed outside of Terraform	**Drift: mudança fora do Terraform**
atualizar estado sem mudar recursos	**plan -refresh-only e apply -refresh-only**
ler outputs de outra configuração	**terraform_remote_state**
caminho do estado no bucket S3	**key**
criptografia no S3	**encrypt = true**
lock no S3 sem DynamoDB	**use_lockfile = true**
Azure Blob Storage	**Backend azurerm**
Google Cloud Storage	**Backend gcs**
estado corrompido no bucket	**Versionamento de objetos no bucket**
equipe com cópias locais do estado	**Estados divergentes: usar backend remoto com bloqueio**
workspaces não default com backend local	**terraform.tfstate.d**

🎯 **Se você lembrar de só uma coisa deste capítulo...**

- O bloco backend não aceita variáveis: use -backend-config.
- plan -refresh-only mostra; apply -refresh-only grava; terraform refresh é o antigo sem revisão.
- S3: key, encrypt = true e use_lockfile = true.
- Estado pode ter segredos em texto claro: criptografia, acesso restrito e nunca no Git.
- force-unlock exige o ID do bloqueio e certeza de que nada está rodando.

Essenciais para a prova (checklist)

- ☐ Sei onde o estado fica sem backend e o que é o tfstate.backup.
- ☐ Sei configurar backend remoto e por que não aceita variáveis.
- ☐ Sei os argumentos key, encrypt e use_lockfile do backend S3.
- ☐ Sei os nomes dos backends azurerm e gcs.
- ☐ Sei para que serve o locking e como usar -lock-timeout e force-unlock.
- ☐ Sei migrar estado com init -migrate-state.
- ☐ Sei tratar drift com -refresh-only ou atualizando o código.
- ☐ Sei a diferença entre state pull e state push.
- ☐ Sei ler outputs de outra configuração com terraform_remote_state.

Questões de revisão do capítulo

R4.1 · Gerenciamento de State

Um plan mostra que uma mudança manual em produção deve ser mantida. Qual é a forma correta de incorporá-la?

A. Ignorar o aviso em todos os próximos planos
B. Editar o estado à mão para ocultar o desvio
C. Atualizar o código para refletir a mudança
D. Rodar apply para reverter e depois refazer

R4.2 · Gerenciamento de State

Durante um plan, aparece a mensagem Objects have changed outside of Terraform. O que ela indica?

A. Houve mudanças feitas fora do Terraform
B. O estado foi apagado do backend remoto da equipe
C. O código tem erro de sintaxe grave
D. O provider precisa de nova versão

R4.3 · Gerenciamento de State

Qual é um benefício de usar o HCP Terraform como backend de estado?

A. Histórico de versões do estado e bloqueio
B. Dispensa de autenticação para ler o estado
C. Remoção automática de dados sensíveis
D. Estado gravado no Git junto com o código

R4.4 · Gerenciamento de State

Qual afirmação sobre o bloqueio de estado no backend local é correta?

A. Usa o bloqueio do sistema de arquivos
B. Bloqueia o estado de outras máquinas da rede
C. Exige uma tabela externa de bloqueio
D. Não existe bloqueio no backend local

Respostas comentadas

R4.1. C) Atualizar o código para refletir a mudança

Por quê: Depois de ajustar o código, o plan deve voltar a ficar sem mudanças.

✗ A) Incorreta. O desvio segue lá.
✗ B) Incorreta. Arriscado e não resolve.
✗ D) Incorreta. Perde a mudança.

> **Dica de prova:** Manter mudança manual: levar ao código.

R4.2. A) Houve mudanças feitas fora do Terraform

Por quê: Decida se atualiza o código para refletir a mudança ou se deixa o apply revertê-la.

✗ B) Incorreta. Não é o significado.
✗ C) Incorreta. Seria erro de validação.
✗ D) Incorreta. Não é o significado.

> **Dica de prova:** Mudança fora do Terraform: drift.

R4.3. A) Histórico de versões do estado e bloqueio

Por quê: O estado também fica criptografado em repouso.

✗ B) Incorreta. Exige autenticação.
✗ C) Incorreta. Não remove.
✗ D) Incorreta. Não é o caso.

> **Dica de prova:** HCP como backend: versões e lock.

R4.4. A) Usa o bloqueio do sistema de arquivos

Por quê: Para equipes, prefira backends remotos com bloqueio compartilhado.

✗ B) Incorreta. Só vale no disco local.

✗ C) Incorreta. Não é necessário.

✗ D) Incorreta. Existe bloqueio local.

> **Dica de prova:** Backend local: lock do sistema de arquivos.

05

CAPÍTULO 5

HCP Terraform

Preciso conhecer a organização do HCP Terraform (organização, projetos, workspaces, equipes, owners), os três fluxos de run (VCS, CLI e API), modos de execução (remoto, local, agentes) e os recursos de governança (policy sets com Sentinel ou OPA, níveis de enforcement, run tasks, estimativa de custos, health assessments). A prova também cobra o bloco cloud, terraform login, variable sets e precedência, compartilhamento de estado e credenciais dinâmicas por OIDC. Onde mais erro: confundir agentes com run tasks, soft com hard mandatory, e achar que o variable set vence a variável do workspace.

Neste capítulo

18 tópicos-chave · 13 pegadinhas · 6 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Workspace

Uma configuração com estado, variáveis e runs próprios. Prática comum: um workspace por ambiente (dev, hml, prd). Runs no mesmo workspace entram em fila e rodam um de cada vez, com estado bloqueado durante o run.

Projetos

Agrupam workspaces para organizar acesso de equipes e visibilidade. Dar acesso no projeto cobre todos os workspaces atuais e futuros da aplicação.

Equipes e owners

Permissões são dadas por equipe, no nível do projeto ou do workspace. A equipe owners tem acesso administrativo total; dê menor privilégio às demais.

Fluxo VCS

Runs disparados automaticamente por commits e pull requests. Pull request recebe plano especulativo, sem apply, exibido como verificação no PR. Em monorepo, defina o diretório de trabalho do workspace e, se quiser, limite os gatilhos a caminhos.

Fluxo CLI

terraform plan no terminal roda remotamente e a saída aparece localmente; o run fica registrado no workspace.

Fluxo API

Indicado quando outra ferramenta de automação envia versões de configuração e cria runs.

Bloco cloud

Dentro de terraform, conecta a configuração da CLI ao HCP Terraform indicando organização e workspaces: name para um só, tags para vários. Com estado local existente, o init oferece migrar o estado para o workspace.

terraform login

Obtém e salva localmente um token de API no arquivo de credenciais da CLI; terraform logout o remove. Também é necessário para baixar módulos do registry privado.

Modos de execução

Remoto: plan e apply na infraestrutura do HCP Terraform. Local: runs na máquina e estado no HCP. Agentes: executam runs em redes privadas, com conexões de saída, sem abrir portas de entrada.

Variáveis no workspace

Podem ser variáveis Terraform ou de ambiente; credenciais de provider costumam ser de ambiente marcadas como sensíveis. Variable sets compartilham variáveis com a organização, projetos ou workspaces.

Precedência de variable set

A variável definida no próprio workspace vence a do conjunto de variáveis, salvo quando o conjunto é marcado como prioritário.

Políticas como código

Sentinel e OPA (linguagem Rego), aplicadas por policy sets à organização, projetos ou workspaces, antes do apply. Níveis Sentinel: advisory só registra; soft mandatory permite que usuário autorizado prossiga; hard mandatory não pode ser sobreposto.

Run tasks

Integram ferramentas de terceiros (como segurança) ao run, em etapas antes e depois do plan; podem ser consultivas ou obrigatórias.

Ordem do run

Plan, estimativa de custos, políticas e apply. A estimativa de custos mostra o impacto financeiro e pode ser usada por políticas para barrar mudanças caras.

Health assessments

Avaliações de saúde verificam periodicamente se a infraestrutura real corresponde ao estado (detecção de desvio) e também checam blocos check continuamente.

Credenciais dinâmicas

Via OIDC, a nuvem confia no HCP Terraform como provedor de identidade, sem chaves de acesso de longa duração no workspace.

Compartilhamento de estado

Para um workspace ler outputs de outro, configure o compartilhamento de estado; a leitura usa tfe_outputs ou terraform_remote_state.

Registry privado e auditoria

Registry privado compartilha módulos e providers internos só com a organização. O histórico de runs mostra quem aprovou e aplicou; planos de maior porte oferecem trilha de auditoria da organização. Tags nos workspaces ajudam a filtrar entre centenas.

Macetes para memorizar

💡 Fluxos

V-C-A: VCS (commit), CLI (terminal), API (outra ferramenta).

💡 Enforcement Sentinel

Advisory avisa, Soft se dobra com autorização, Hard não dobra.

💡 Agentes x run tasks

Agente entra na rede privada; run task chama um especialista de fora.

💡 Ordem do run

P-C-P-A: Plan, Custos, Políticas, Apply.

💡 Execução local

Local roda em casa, guarda o cofre no HCP.

💡 Variable set

A casa (workspace) manda, a menos que o conjunto seja VIP (prioritário).

Tabela de memorização: não confunda

Quando é...	...e quando é
Agentes Executar runs em rede privada sem acesso público.	**Run tasks** Ferramenta de terceiros analisar o plano durante o run.

Soft mandatory Usuário autorizado pode sobrepor a falha.	**Hard mandatory** Falha bloqueia sempre; advisory só registra.
Execução remota Plan e apply no HCP Terraform.	**Execução local** Runs na máquina, estado no HCP.
Fluxo VCS Commits e PRs disparam runs.	**Fluxo API** Outra ferramenta envia configuração e cria runs.
Variável do workspace Vence por padrão.	**Variable set** Vence só se marcado como prioritário; serve para compartilhar entre workspaces.
name no bloco cloud Um único workspace.	**tags no bloco cloud** Vários workspaces.
Projetos Agrupar e dar acesso a equipes.	**Tags de workspace** Encontrar e filtrar workspaces.

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que o variable set vence a variável do workspace.
Como não cair: O workspace vence por padrão; só o conjunto prioritário sobrepõe.

⚠ **Alerta de prova:** Escolher agentes para integrar ferramenta de segurança de terceiros.
Como não cair: Agentes executam runs em redes privadas; integração com terceiros é run task.

⚠ **Alerta de prova:** Achar que advisory ou hard mandatory deixam prosseguir com aprovação.
Como não cair: Só soft mandatory permite que usuário autorizado prossiga.

⚠ **Alerta de prova:** Achar que pull request dispara apply.
Como não cair: PR gera plano especulativo, sem apply.

⚠ **Alerta de prova:** Inverter o modo local: runs no HCP e estado na máquina.
Como não cair: Local = runs na máquina, estado no HCP.

⚠ **Alerta de prova:** Achar que dois runs no mesmo workspace rodam juntos ou se cancelam.
Como não cair: Entram em fila, um de cada vez; há um só estado.

⚠ **Alerta de prova:** Achar que o init com bloco cloud apaga o estado local ou recria recursos.
Como não cair: O init oferece migrar o estado para o workspace.

⚠ **Alerta de prova:** Conectar a CLI com provider chamado hcp ou data source de workspace.
Como não cair: O certo é o bloco cloud dentro de terraform.

⚠ **Alerta de prova:** Escolher terraform token, auth ou connect.
Como não cair: O comando é terraform login (logout remove).

⚠ **Alerta de prova:** Dar acesso a cada workspace novo manualmente ou compartilhar token de usuário.
Como não cair: Dê acesso no projeto: cobre workspaces atuais e futuros.

⚠ **Alerta de prova:** Achar que plano especulativo é um fluxo do HCP Terraform.
Como não cair: Os fluxos são VCS, CLI e API.

⚠ **Alerta de prova:** Achar que políticas vêm antes do plan ou que o apply vem antes dos custos.
Como não cair: Ordem: plan, custos, políticas, apply.

⚠ **Alerta de prova:** Usar count no bloco cloud para vários workspaces.
Como não cair: cloud não aceita count; use tags.

Como cai na prova

📝 **Como cai:** Cenário de empresa com uma necessidade de governança ou integração.
Como resolver: Regras sobre o plano: políticas (Sentinel/OPA) em policy set; ferramenta de terceiros: run task; rede privada: agente; custos: estimativa.

📝 **Como cai:** Organização e acesso em escala.
Como resolver: Agrupar e dar acesso: projetos; encontrar: tags; acesso total: owners; compartilhar variáveis: variable sets.

📝 **Como cai:** O que acontece em um fluxo (PR aberto, plan na CLI, init com bloco cloud, dois runs juntos).
Como resolver: PR: plano especulativo; CLI: remoto com saída local; init: oferece migrar estado; runs: fila.

📝 **Como cai:** Precedência ou nível de política.
Como resolver: Workspace vence variable set salvo prioridade; só soft mandatory pode ser sobreposto.

📝 **Como cai:** Autenticação e credenciais.
Como resolver: CLI: terraform login; nuvem sem chaves longas: credenciais dinâmicas OIDC; credenciais de provider: variáveis de ambiente sensíveis.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
outra ferramenta envia configuração e dispara runs	**Fluxo orientado por API**
commits e pull requests disparam runs	**Fluxo VCS**
sem chaves de longa duração no workspace	**Credenciais dinâmicas via OIDC**
mesmas políticas em vários workspaces	**Policy set**
acesso a workspaces atuais e futuros da aplicação	**Permissão no projeto**
rede privada sem acesso público	**Agentes do HCP Terraform**
ferramenta de segurança analisa o plano	**Run tasks**
mesmas credenciais em vários workspaces	**Variable sets**
prosseguir com falha de política por usuário autorizado	**Soft mandatory**
verificar periodicamente desvio	**Health assessments**
impacto financeiro no run	**Estimativa de custos**
monorepo, workspace só para infra/rede	**Diretório de trabalho do workspace**
quem aprovou e aplicou	**Histórico de runs do workspace**
ler outputs de outro workspace	**Compartilhamento de estado + tfe_outputs/terraform_remote_state**

🎯 **Se você lembrar de só uma coisa deste capítulo...**

- Fluxos: VCS, CLI e API; PR gera plano especulativo sem apply.
- Variável do workspace vence variable set, salvo conjunto prioritário.
- Advisory registra, soft mandatory pode ser sobreposto, hard mandatory não.
- Agentes = rede privada; run tasks = ferramentas de terceiros no run.
- Ordem do run: plan, custos, políticas, apply.

Essenciais para a prova (checklist)

- ☐ Sei o que é workspace, projeto, equipe e a equipe owners.
- ☐ Sei diferenciar os fluxos VCS, CLI e API.
- ☐ Sei configurar o bloco cloud (name x tags) e usar terraform login.
- ☐ Sei os modos de execução remoto, local e com agentes.
- ☐ Sei usar variable sets e a regra de precedência com o workspace.
- ☐ Sei os níveis advisory, soft mandatory e hard mandatory.
- ☐ Sei para que servem run tasks, estimativa de custos e health assessments.
- ☐ Sei como um workspace lê outputs de outro.
- ☐ Sei o que são credenciais dinâmicas via OIDC.

Questões de revisão do capítulo

R5.1 · HCP Terraform

Como o HCP Terraform pode autenticar na nuvem sem guardar chaves de acesso de longa duração no workspace?

A. Chaves fixas escritas no código Terraform do repo
B. Tokens de usuário em tags do workspace
C. Credenciais dinâmicas de provider via OIDC
D. Senhas gravadas nos outputs do workspace

R5.2 · HCP Terraform

Como aplicar um mesmo conjunto de políticas a vários workspaces do HCP Terraform?

A. Com um output de cada workspace
B. Com uma variável policy em cada run
C. Com cópias das regras em cada código
D. Com um policy set vinculado a eles

R5.3 · HCP Terraform

Uma equipe deve ter acesso a todos os workspaces da aplicação de pagamentos, atuais e futuros. Qual é a abordagem mais simples?

A. Tornar o estado público na empresa
B. Dar acesso no projeto da aplicação
C. Compartilhar o token de um usuário
D. Dar acesso a cada workspace novo

R5.4 · HCP Terraform

Em um workspace do HCP Terraform, como disparar manualmente um novo plano pela interface web?

A. Com a troca do nome do workspace
B. Com a ação de iniciar um novo run
C. Com a exclusão do conjunto de variáveis
D. Com a edição do arquivo de estado

Respostas comentadas

R5.1. C) Credenciais dinâmicas de provider via OIDC

Por quê: A nuvem confia no HCP Terraform como provedor de identidade.

✗ A) Incorreta. Expõe credenciais.
✗ B) Incorreta. Expõe credenciais.
✗ D) Incorreta. Expõe credenciais.

> **Dica de prova:** Sem chaves longas: credenciais dinâmicas.

R5.2. D) Com um policy set vinculado a eles

Por quê: Policy sets podem valer para a organização, projetos ou workspaces.

✗ A) Incorreta. Não aplica políticas.
✗ B) Incorreta. Não aplica políticas.
✗ C) Incorreta. Difícil de manter.

> **Dica de prova:** Políticas em vários workspaces: policy sets.

R5.3. B) Dar acesso no projeto da aplicação

Por quê: Permissões de projeto simplificam a gestão em escala.

✗ A) Incorreta. Expõe dados sensíveis.
✗ C) Incorreta. Viola boas práticas.
✗ D) Incorreta. Exige manutenção manual.

> **Dica de prova:** Acesso a um grupo de workspaces: projeto.

R5.4. B) Com a ação de iniciar um novo run

Por quê: Também é possível escolher runs só de plano ou de refresh.

✗ A) Incorreta. Não dispara runs.

✗ C) Incorreta. Não dispara runs.

✗ D) Incorreta. Não dispara runs.

> **Dica de prova:** Run manual: iniciar novo run.

06

CAPÍTULO 6

Infraestrutura como Código com Terraform

Aqui eu preciso explicar por que IaC existe e o que o Terraform faz de diferente: descrever o estado final desejado em arquivos versionáveis e aplicar com um só fluxo em vários provedores. A prova cobra benefícios (consistência, repetibilidade, rastreabilidade, reuso, recuperação de desastre, custo com ambientes efêmeros) em cenários curtos de empresa. Também aparecem verdadeiro ou falso sobre multi-cloud e sobre a linguagem (HCL ou JSON). Onde mais erro é cair em alternativas exageradas do tipo 'dispensa testes', 'custo zero', 'dispensa credenciais' ou 'a IaC desfaz sozinha'. Regra prática: benefício de IaC nunca elimina revisão, testes, credenciais, backup de dados nem auditoria.

Neste capítulo

17 tópicos-chave · 12 pegadinhas · 6 macetes · 13 gatilhos · 4 questões comentadas

Tópicos-chave

Infraestrutura como Código (IaC)

Definir a infraestrutura em arquivos versionáveis, que podem ser revisados, versionados e reaplicados. É o oposto de criar recursos clicando no console ou documentar servidores em planilhas.

Linguagem declarativa

Descreve o estado final desejado, não os passos. O Terraform compara o desejado com o estado atual e monta um plano com as ações necessárias; a ordem vem das dependências, não da ordem dos blocos no arquivo.

Idempotência

Aplicar a mesma configuração duas vezes, sem mudança no código nem na infraestrutura, resulta em nenhuma alteração proposta na segunda vez. O estado reconhece os objetos, por isso não há erro de 'já existe' nem duplicatas.

Provider

Plugin que traduz a configuração em chamadas de API de cada plataforma: nuvem pública, nuvem privada, SaaS, DNS, monitoramento, GitHub, Kubernetes, Helm. É o que permite gerenciar serviços variados com a mesma ferramenta.

Multi-cloud e nuvem híbrida

Uma mesma configuração pode combinar vários providers (ex.: vSphere no datacenter e AWS). Referências entre providers diferentes funcionam e o Terraform ordena a criação pelo grafo, como um DNS apontando para o balanceador de outra nuvem.

Fluxo agnóstico ao serviço

Um só fluxo e uma só linguagem para qualquer serviço que tenha provider. Aprender o fluxo uma vez permite gerenciar plataformas muito diferentes; ferramentas de um provedor gerenciam só os serviços dele.

HCL e .tf.json

Os arquivos .tf usam HCL (HashiCorp Configuration Language). Existe sintaxe JSON equivalente em arquivos .tf.json, útil quando a configuração é gerada por outros programas. YAML, XML, CSV, INI, SQL ou Python não são suportados como linguagem nativa.

Foco do Terraform

Provisionar e gerenciar infraestrutura. Instalar pacotes dentro de servidores já criados é foco de gestão de configuração (Ansible, Chef); monitorar CPU e compilar código são outros domínios.

Infraestrutura imutável

Substituir recursos em vez de alterá-los no local. Reduz servidores únicos e difíceis de reproduzir; editar servidores em produção por acesso remoto gera desvio de configuração.

Paridade de ambientes

Homologação e produção usam o mesmo código; variáveis e tfvars por ambiente controlam só o que precisa mudar, como tamanho de máquinas.

Ambientes efêmeros e custo

IaC permite criar e destruir ambientes com facilidade, sob demanda. Reduz custo desligando o que não é usado; não muda o preço por hora do provedor.

Recuperação de desastre

Se uma região cai, o ambiente pode ser recriado a partir do código em outra região. A IaC recria a infraestrutura; dados continuam exigindo backup e replicação próprios.

Controle de mudanças

Mudanças passam por commits e revisão; o histórico do controle de versão mostra quem mudou o quê e quando. O código também funciona como documentação viva do ambiente real.

Compliance e testes antes da produção

Validação e plano rodam em pipelines de CI; ferramentas como Sentinel e OPA avaliam o plano contra políticas antes do apply. O estado não avalia regras nem risco.

Self-service com padrões

Uma plataforma interna oferece módulos aprovados para as equipes criarem seus recursos. Módulos e políticas garantem que o self-service siga as regras da empresa e aceleram novos projetos.

Níveis de provider no Registry

Official é mantido pela própria HashiCorp; Partner, por empresas parceiras validadas; Community, por qualquer pessoa da comunidade.

Desvio por mudança manual

Quando alguém altera algo no console, o código serve de referência para detectar e corrigir a diferença, ou para incorporá-la ao código. A IaC não bloqueia o console nem apaga recursos sem plano.

Macetes para memorizar

💡 Benefícios de IaC

CRRR: Consistência, Repetibilidade, Rastreabilidade, Reuso. Se a alternativa diz 'dispensa' ou 'custo zero', é errada.

💡 Declarativo

Diga O QUÊ, não COMO: o Terraform calcula o caminho pelo grafo.

💡 Níveis do Registry

O-P-C: Official = HashiCorp, Partner = parceira, Community = qualquer um.

💡 Foco da ferramenta

Terraform levanta a casa; Ansible/Chef mobília a casa.

💡 Desastre

Código recria a casa, backup devolve os móveis (dados).

💡 Linguagem

.tf é HCL; .tf.json é para máquina.

Tabela de memorização: não confunda

Quando é...	...e quando é
Terraform (provisionamento) Criar e gerenciar a infraestrutura: redes, instâncias, bancos, DNS, clusters.	**Ansible/Chef (gestão de configuração)** Instalar e configurar software dentro de máquinas já criadas.

Declarativo Você descreve o resultado final e o Terraform calcula os passos pelo grafo.	**Imperativo** Você lista cada chamada de API ou passo de script na ordem.
HCL (.tf) Configuração escrita por pessoas: mais legível.	**JSON (.tf.json)** Configuração gerada automaticamente por outros programas.
Provider Official Mantido pela própria HashiCorp.	**Provider Partner** Publicado por empresa parceira validada; Community é de qualquer pessoa.
Infraestrutura imutável Substituir o recurso por um novo a partir do código.	**Remendar no local** Editar servidor em produção por acesso remoto: gera desvio e servidores únicos.
IaC recria infraestrutura Região caiu: recriar o ambiente em outra região pelo código.	**Backup recupera dados** Discos e bancos precisam de backup e replicação próprios.

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que uma configuração só pode ter recursos de um provedor de nuvem.
Como não cair: Falso: é comum combinar nuvem, DNS, monitoramento, Kubernetes e Helm na mesma configuração.

⚠ **Alerta de prova:** Marcar 'executa os blocos na ordem do arquivo' como motivo de ser declarativo.
Como não cair: A ordem vem das dependências (grafo). Declarativo é descrever o estado final.

⚠ **Alerta de prova:** Escolher 'custo zero de infraestrutura' ou 'reduz o preço por hora' como benefício.
Como não cair: Os recursos continuam pagos. O ganho de custo vem de criar e destruir ambientes efêmeros com facilidade.

⚠ **Alerta de prova:** Aceitar 'dispensa testes', 'dispensa revisão' ou 'dispensa controle de versão'.
Como não cair: IaC reforça testes, revisão por pares e versionamento; nada disso deixa de existir.

⚠ **Alerta de prova:** Achar que a IaC dispensa credenciais nos provedores.
Como não cair: Credenciais continuam necessárias; o Terraform usa as APIs públicas dos provedores.

⚠ **Alerta de prova:** Achar que a IaC copia os dados dos discos para a nova região na recuperação de desastre.
Como não cair: A IaC recria a infraestrutura; dados exigem backup e replicação próprios.

⚠ **Alerta de prova:** Achar que 'a nuvem desfaz sozinha mudanças com erro' ou que o estado bloqueia mudanças perigosas.
Como não cair: Não há desfazer automático e o estado não avalia risco; quem verifica é CI com validate, plan e políticas.

⚠ **Alerta de prova:** Pensar que o Terraform precisa de agentes instalados nos servidores gerenciados.
Como não cair: O Terraform não exige agentes nos servidores: providers chamam as APIs. Não confunda com os agentes do HCP Terraform, que executam runs em rede privada.

⚠ **Alerta de prova:** Achar que um 'módulo único para todas as nuvens' substitui providers.
Como não cair: Módulos não substituem providers; cada API precisa do seu provider.

⚠ **Alerta de prova:** Marcar YAML como linguagem dos arquivos .tf.
Como não cair: A linguagem é HCL; a alternativa suportada é JSON em .tf.json.

⚠ **Alerta de prova:** Escolher 'uso de um estado único para a empresa' como benefício do self-service.
Como não cair: Estado único para a empresa não é recomendado; o benefício é provisionamento self-service com módulos aprovados.

⚠ **Alerta de prova:** Achar que o Terraform dá acesso a recursos que os provedores não expõem ou negocia preços.
Como não cair: O Terraform só usa as APIs públicas e não altera preços.

Como cai na prova

📝 **Como cai:** Cenário de empresa com uma dor (custo, paridade, desastre, auditoria) e quatro benefícios possíveis.
Como resolver: Identifique a dor e escolha o benefício que a resolve diretamente; descarte alternativas absolutas ('elimina', 'dispensa', 'custo zero').

📝 **Como cai:** Verdadeiro ou falso sobre multi-cloud ou um só provedor.
Como resolver: Lembre que vários providers na mesma configuração são permitidos e o fluxo é o mesmo.

📝 **Como cai:** Pergunta conceitual: 'por que a linguagem é declarativa' ou 'o que caracteriza infraestrutura imutável'.
Como resolver: Declarativo = estado final desejado; imutável = substituir em vez de alterar. Descarte ordem de arquivo e scripts shell.

📝 **Como cai:** Comparação do Terraform com ferramenta de um único provedor ou de gestão de configuração.
Como resolver: Vantagem do Terraform: um só fluxo e linguagem para vários provedores; foco em provisionamento.

📝 **Como cai:** Aplicar a mesma configuração de novo, o que acontece.
Como resolver: Sem mudanças no código nem na infraestrutura, nenhuma alteração é proposta; o estado evita duplicatas.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
ambientes iguais várias vezes por semana	**Repetibilidade com a mesma configuração**
homologação parecida com produção	**Mesmo código com variáveis/tfvars por ambiente**
ambientes usados só durante o dia, custo	**Criar e destruir ambientes efêmeros**
região indisponível	**Recriar o ambiente a partir do código (dados exigem backup)**
quem mudou o quê e quando	**Commits e revisão no controle de versão**
compliance, regras de segurança antes do apply	**Políticas sobre o código/plano (Sentinel, OPA) em CI**
equipes criam bancos com módulos aprovados	**Provisionamento self-service com padrões**
GitHub, monitoramento e nuvem com a mesma ferramenta	**Um provider para cada serviço com API**
vSphere no datacenter e AWS	**Providers de cada ambiente no mesmo código**
aplicar duas vezes sem mudanças	**Nenhuma alteração proposta (idempotência)**
configuração gerada por outro programa	**JSON em arquivos .tf.json**
substituir em vez de alterar	**Infraestrutura imutável**
regra de firewall alterada à mão	**O código serve de referência para detectar e corrigir o desvio**

🎯 **Se você lembrar de só uma coisa deste capítulo...**

- IaC não dispensa credenciais, testes, revisão, auditoria nem backup de dados.
- Vários providers na mesma configuração é permitido e comum (nuvem + DNS + monitoramento + Kubernetes + Helm).
- Official = mantido pela HashiCorp; Partner = empresa parceira validada; Community = comunidade.

Essenciais para a prova (checklist)

- ☐ Sei definir IaC e explicar por que a configuração do Terraform é declarativa.
- ☐ Sei associar cada benefício de IaC (consistência, repetibilidade, rastreabilidade, reuso, custo, desastre) a um cenário.
- ☐ Sei que o Terraform é idempotente: aplicar de novo sem mudanças não propõe nada.
- ☐ Sei que uma configuração pode combinar providers de nuvens, SaaS e datacenter local.
- ☐ Sei diferenciar provisionamento (Terraform) de gestão de configuração (Ansible, Chef).
- ☐ Sei que a linguagem é HCL e que existe .tf.json.
- ☐ Sei os três níveis de provider no Registry: Official, Partner e Community.
- ☐ Sei o que é infraestrutura imutável e por que remendar em produção gera desvio.

Questões de revisão do capítulo

R6.1 · Infraestrutura como Código com Terraform

Uma equipe cria ambientes de teste que só são usados durante o dia. Como a IaC ajuda a controlar custos?

A. Manter todos os ambientes ligados sem uso
B. Desligar a cobrança do provedor de nuvem
C. Reduzir o preço por hora dos recursos usados
D. Recriar e destruir ambientes com facilidade

R6.2 · Infraestrutura como Código com Terraform

Como a IaC ajuda equipes de compliance a verificar regras de segurança na infraestrutura?

A. Regras checadas no código antes do apply
B. O estado bloqueia recursos fora das regras
C. Os provedores passam a auditar tudo sozinhos
D. A auditoria deixa de ser necessária na empresa

R6.3 · Infraestrutura como Código com Terraform

Um registro DNS em um provedor precisa apontar para o balanceador criado em outra nuvem. Como o Terraform trata isso?

A. Exige duas configurações sem ligação alguma
B. Referências entre providers diferentes
C. Copia o IP manualmente para uma variável fixa
D. Proíbe recursos de providers distintos juntos

R6.4 · Infraestrutura como Código com Terraform

Como a IaC permite testar mudanças de infraestrutura antes de chegarem à produção?

A. O código dispensa ambientes de teste
B. Validação e plano rodam em pipelines de CI
C. A nuvem desfaz sozinha mudanças com erro
D. O estado bloqueia mudanças perigosas na nuvem

Respostas comentadas

R6.1. D) Recriar e destruir ambientes com facilidade

Por quê: Ambientes efêmeros são criados sob demanda e destruídos depois.

✗ A) Incorreta. Aumentaria o custo.
✗ B) Incorreta. Não é possível.
✗ C) Incorreta. O provedor define os preços.

> **Dica de prova:** Custo: ambientes efêmeros.

R6.2. A) Regras checadas no código antes do apply

Por quê: Ferramentas como Sentinel e OPA avaliam o plano contra políticas.

✗ B) Incorreta. O estado não avalia regras.
✗ C) Incorreta. Não é o caso.
✗ D) Incorreta. Continua necessária.

> **Dica de prova:** Compliance: políticas sobre o código.

R6.3. B) Referências entre providers diferentes

Por quê: O Terraform ordena a criação pelo grafo, mesmo entre providers diferentes.

✗ A) Incorreta. Podem estar juntas.
✗ C) Incorreta. Não é necessário.
✗ D) Incorreta. É permitido.

> **Dica de prova:** Multi-cloud: referências entre providers.

R6.4. B) Validação e plano rodam em pipelines de CI

Por quê: Políticas como código e testes automatizados reforçam essa verificação.

✗ A) Incorreta. Ambientes de teste seguem úteis.

✗ C) Incorreta. Não há desfazer automático.

✗ D) Incorreta. O estado não avalia risco.

> **Dica de prova:** Testar antes: CI com validate e plan.

07

CAPÍTULO 7

Manutenção da Infraestrutura

Aqui eu preciso saber importar recursos (bloco import declarativo e terraform import na CLI), mover e remover endereços (moved, state mv, removed, state rm), inspecionar o estado (state list, state show, show, output -json, graph) e depurar com logs (TF_LOG e variações). A prova pega muito na sintaxe e no que cada comando NÃO faz, como o terraform import da CLI não gerar código. Onde mais erro: ordem dos argumentos do import, níveis de log e diferença entre parar de gerenciar e destruir.

Neste capítulo

16 tópicos-chave · 13 pegadinhas · 5 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Bloco import

Disponível desde o Terraform 1.5, traz um recurso existente para o estado de forma declarativa e revisável no plano, com to e id. Desde o 1.6 o id aceita expressões (o valor precisa ser conhecido no plan). Com for_each no bloco import, importa várias instâncias de uma vez, uma por chave do mapa.

Gerar configuração do import

terraform plan -generate-config-out=gerado.tf gera o HCL dos recursos declarados em blocos import. Revise o código gerado antes de aplicar.

Import em módulo filho

O bloco import fica no módulo raiz, mesmo para recurso de módulo filho: to = module.rede.aws_vpc.principal.

terraform import (CLI)

Sintaxe: terraform import ENDEREÇO ID, como terraform import aws_instance.web i-0abc123. Só altera o estado: o bloco resource precisa existir antes e nenhum código é gerado.

Depois do import

O plan deve mostrar nenhuma mudança, indicando código fiel ao recurso real. Após o apply, o bloco import pode ser mantido ou removido; removê-lo não destrói nada.

Formato do ID de import

Consta na documentação do recurso no registry; pode ser identificador simples ou combinação de campos.

Bloco moved e state mv

Renomear aws_instance.app para aws_instance.api sem recriar: bloco moved (declarativo, fica no código e aparece no plano, preferível em equipe) ou terraform state mv (imperativo).

Bloco removed e state rm

Parar de gerenciar sem destruir: bloco removed com lifecycle destroy = false (desde o 1.7, declarativo) ou terraform state rm ENDEREÇO (imperativo). O recurso continua na nuvem.

state replace-provider

Atualiza o estado quando um provider muda de namespace no registry. Depois, ajuste required_providers e rode init.

Inspeção do estado

terraform state list lista endereços (aceita filtro como module.rede); state show ENDEREÇO mostra atributos de um recurso; terraform show exibe o estado inteiro ou um plano salvo (com -json para ferramentas).

terraform output e graph

output -json entrega outputs estruturados para scripts; -raw entrega um único valor sem aspas. terraform graph gera o grafo de dependências em DOT, convertível em imagem com Graphviz.

TF_LOG

Ativa logs detalhados. Níveis do mais para o menos detalhado: TRACE, DEBUG, INFO, WARN, ERROR. TF_LOG=JSON gera nível TRACE em JSON, formato que não é interface estável.

TF_LOG_PATH

Grava os logs em arquivo, mas só indica onde gravar: sem TF_LOG definido, nenhum arquivo é criado. Para desligar, remova TF_LOG do ambiente.

TF_LOG_CORE e TF_LOG_PROVIDER

Separam os logs do núcleo e dos providers; TF_LOG vale para ambos.

crash.log

Gravado quando o Terraform encerra com falha interna (panic); pode ser anexado a um relato de bug após remover dados sensíveis.

Cuidado com logs

Logs detalhados, como TRACE, podem conter dados sensíveis; revise antes de compartilhar e apague arquivos antigos. Logs não alteram estado nem recursos.

Macetes para memorizar

💡 import CLI

Endereço antes, ID depois: 'para onde' e depois 'quem'.

💡 Níveis de log

T-D-I-W-E: Tudo, Detalhes, Informação, Warning, Erro (do mais para o menos).

💡 Blocos declarativos

import traz, moved muda de lugar, removed solta (sem destruir com destroy = false).

💡 Versões dos blocos

Import 1.5, id com expressão 1.6, removed 1.7: escadinha 5-6-7.

💡 TF_LOG_PATH

PATH é o endereço, TF_LOG é o carteiro: sem carteiro, nada chega.

Tabela de memorização: não confunda

Quando é...	...e quando é
Bloco import Declarativo, revisável no plano, aceita for_each e pode gerar código com -generate-config-out.	**terraform import (CLI)** Imperativo, só estado; o bloco resource precisa existir antes.
moved / state mv Mudar o endereço sem recriar.	**removed / state rm** Tirar do controle do Terraform sem destruir.
state list Listar endereços no estado (com filtro opcional).	**state show ENDEREÇO** Ver todos os atributos de um recurso.
terraform show Estado inteiro ou plano salvo.	**terraform state show** Um único recurso do estado.

TF_LOG Define o nível (liga os logs).	**TF_LOG_PATH** Define o arquivo de destino (precisa de TF_LOG).
state rm Recurso continua existindo na nuvem.	**destroy -target** Recurso é destruído.

Números que caem

Item	Valor
Bloco import	**Desde o Terraform 1.5**
id do bloco import aceitando expressões	**Desde o Terraform 1.6**
Bloco removed	**Terraform 1.7**
Níveis de TF_LOG (mais para menos detalhado)	**TRACE, DEBUG, INFO, WARN, ERROR**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que terraform import da CLI cria o bloco resource em main.tf.
Como não cair: Não gera código; para gerar, use bloco import com plan -generate-config-out.

⚠ **Alerta de prova:** Inverter a ordem: terraform import i-0abc123 aws_instance.web.
Como não cair: Primeiro o endereço, depois o ID.

⚠ **Alerta de prova:** Usar bloco removed ao renomear recurso.
Como não cair: removed tira do estado; renomear é moved.

⚠ **Alerta de prova:** Achar que o bloco import precisa ficar para sempre ou virar removed.
Como não cair: Pode ser mantido ou removido; a remoção não destrói nada e removed tiraria o recurso do estado.

⚠ **Alerta de prova:** Definir TF_LOG_PATH e esperar arquivo de log.
Como não cair: Sem TF_LOG não há log; TF_LOG_PATH só indica o destino.

⚠ **Alerta de prova:** Usar TF_LOG=FILE, DATA, RAW ou YAML.
Como não cair: Valores válidos incluem os níveis (TRACE a ERROR) e JSON.

⚠ **Alerta de prova:** Achar que INFO ou WARN é mais detalhado que DEBUG.
Como não cair: Ordem: TRACE > DEBUG > INFO > WARN > ERROR.

⚠ **Alerta de prova:** Usar apply -replace ou destroy -target para parar de gerenciar.
Como não cair: -replace recria e -target destrói; parar sem destruir é state rm ou removed.

⚠ **Alerta de prova:** Usar count no bloco moved ou alias no import para importar em massa.
Como não cair: Importação em massa: for_each no bloco import.

⚠ **Alerta de prova:** Achar que o id do import só aceita texto literal.
Como não cair: Aceita expressões desde o 1.6, desde que conhecidas no plan.

⚠ **Alerta de prova:** Achar que plan com substituição ou destruição após import está ok.
Como não cair: Import bem feito = plano sem mudanças; ajuste o código até ficar limpo.

⚠ **Alerta de prova:** Usar terraform state mv-provider ou providers rename.
Como não cair: O comando é terraform state replace-provider.

⚠ **Alerta de prova:** Escolher terraform print, list, cat, map, tree ou deps.
Como não cair: Comandos reais: show, state list, graph.

Como cai na prova

📝 **Como cai:** Cenário de refatoração: renomear, mover para módulo, parar de gerenciar.
Como resolver: Mudar endereço = moved/state mv; parar de gerenciar = removed/state rm; nunca destroy ou -replace.

📝 **Como cai:** Importar recurso existente.
Como resolver: Declarativo: bloco import (to, id, for_each, -generate-config-out). CLI: terraform import ENDEREÇO ID com o resource já escrito. Valide com plan sem mudanças.

📝 **Como cai:** Depuração com variáveis de ambiente.
Como resolver: TF_LOG liga e define nível; TF_LOG_PATH define arquivo; _CORE e _PROVIDER separam; desligar é remover TF_LOG.

📝 **Como cai:** Qual comando mostra determinada informação.
Como resolver: Lista de endereços: state list; um recurso: state show; tudo ou plano: show; outputs: output -json; grafo: graph.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
recurso renomeado no código sem recriar	**Bloco moved (ou state mv)**
deixar de gerenciar sem destruir, revisável no plano	**removed com destroy = false**
deixar de gerenciar sem destruir pela CLI	**terraform state rm ENDEREÇO**
trazer recurso existente de forma declarativa	**Bloco import com to e id**
gerar HCL do recurso importado	**plan -generate-config-out=arquivo.tf**
importar várias instâncias de um mapa	**for_each no bloco import**
provider mudou de namespace	**terraform state replace-provider**
logs em arquivo	**TF_LOG + TF_LOG_PATH**
logs só dos providers	**TF_LOG_PROVIDER**
panic	**crash.log**
outputs para script	**terraform output -json (-raw para um valor)**
grafo em DOT	**terraform graph**
atributos de um único recurso	**terraform state show ENDEREÇO**
formato de ID para importar	**Documentação do recurso no registry**

🎯 **Se você lembrar de só uma coisa deste capítulo...**

- terraform import da CLI não gera código: o resource precisa existir antes.
- Renomear = moved; parar de gerenciar sem destruir = removed (destroy = false) ou state rm.
- TF_LOG_PATH sem TF_LOG não grava nada.
- Import bem feito termina com plan sem mudanças.

Essenciais para a prova (checklist)

- ☐ Sei escrever um bloco import com to e id, inclusive para módulo filho e com for_each.

- ☐ Sei gerar configuração com plan -generate-config-out.
- ☐ Sei a sintaxe de terraform import ENDEREÇO ID e sua limitação.
- ☐ Sei diferenciar moved, state mv, removed e state rm.
- ☐ Sei usar state list, state show, show, output -json e graph.
- ☐ Sei os níveis de TF_LOG e as variáveis TF_LOG_PATH, TF_LOG_CORE e TF_LOG_PROVIDER.
- ☐ Sei o que é crash.log e por que revisar logs antes de compartilhar.
- ☐ Sei quando usar terraform state replace-provider.

Questões de revisão do capítulo

R7.1 · Manutenção da Infraestrutura

Os logs de TF_LOG estão poluindo a saída do pipeline. Como gravá-los em um arquivo?

A. Adicionar um bloco logging no código
B. Definir TF_LOG_PATH com o caminho
C. Usar a opção -log-file no plan
D. Definir TF_LOG=FILE no ambiente

R7.2 · Manutenção da Infraestrutura

No bloco import, o argumento id pode vir de uma variável em vez de um texto fixo?

A. Sim, mas só com dados do estado
B. Não, o id vem sempre do provider
C. Não, aceita só texto literal fixo
D. Sim, aceita expressões desde o 1.6

R7.3 · Manutenção da Infraestrutura

Qual valor de TF_LOG gera logs no nível TRACE em formato JSON, para ferramentas de análise?

A. TF_LOG=DATA
B. TF_LOG=JSON
C. TF_LOG=RAW
D. TF_LOG=YAML

R7.4 · Manutenção da Infraestrutura

Qual comando renomeia um endereço no estado de forma imperativa, como alternativa ao bloco moved?

A. terraform state pull
B. terraform state mv
C. terraform state rm
D. terraform state list

Respostas comentadas

R7.1. B) Definir TF_LOG_PATH com o caminho

Por quê: TF_LOG_CORE e TF_LOG_PROVIDER separam logs do núcleo e dos providers.

✗ A) Incorreta. Bloco inexistente.
✗ C) Incorreta. Opção inexistente.
✗ D) Incorreta. Nível inexistente.

> **Dica de prova:** Logs em arquivo: TF_LOG_PATH.

R7.2. D) Sim, aceita expressões desde o 1.6

Por quê: O valor precisa ser conhecido durante o plan.

✗ A) Incorreta. Aceita variáveis.
✗ B) Incorreta. É informado no bloco.
✗ C) Incorreta. Aceita expressões.

> **Dica de prova:** import id: aceita expressões.

R7.3. B) TF_LOG=JSON

Por quê: O formato JSON pode mudar entre versões e não é uma interface estável.

✗ A) Incorreta. Valor inválido.
✗ C) Incorreta. Valor inválido.
✗ D) Incorreta. Valor inválido.

> **Dica de prova:** Logs em JSON: TF_LOG=JSON.

R7.4. B) terraform state mv

Por quê: O bloco moved é preferível em equipe, pois fica no código e aparece no plano.

✗ A) Incorreta. Baixa o estado.
✗ C) Incorreta. Remove do estado.
✗ D) Incorreta. Lista endereços.

> **Dica de prova:** Renomear no estado pela CLI: state mv.

08

CAPÍTULO 8

Módulos do Terraform

Preciso saber como módulos se comunicam (variáveis entram, outputs saem, irmãos se ligam pelo raiz), as origens possíveis (caminho local, registry público, registry privado, Git com ?ref= e //subdiretório) e o versionamento (version só para registry, ~>, versão exata em produção, init -upgrade). A prova cobra também meta-argumentos no bloco module (count, for_each, depends_on, providers) e o bloco moved ao refatorar para módulo. Onde mais erro: achar que o filho enxerga var do pai, usar version em Git ou local, e achar que o lock guarda versão de módulo.

Neste capítulo

17 tópicos-chave · 13 pegadinhas · 6 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Módulo raiz x filho

Raiz são os arquivos do diretório de trabalho; todo módulo chamado a partir dele é filho. O nome não tem relação com quantidade de recursos ou origem.

Interface entre módulos

Pai para filho: argumentos no bloco module (variáveis de entrada). Filho para pai: blocos output, lidos como module.NOME.OUTPUT. Variáveis, locals e recursos do filho são encapsulados.

Escopo de variáveis

Cada módulo tem seu escopo; não existem variáveis globais nem herança. Raiz e filho podem ter variáveis com o mesmo nome sem conflito.

Módulos irmãos

Irmãos não se enxergam: o raiz passa module.rede.subnet_id como argumento ao módulo app.

Variável sem default

Se o bloco module não informar o argumento, o Terraform acusa argumento ausente. Defina default para tornar a entrada opcional.

Estrutura convencional

main.tf, variables.tf e outputs.tf, com README de exemplos esperado no registry. Boa prática: expor só variáveis necessárias e documentadas, com padrões sensatos.

Origens (source)

Caminho local (./modulos/rede, ../compartilhado/rede), registry público (namespace/nome/provider, como hashicorp/consul/aws), registry privado (app.terraform.io/ORG/NOME/PROVIDER) e Git. source não aceita variáveis.

Git: ?ref= e //

?ref=v1.2.0 aponta uma tag ou referência; //modulos/vpc indica subdiretório do repositório. Podem ser combinados.

Argumento version

Fixa versão só para módulos de registry. Não vale para Git (use ?ref=) nem para caminho local, que não tem versões publicadas.

Versionamento semântico

Major nova (2.4.1 para 3.0.0) pode ter mudanças incompatíveis; minor traz recursos compatíveis; patch, correções. ~> 1.2.0 aceita de 1.2.0 até antes de 1.3.0; ~> 2.0 aceita até antes de 3.0.

Versão em produção

Fixar a versão evita mudanças inesperadas ao rodar init. Para atualizações sempre deliberadas, use versão exata (= 3.4.1) e atualize via PR revisando o plano.

Download de módulos

terraform init baixa módulos para .terraform/modules; terraform get baixa ou atualiza só módulos. Rode init ao adicionar bloco module ou mudar source; init -upgrade atualiza para versão nova dentro da restrição. Não edite .terraform.

Lock e módulos

.terraform.lock.hcl registra somente providers. Versões de módulos são fixadas no próprio bloco module.

Meta-argumentos no bloco module

count (condicional 0/1, acesso module.bastion[0]), for_each (uma instância por chave, module.nome["chave"]), depends_on (ordem sem referência) e providers (repassa configuração com alias).

Mesmo módulo duas vezes

Sem count nem for_each, declare dois blocos module distintos, cada um com nome único, argumentos e endereço próprios.

moved ao refatorar

Mover recurso do raiz para dentro de um módulo sem recriar: bloco moved com from = aws_vpc.principal e to = module.rede.aws_vpc.principal.

Registry e selos

Terraform Registry reúne módulos prontos e documentados; repositório público deve se chamar terraform-PROVIDER-NOME e as tags semânticas viram versões. Partner indica publicação por empresa parceira validada. Registry privado exige terraform login.

Macetes para memorizar

💡 Comunicação

Variável é a porta de entrada, output é a porta de saída, o raiz é o corredor entre irmãos.

💡 Versão por origem

Registry usa version, Git usa ?ref=, local não usa nada.

💡 Origem Git

Duas barras descem a pasta (//), ?ref escolhe a versão.

💡 SemVer

MAJOR quebra, MINOR soma, PATCH conserta.

💡 Lock

Lock é de provider; módulo se tranca no próprio bloco.

💡 Nome no registry

terraform-PROVIDER-NOME: começa com terraform, termina com o seu nome.

Tabela de memorização: não confunda

Quando é...	...e quando é

version Módulo do registry (público ou privado).	**?ref=** Módulo de repositório Git: ?ref= no source aponta a tag ou outra referência (ex.: ?ref=v1.2.0).
variável no bloco module Levar valor do pai para o filho.	**output do filho** Levar valor do filho para o pai (module.NOME.OUTPUT).
terraform get Baixar/atualizar só módulos.	**terraform init** Passo usual: módulos, providers e backend.
~> 2.0 De 2.0 até antes de 3.0.	**~> 2.0.0** Só 2.0.x.
versão exata (= 3.4.1) Produção com atualização sempre deliberada.	**~> 3.0 / >= 3.0** Aceitam versões novas automaticamente.
moved Mudar endereço (inclusive para dentro de módulo) sem recriar.	**removed** Tirar o recurso do estado.
for_each no module Várias instâncias do módulo a partir de um mapa.	**dynamic** Blocos aninhados repetidos dentro de um recurso.

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que o filho pode ler var.regiao do raiz diretamente.
Como não cair: O filho não enxerga variáveis do pai; passe o valor como argumento no bloco module.

⚠ **Alerta de prova:** Achar que o app lê module.rede.subnet_id direto do irmão.
Como não cair: Irmãos não se enxergam; o raiz conecta os dois.

⚠ **Alerta de prova:** Achar que o raiz lê local.nome_final do filho.
Como não cair: Locals do filho são privados; o filho precisa expor em output.

⚠ **Alerta de prova:** Usar version = "v1.2.0" em módulo Git.
Como não cair: version não vale para Git; use ?ref=v1.2.0 no source.

⚠ **Alerta de prova:** Achar que o lock registra versões de módulos ou da CLI.
Como não cair: Lock: só providers.

⚠ **Alerta de prova:** Achar que ~> 1.2.0 vai até antes de 2.0.0.
Como não cair: Isso seria ~> 1.2; ~> 1.2.0 é só patches.

⚠ **Alerta de prova:** Achar que >= 2.0 exclui a 3.0.
Como não cair: >= não tem teto; para parar antes da 3.0 use ~> 2.0.

⚠ **Alerta de prova:** Achar que a variável sem default vira null ou herda do raiz.
Como não cair: Sem default é obrigatória: o Terraform acusa argumento ausente.

⚠ **Alerta de prova:** Controlar a criação do módulo com source = var.criar ou create_module.
Como não cair: source não aceita variáveis; use count = var.criar ? 1 : 0.

⚠ **Alerta de prova:** Achar que // na origem indica versão ou backend.
Como não cair: // indica subdiretório; versão é ?ref= ou version.

⚠ **Alerta de prova:** Usar bloco removed ou import ao mover recurso para dentro de módulo.
Como não cair: removed tira do estado e import traz objeto externo; o certo é moved.

⚠ **Alerta de prova:** Repassar provider com alias usando alias no bloco module.
Como não cair: alias não existe no bloco module; use o argumento providers.

⚠ **Alerta de prova:** Escolher terraform fetch, sync ou pull para baixar módulos.
Como não cair: O comando é terraform get (na prática, o init já baixa).

Como cai na prova

📝 **Como cai:** Como um valor chega de um módulo a outro (pai, filho, irmão).
Como resolver: Lembre que não há globais: entrada por argumento, saída por output, irmãos via raiz. Descarte var., data.module e local. do outro módulo.

📝 **Como cai:** Qual source corresponde a qual origem.
Como resolver: ./ ou ../ é local; namespace/nome/provider é registry; app.terraform.io/... é privado; URL com ?ref= é Git.

📝 **Como cai:** Qual restrição de versão atende um requisito.
Como resolver: Até antes da próxima major: ~> X.Y; só patches: ~> X.Y.Z; controle total: versão exata.

📝 **Como cai:** Qual comando rodar depois de mudar um módulo.
Como resolver: Bloco novo ou source novo: init; versão nova compatível: init -upgrade; só módulos: get.

📝 **Como cai:** Meta-argumento no bloco module para um cenário.
Como resolver: Condicional: count; várias por mapa: for_each; ordem: depends_on; alias: providers.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
valor do filho para o pai	**output no filho + module.NOME.OUTPUT**
valor do pai para o filho	**Argumento no bloco module**
módulos irmãos rede e app	**O raiz passa module.rede.X ao app**
módulo de tag Git	**?ref=v1.2.0 no source**
subdiretório do repositório	**//caminho no source**
registry privado do HCP Terraform	**app.terraform.io/ORG/NOME/PROVIDER + terraform login**
nome de repositório para o registry público	**terraform-PROVIDER-NOME**
módulo só quando a flag for verdadeira	**count = var.flag ? 1 : 0**
uma instância de módulo por ambiente a partir de mapa	**for_each no bloco module**
ordem entre módulos sem referências	**depends_on no bloco module**
novo bloco module ou source alterado	**terraform init antes do plan**
nova versão compatível do módulo	**terraform init -upgrade**
recurso movido para dentro de module.rede	**Bloco moved**
onde ficam módulos baixados	**.terraform/modules**

🎯 **Se você lembrar de só uma coisa deste capítulo...**

- Não existem variáveis globais: filho não lê var do pai, pai não lê local do filho, irmãos se ligam pelo raiz.
- version só para registry; Git usa ?ref=; caminho local não tem versão.
- .terraform.lock.hcl guarda só providers.
- Mudou source ou adicionou module: terraform init antes do plan.

Essenciais para a prova (checklist)

- ☐ Sei a diferença entre módulo raiz e filho.
- ☐ Sei passar valores do pai para o filho e ler outputs com module.NOME.OUTPUT.
- ☐ Sei conectar módulos irmãos pelo raiz.
- ☐ Sei reconhecer cada formato de source (local, registry, privado, Git com ?ref= e //).
- ☐ Sei ler restrições ~> X.Y e ~> X.Y.Z e quando usar versão exata.
- ☐ Sei que o lock não guarda versões de módulos.
- ☐ Sei usar count, for_each, depends_on e providers no bloco module.
- ☐ Sei usar moved ao refatorar recurso para dentro de um módulo.
- ☐ Sei o padrão de nome terraform-PROVIDER-NOME e onde ficam os módulos baixados.

Questões de revisão do capítulo

R8.1 · Módulos do Terraform

Como fixar a versão de um módulo do Terraform Registry no bloco module?

A. Com o argumento version
B. Com o bloco required_providers
C. Com o argumento release
D. Com um sufixo no nome local

R8.2 · Módulos do Terraform

Qual estrutura de arquivos é convencional em um módulo reutilizável?

A. module.json, vars.json e out.json
B. main.tf, variables.tf e outputs.tf
C. init.tf, plan.tf, apply.tf e run.tf
D. state.tf, lock.tf e backend.tf

R8.3 · Módulos do Terraform

Qual faixa de versões a restrição ~> 1.2.0 permite para um módulo?

A. De 1.2.0 até antes de 1.3.0
B. De 1.2.0 em diante, sem teto
C. De 1.2.0 até antes de 2.0.0
D. Exatamente a versão 1.2.0

R8.4 · Módulos do Terraform

Como criar duas VPCs diferentes usando o mesmo módulo, sem count nem for_each?

A. Declarar dois blocos module distintos
B. Declarar duas versões no mesmo bloco module
C. Declarar um bloco module com dois sources
D. Declarar o módulo duas vezes com o mesmo nome

Respostas comentadas

R8.1. A) Com o argumento version

Por quê: O argumento version só vale para módulos de registry; para Git, use ?ref= na origem.

✗ B) Incorreta. É para providers.
✗ C) Incorreta. Não existe no bloco module.
✗ D) Incorreta. O nome local não fixa versão.

> **Dica de prova:** Versão de módulo do registry: version.

R8.2. B) main.tf, variables.tf e outputs.tf

Por quê: Um README com exemplos de uso também é esperado no registry.

✗ A) Incorreta. Não é a convenção.
✗ C) Incorreta. Não é a convenção.
✗ D) Incorreta. Não é a convenção.

> **Dica de prova:** Módulo padrão: main, variables, outputs.

R8.3. A) De 1.2.0 até antes de 1.3.0

Por quê: O operador ~> permite variar só o componente mais à direita informado.

✗ B) Incorreta. Há limite superior.
✗ C) Incorreta. Seria ~> 1.2.
✗ D) Incorreta. Seria = 1.2.0.

> **Dica de prova:** ~> 1.2.0: só patches.

R8.4. A) Declarar dois blocos module distintos

Por quê: Cada bloco recebe seus próprios argumentos e tem endereço próprio.

✗ B) Incorreta. Não cria instâncias.
✗ C) Incorreta. Cada bloco tem um source.
✗ D) Incorreta. Nomes devem ser únicos.

> **Dica de prova:** Mesmo módulo duas vezes: dois blocos.

DIA DA PROVA

Estratégia de prova

A prova HashiCorp Certified: Terraform Associate (004) tem cerca de 50 a 60 questões em 60 minutos, baseadas no Terraform 1.12, misturando verdadeiro ou falso, múltipla escolha, múltipla resposta e preenchimento. Os enunciados são curtos e práticos: um cenário de equipe ou pipeline e a pergunta 'qual comando, opção, bloco ou argumento usar'. Muitas alternativas erradas são opções ou comandos inventados com nome plausível (-yes, -force, terraform fetch, hidden = true), e outras são recursos reais usados no lugar errado. A prova também valoriza as práticas recomendadas atuais: blocos declarativos (import, moved, removed), -replace no lugar de taint, -refresh-only no lugar de refresh, ephemeral e write-only para segredos. Pelos pesos de referência (a HashiCorp não publica pesos oficiais), Configuração (22,9%) e Fluxo de Trabalho Principal (17,5%) são os domínios com mais questões, somando cerca de 40% da prova.

Como responder

✓ Com cerca de um minuto por questão, leia primeiro a pergunta final e marque mentalmente o verbo: criar, ler, mover, remover, destruir, validar, formatar.

✓ Em questões de comando, elimine primeiro opções que não existem; normalmente sobram duas e o detalhe do cenário (sem prompt, sem migrar, sem alterar arquivos) decide.

✓ Em múltipla resposta (Escolha DUAS), avalie cada alternativa isoladamente como verdadeiro ou falso antes de escolher o par.

✓ Em trechos de código, confira prefixo (var., local., data., module.), ordem tipo.nome.atributo e indexação ([0] para count, ["chave"] para for_each).

✓ Em restrições de versão, escreva a faixa: ~> X.Y vai até antes de X+1; ~> X.Y.Z vai até antes de X.Y+1.

✓ Quando duas alternativas parecem certas, prefira a declarativa e atual (moved, removed, bloco import, -replace, apply -refresh-only) à imperativa ou obsoleta.

✓ Em verdadeiro ou falso, desconfie de absolutos e lembre os fatos fixos: validate precisa de init, apply de plano salvo não pede confirmação, fmt por padrão reescreve.

✓ Marque as dúvidas e volte no fim; não gaste mais de dois minutos em uma questão de sintaxe.

Erros que derrubam candidatos

✗ Achar que sensitive = true tira o segredo do estado.

✗ Errar a precedência de variáveis (TF_VAR_ perde para tfvars; -var vence tudo).

✗ Confundir validate (sintaxe e referências) com fmt -check (estilo) e com plan (comparação com o mundo real).

✗ Usar version em módulo Git ou local e achar que o lock guarda versões de módulos.

✗ Achar que terraform import da CLI gera código ou inverter endereço e ID.

✗ Confundir state rm/removed (parar de gerenciar) com destroy -target (destruir).

✗ Confundir agentes com run tasks e soft com hard mandatory no HCP Terraform.

Na semana da prova

- ☐ Refaça as provas práticas e anote cada opção de comando que errou em uma lista única de flags (init, plan, apply, fmt, state).
- ☐ Revise as tabelas de comparação: count x for_each, precondition x postcondition x check, moved x removed, plan x apply -refresh-only.
- ☐ Escreva de memória a ordem de precedência de variáveis e os argumentos do lifecycle.
- ☐ Treine restrições de versão (~>, >=, =) até ler a faixa sem pensar.
- ☐ Revise as versões de recursos novos: import 1.5, removed 1.7, provider functions 1.8, validation cruzada 1.9, ephemeral 1.10, write-only 1.11.
- ☐ Na véspera, só checklists e o item 'lembrar' de cada domínio; durma bem, a prova tem só 60 minutos.

REVISÃO FINAL

Folha de cola

Leia na véspera e na manhã da prova: os gatilhos e as pegadinhas mais importantes de cada domínio.

1. Configuração do Terraform

‣ valor com -var, tfvars e default ao mesmo tempo → **-var vence**

‣ a.auto.tfvars e b.auto.tfvars com a mesma variável → **b vence (ordem léxica)**

‣ aceitar só dev, hml ou prd com mensagem → **validation com contains e error_message**

‣ falhar se não recebeu IP público após criar → **postcondition no lifecycle**

‣ conferir imagem antes de criar → **precondition no lifecycle**

⚠ Crescente: default < TF_VAR_ < terraform.tfvars < *.auto.tfvars < -var/-var-file.

⚠ sensitive só oculta na saída; para tirar do estado use ephemeral ou write-only.

⚠ Referências válidas: local.prefixo e var.regiao.

2. Fluxo de Trabalho Principal

‣ pipeline sem digitar yes → **-auto-approve**

‣ pipeline falhar em vez de esperar variável → **-input=false**

‣ código de saída 2 quando há mudanças → **plan -detailed-exitcode**

‣ pipeline falhar se .tf fora do padrão sem alterar → **terraform fmt -check (com -recursive)**

‣ aplicar exatamente o que foi revisado → **plan -out=tfplan e apply tfplan**

⚠ validate não acessa a nuvem; credenciais inválidas aparecem no plan/apply e comparar com a infraestrutura real é papel do plan.

⚠ Verdadeiro: sem init não há providers para validar; em CI use init -backend=false.

⚠ Falso: o plano salvo é aplicado sem prompt, por isso revise antes.

3. Fundamentos do Terraform

‣ origem e versão de provider, hashicorp/aws ~> 5.0 → **required_providers dentro de terraform**

‣ região padrão do provider → **Bloco provider**

‣ duas regiões da AWS → **Segundo bloco provider com alias + provider = aws.ALIAS**

‣ versões exatas registradas pelo init → **.terraform.lock.hcl (versionar)**

‣ onde ficam os plugins baixados → **Subdiretório .terraform (não versionar)**

⚠ ~> trava o componente à esquerda do último informado: ~> 5.1 fica abaixo de 6.0.

⚠ O lock não contém senhas, não é recriado pelo plan e deve ser versionado.

⚠ Falso: o estado pode conter segredos e causar conflitos; use backend remoto com bloqueio e criptografia.

4. Gerenciamento de State

‣ estado bloqueado após execução interrompida → **terraform force-unlock ID_DO_LOCK**

‣ segunda pessoa esperar até 5 minutos → **-lock-timeout=5m**

‣ trocar backend local por remoto levando o estado → **terraform init -migrate-state**

‣ Objects have changed outside of Terraform → **Drift: mudança fora do Terraform**

‣ atualizar estado sem mudar recursos → **plan -refresh-only e apply -refresh-only**

⚠ O backend é lido antes das variáveis; use configuração parcial com -backend-config.

⚠ Tem bloqueio do sistema de arquivos, que só vale no disco local.

⚠ O estado continua podendo conter segredos; protegê-lo é papel de criptografia e acesso.

5. HCP Terraform

‣ outra ferramenta envia configuração e dispara runs → **Fluxo orientado por API**

‣ commits e pull requests disparam runs → **Fluxo VCS**

‣ sem chaves de longa duração no workspace → **Credenciais dinâmicas via OIDC**

‣ mesmas políticas em vários workspaces → **Policy set**

‣ acesso a workspaces atuais e futuros da aplicação → **Permissão no projeto**

⚠ O workspace vence por padrão; só o conjunto prioritário sobrepõe.

⚠ Agentes executam runs em redes privadas; integração com terceiros é run task.

⚠ Só soft mandatory permite que usuário autorizado prossiga.

6. Infraestrutura como Código com Terraform

‣ ambientes iguais várias vezes por semana → **Repetibilidade com a mesma configuração**

‣ homologação parecida com produção → **Mesmo código com variáveis/tfvars por ambiente**

‣ ambientes usados só durante o dia, custo → **Criar e destruir ambientes efêmeros**

‣ região indisponível → **Recriar o ambiente a partir do código (dados exigem backup)**

‣ quem mudou o quê e quando → **Commits e revisão no controle de versão**

⚠ Falso: é comum combinar nuvem, DNS, monitoramento, Kubernetes e Helm na mesma configuração.

⚠ A ordem vem das dependências (grafo). Declarativo é descrever o estado final.

⚠ Os recursos continuam pagos. O ganho de custo vem de criar e destruir ambientes efêmeros com facilidade.

7. Manutenção da Infraestrutura

‣ recurso renomeado no código sem recriar → **Bloco moved (ou state mv)**

‣ deixar de gerenciar sem destruir, revisável no plano → **removed com destroy = false**

‣ deixar de gerenciar sem destruir pela CLI → **terraform state rm ENDEREÇO**

‣ trazer recurso existente de forma declarativa → **Bloco import com to e id**

‣ gerar HCL do recurso importado → **plan -generate-config-out=arquivo.tf**

⚠ Não gera código; para gerar, use bloco import com plan -generate-config-out.

⚠ Primeiro o endereço, depois o ID.

⚠ removed tira do estado; renomear é moved.

8. Módulos do Terraform

‣ valor do filho para o pai → **output no filho + module.NOME.OUTPUT**

‣ valor do pai para o filho → **Argumento no bloco module**

‣ módulos irmãos rede e app → **O raiz passa module.rede.X ao app**

‣ módulo de tag Git → **?ref=v1.2.0 no source**

‣ subdiretório do repositório → **//caminho no source**

⚠ O filho não enxerga variáveis do pai; passe o valor como argumento no bloco module.

⚠ Irmãos não se enxergam; o raiz conecta os dois.

⚠ Locals do filho são privados; o filho precisa expor em output.

GABARITO

Respostas do diagnóstico

D1. C) flatten → revise o capítulo 1

Por quê: flatten é muito usado para preparar dados para for_each.

✗ A) Incorreta. Junta listas informadas.
✗ B) Incorreta. Remove repetidos.
✗ D) Incorreta. Remove vazios.

> **Dica de prova:** Listas aninhadas: flatten.

D2. C) As instâncias seguintes mudam de índice → revise o capítulo 1

Por quê: Com for_each e chaves estáveis, só a instância removida é afetada.

✗ A) Incorreta. Não há bloqueio.
✗ B) Incorreta. Não é o comportamento.
✗ D) Incorreta. Ele seria destruído.

> **Dica de prova:** count com lista mutável: prefira for_each.

D3. B) Sim, desde o Terraform 1.9 → revise o capítulo 1

Por quê: Antes do 1.9, a condição só podia referenciar a própria variável.

✗ A) Incorreta. Não é o caso.
✗ C) Incorreta. Isso era a regra antiga.
✗ D) Incorreta. Não é o caso.

> **Dica de prova:** validation cruzada: Terraform 1.9+.

D4. B) Mostra o impacto antes de mudar algo → revise o capítulo 2

Por quê: Destruições e substituições inesperadas aparecem com destaque no plano.

✗ A) Incorreta. O estado segue necessário.
✗ C) Incorreta. O plan não aplica.
✗ D) Incorreta. Isso é init -upgrade.

> **Dica de prova:** PR com plan: impacto visível.

D5. B) Um novo plano é gerado e exibido → revise o capítulo 2

Por quê: Em pipelines, salvar o plano garante que o aplicado é o que foi revisado.

✗ A) Incorreta. Não é o comportamento.
✗ C) Incorreta. Sem arquivo, gera outro.
✗ D) Incorreta. O plano salvo é opcional.

> **Dica de prova:** apply sem arquivo: gera plano na hora.

D6. D) Retorna código de saída não zero → revise o capítulo 2

Por quê: Os arquivos com problema são listados na saída.

✗ A) Incorreta. Não abre editores.

✗ B) Incorreta. Com -check não reescreve.

✗ C) Incorreta. Não apaga arquivos.

> **Dica de prova:** fmt -check: falha sem reescrever.

D7. A) version = "5.31.0" → revise o capítulo 3

Por quê: O operador = explícito tem o mesmo efeito que não usar operador.

✗ B) Incorreta. Exclui justamente essa versão.

✗ C) Incorreta. Permite versões 5.x maiores.

✗ D) Incorreta. Permite qualquer maior.

> **Dica de prova:** Versão exata: sem operador ou com =.

D8. C) Do namespace hashicorp no registry público → revise o capítulo 3

Por quê: Declare sempre source e version para evitar ambiguidade, sobretudo com providers de outros namespaces.

✗ A) Incorreta. Espelhos exigem configuração.

✗ B) Incorreta. Estado não fornece providers.

✗ D) Incorreta. Módulos não contêm providers.

> **Dica de prova:** Provider não declarado: hashicorp/NOME.

D9. C) required_providers dentro de terraform → revise o capítulo 3

Por quê: O bloco provider configura o provider; required_providers diz de onde baixá-lo e qual versão aceitar.

✗ A) Incorreta. Não declara origem nem versão.

✗ B) Incorreta. Variáveis não instalam plugins.

✗ D) Incorreta. Outputs expõem valores.

> **Dica de prova:** Origem e versão de provider: required_providers.

D10. B) Não o vê mais nem o gerencia → revise o capítulo 4

Por quê: Recursos órfãos continuam gerando custo e devem ser tratados manualmente ou importados.

✗ A) Incorreta. Não está no estado.

✗ C) Incorreta. Não está no código.

✗ D) Incorreta. Não há import automático.

> **Dica de prova:** Fora do código e do estado: órfão.

D11. C) Impedir execuções simultâneas no mesmo estado → revise o capítulo 4

Por quê: Se o bloqueio ficar preso por falha, terraform force-unlock pode liberá-lo com cuidado.

✗ A) Incorreta. É outro controle.

✗ B) Incorreta. Versionamento é outro recurso.

✗ D) Incorreta. Controle de acesso é outro recurso.

> **Dica de prova:** Locking: uma operação de escrita por vez.

D12. D) terraform apply -refresh-only → revise o capítulo 4

Por quê: O antigo terraform refresh faz o mesmo sem revisão e não é recomendado.

✗ A) Incorreta. Opção inexistente.

✗ B) Incorreta. Opção inexistente.

✗ C) Incorreta. Opção inexistente.

> **Dica de prova:** Aceitar desvio no estado: apply -refresh-only.

Agora, treine no formato real

Faça as 6 provas completas do curso Terraform Associate: 342 questões inéditas, tempo do exame, explicação de cada alternativa e link da documentação para revisar.

Ver as 6 provas completas e o cupom do mês →

Até o dia da prova, uma revisão por dia no seu feed

Instagram @strecht.simulados

Um post de prova por dia: questão comentada, pegadinhas e o "não confunda" de cada certificação.

Seguir no Instagram →

YouTube Strecht Simulados

Shorts diários com pegadinhas de prova e, aos sábados, um simulado comentado.

Inscrever-se no canal →

Material independente de preparação, sem afiliação ou endosso de HashiCorp. As questões são inéditas e não reproduzem a prova real.