HASHICORP TERRAFORM ASSOCIATE PRACTICE EXAM (SAMPLE)

STUDY GUIDE

SAMPLE STUDY GUIDE WITH LIMITED QUESTIONS

VISIT US ONLINE FOR THE FULL VERSION STUDY GUIDE

https://hashicorpterraform.examzify.com

Table of Contents

Introduction

Preparing for a certification exam can feel overwhelming, but with the right tools, it becomes an opportunity to build confidence, sharpen your skills, and move one step closer to your goals. At Examzify, we believe that effective exam preparation isn't just about memorization, it's about understanding the material, identifying knowledge gaps, and building the test-taking strategies that lead to success.

This guide was designed to help you do exactly that.

Whether you're preparing for a licensing exam, professional certification, or entry-level qualification, this book offers structured practice to reinforce key concepts. You'll find a wide range of multiple-choice questions, each followed by clear explanations to help you understand not just the right answer, but why it's correct.

The content in this guide is based on real-world exam objectives and aligned with the types of questions and topics commonly found on official tests. It's ideal for learners who want to:

- Practice answering questions under realistic conditions,
- Improve accuracy and speed,
- Review explanations to strengthen weak areas, and
- Approach the exam with greater confidence.

We recommend using this book not as a stand-alone study tool, but alongside other resources like flashcards, textbooks, or hands-on training. For best results, we recommend working through each question, reflecting on the explanation provided, and revisiting the topics that challenge you most.

Remember: successful test preparation isn't about getting every question right the first time, it's about learning from your mistakes and improving over time. Stay focused, trust the process, and know that every page you turn brings you closer to success.

Let's begin.

How to Use This Guide

This guide is designed to help you study more effectively and approach your exam with confidence. Whether you're reviewing for the first time or doing a final refresh, here's how to get the most out of your Examzify study guide:

1. Start with a Diagnostic Review

Skim through the questions to get a sense of what you know and what you need to focus on. Your goal is to identify knowledge gaps early.

2. Study in Short, Focused Sessions

Break your study time into manageable blocks (e.g. 30 – 45 minutes). Review a handful of questions, reflect on the explanations.

3. Learn from the Explanations

After answering a question, always read the explanation, even if you got it right. It reinforces key points, corrects misunderstandings, and teaches subtle distinctions between similar answers.

4. Track Your Progress

Use bookmarks or notes (if reading digitally) to mark difficult questions. Revisit these regularly and track improvements over time.

5. Simulate the Real Exam

Once you're comfortable, try taking a full set of questions without pausing. Set a timer and simulate test-day conditions to build confidence and time management skills.

6. Repeat and Review

Don't just study once, repetition builds retention. Re-attempt questions after a few days and revisit explanations to reinforce learning. Pair this guide with other Examzify tools like flashcards, and digital practice tests to strengthen your preparation across formats.

There's no single right way to study, but consistent, thoughtful effort always wins. Use this guide flexibly, adapt the tips above to fit your pace and learning style. You've got this!

Questions

1. What is the purpose of the .terraform.lock.hcl file?

A. Preventing Terraform runs from occurring

B. Tracking provider dependencies

C. Storing references to workspaces locked

D. There is no such file

2. When you modify a resource in Azure Portal that is managed by Terraform, when does Terraform reflect that change in its state?

A. During the next plan or apply

B. Immediately in the current state file

C. In a separate temporary state file

D. Never updates the state

3. Which option cannot be used to keep secrets out of Terraform configuration files?

A. Secure string

B. Terraform provider

C. Environment variables

D. The -var flag

4. When writing Terraform code, how should you indent each nesting level relative to the level above it?

A. With four spaces

B. With a tab

C. With three spaces

D. With two spaces

5. Which file stores the Terraform state locally?

A. terraform.tfstate

B. terraform.state

C. state.tf

D. terraform.json

6. Which statement describes the timing of state updates in Terraform when planning or applying changes?

A. Immediately updates the state during the next plan or apply

B. State is not updated by Terraform at all

C. State is refreshed only by manual edits

D. Updates during the next plan or apply

7. A Terraform apply cannot _____ infrastructure.

A. change

B. destroy

C. provision

D. import

8. Most Terraform providers interact with _____.

A. API

B. VCS Systems

C. Shell scripts

D. None of the above

9. If a DevOps team adopts AWS CloudFormation as their standardized method for provisioning public cloud resources, which scenario poses a challenge?

A. The team is asked to build a reusable code base that can deploy resources into any AWS region

B. The team is asked to manage a new application stack built on AWS-native services

C. The organization decides to expand into Azure and wishes to deploy new infrastructure using their existing codebase

D. The DevOps team is tasked with automating a manual provisioning process

10. Which outcome is expected when a creation-time provisioner fails?

A. The resource will be tainted

B. The plan is aborted

C. No effect

D. The resource will be destroyed

Answers

1. B
2. A
3. A
4. D
5. A
6. D
7. D
8. A
9. C
10. A

Explanations

1. What is the purpose of the .terraform.lock.hcl file?

A. Preventing Terraform runs from occurring

B. Tracking provider dependencies

C. Storing references to workspaces locked

D. There is no such file

The file serves to lock provider versions so runs are reproducible. It records which providers your configuration uses, the exact versions that were resolved, their sources, and a checksum for each provider binary. When you run Terraform init again, Terraform reads this lock file and installs the same provider versions, ensuring consistent behavior across machines and environments. This prevents unexpected upgrades from happening just because new provider versions exist. The lock file is generated and updated by Terraform and is about locking provider dependencies, not about blocking runs or storing workspace data.

2. When you modify a resource in Azure Portal that is managed by Terraform, when does Terraform reflect that change in its state?

A. During the next plan or apply

B. Immediately in the current state file

C. In a separate temporary state file

D. Never updates the state

Terraform tracks resources in a state file and only learns about outside changes when it refreshes that state. If you tweak a resource in Azure Portal, Terraform's view becomes out of sync. When you run the next plan (or apply), Terraform refreshes its state by querying Azure for the current attributes of the resource. This refresh updates the in-memory state to reflect the real-world resource, and the plan will show any drift. When you then apply, the state file itself is updated to match the actual resource, bringing everything back into alignment. So the change is reflected the next time you run plan or apply.

3. Which option cannot be used to keep secrets out of Terraform configuration files?

A. Secure string

B. Terraform provider

C. Environment variables

D. The -var flag

In Terraform, you want to avoid placing secret values directly in configuration files and instead provide them at runtime or via external sources. The only option that would not help with that is a secure string, because it implies embedding the secret value as a literal string in the configuration. That would keep secrets in the code, defeating the goal of externalizing them. Environment variables and the -var flag are common ways to supply secrets without hardcoding them in .tf files, and providers can be configured to read credentials from environment variables or secret stores, further keeping secrets out of the configuration itself.

4. When writing Terraform code, how should you indent each nesting level relative to the level above it?

A. With four spaces

B. With a tab

C. With three spaces

D. With two spaces

In Terraform's HCL, indent each nesting level by two spaces. This means the first level inside a block uses two spaces, the next level uses four spaces, and so on. Using two spaces per level keeps code readable and aligns with the official style shown in examples, and avoids the inconsistencies that can come from tabs or other indentation amounts. For example: resource "aws_instance" "web" { ami = "ami-12345678" instance_type = "t2.micro" network_interface { device_index = 0 } } Here, top-level lines inside the resource start with two spaces; the nested block inside has four spaces, showing the two-space-per-level rule. Tabs are discouraged because their width can vary between editors, and options like three or four spaces per level aren't the standard in Terraform.

5. Which file stores the Terraform state locally?

A. terraform.tfstate

B. terraform.state

C. state.tf

D. terraform.json

Terraform keeps a local state file that tracks what it manages and how resources map to real-world objects. By default, when you don't configure a different backend, Terraform writes this state to a file named terraform.tfstate in your working directory. This file is JSON and includes resource addresses, IDs assigned by providers, and current attributes and dependencies, which is what allows Terraform to plan changes accurately and reconcile drift. A backup of the previous state is kept as terraform.tfstate.backup. If you configure a remote backend (like S3, Consul, etc.), the state is stored remotely, and there isn't a local terraform.tfstate file. The other file name options don't match Terraform's local-state convention, so terraform.tfstate is the correct choice.

6. Which statement describes the timing of state updates in Terraform when planning or applying changes?

A. Immediately updates the state during the next plan or apply

B. State is not updated by Terraform at all

C. State is refreshed only by manual edits

D. Updates during the next plan or apply

Terraform keeps a state file that represents your real infrastructure. Before planning changes, it refreshes that state from the actual resources so the plan is based on what exists now. After a successful apply, it writes the updated state back to the state file (or backend) to reflect the new reality. So state updates occur as part of the next plan (via refresh) and are definitely written during the apply. That's why this description matches how Terraform manages state: it is updated during the plan phase to reflect current reality, and then updated again after applying changes to record the new reality.

7. A Terraform apply cannot _____ infrastructure.

A. change

B. destroy

C. provision

D. import

The thing being tested is how Terraform brings existing infrastructure under its management. Terraform apply takes the configuration and makes the real world match it by creating, updating, or deleting resources as needed. However, it does not bring existing resources into Terraform's state on its own—that's what the separate operation is for. To manage resources that already exist, you first use an import step to record those resources in Terraform's state, linking them to the corresponding configuration. Once a resource is imported, apply can then manage it according to the config. So the best choice is that Terraform apply cannot perform the import action itself. It can create new resources, change existing ones to match the configuration, or destroy resources as dictated by the plan, but importing existing infrastructure is done via a separate import operation before applying.

8. Most Terraform providers interact with _____.

A. API

B. VCS Systems

C. Shell scripts

D. None of the above

Providers connect Terraform to external services by talking to that service's API to create and manage resources. They implement the resource lifecycle by issuing API requests, handling authentication, and interpreting responses to keep Terraform's state in sync. This is why most providers interact via APIs—examples include the AWS provider calling AWS APIs to manage EC2 instances or S3 buckets, and the Kubernetes provider using the Kubernetes API. Version control systems or shell scripts aren't the primary mechanism for managing resources in the provider itself (they may appear in other parts of workflows, but not as the provider's core interface).

9. If a DevOps team adopts AWS CloudFormation as their standardized method for provisioning public cloud resources, which scenario poses a challenge?

A. The team is asked to build a reusable code base that can deploy resources into any AWS region

B. The team is asked to manage a new application stack built on AWS-native services

C. The organization decides to expand into Azure and wishes to deploy new infrastructure using their existing codebase

D. The DevOps team is tasked with automating a manual provisioning process

Relying on a single provisioning tool across multiple clouds creates portability challenges. CloudFormation is AWS-specific, understanding and provisioning AWS resources only. Trying to apply the same codebase to Azure would not work out of the box because Azure uses different resource types and deployment mechanisms (like ARM templates or Azure Resource Manager tooling). So expanding into Azure with the existing CloudFormation templates would require rewriting or swapping to a multi-cloud tool, which is why this scenario is the tricky one. The other scenarios fit well with CloudFormation. Templates can be reused across AWS regions to deploy the same stack in different geographic locations, as long as the resources are available in those regions. AWS-native services are exactly what CloudFormation is designed to provision, so building a new stack on AWS services aligns with the tool. Automating a manual provisioning process is a direct use case for CloudFormation, converting ad-hoc steps into repeatable infrastructure as code.

10. Which outcome is expected when a creation-time provisioner fails?

A. The resource will be tainted

B. The plan is aborted

C. No effect

D. The resource will be destroyed

Creation-time provisioners run right after a resource is created to configure it further. If that configuration step fails, Terraform marks the resource as tainted. Tainting tells Terraform that this resource is in an unhealthy state and should be replaced, so on the next apply it will be destroyed and recreated to achieve the desired final state. So the expected outcome is that the resource becomes tainted, leading to recreation on a future apply. This isn't about nothing happening or the plan being aborted by default, and the resource isn't destroyed immediately during the same apply due to the failed provisioner.

Next Steps

Congratulations on reaching the final section of this guide. You've taken a meaningful step toward passing your certification exam and advancing your career.

As you continue preparing, remember that consistent practice, review, and self-reflection are key to success. Make time to revisit difficult topics, simulate exam conditions, and track your progress along the way.

If you need help, have suggestions, or want to share feedback, we'd love to hear from you. Reach out to our team at hello@examzify.com.

Or visit your dedicated course page for more study tools and resources:

https://hashicorpterraform.examzify.com

We wish you the very best on your exam journey. You've got this!