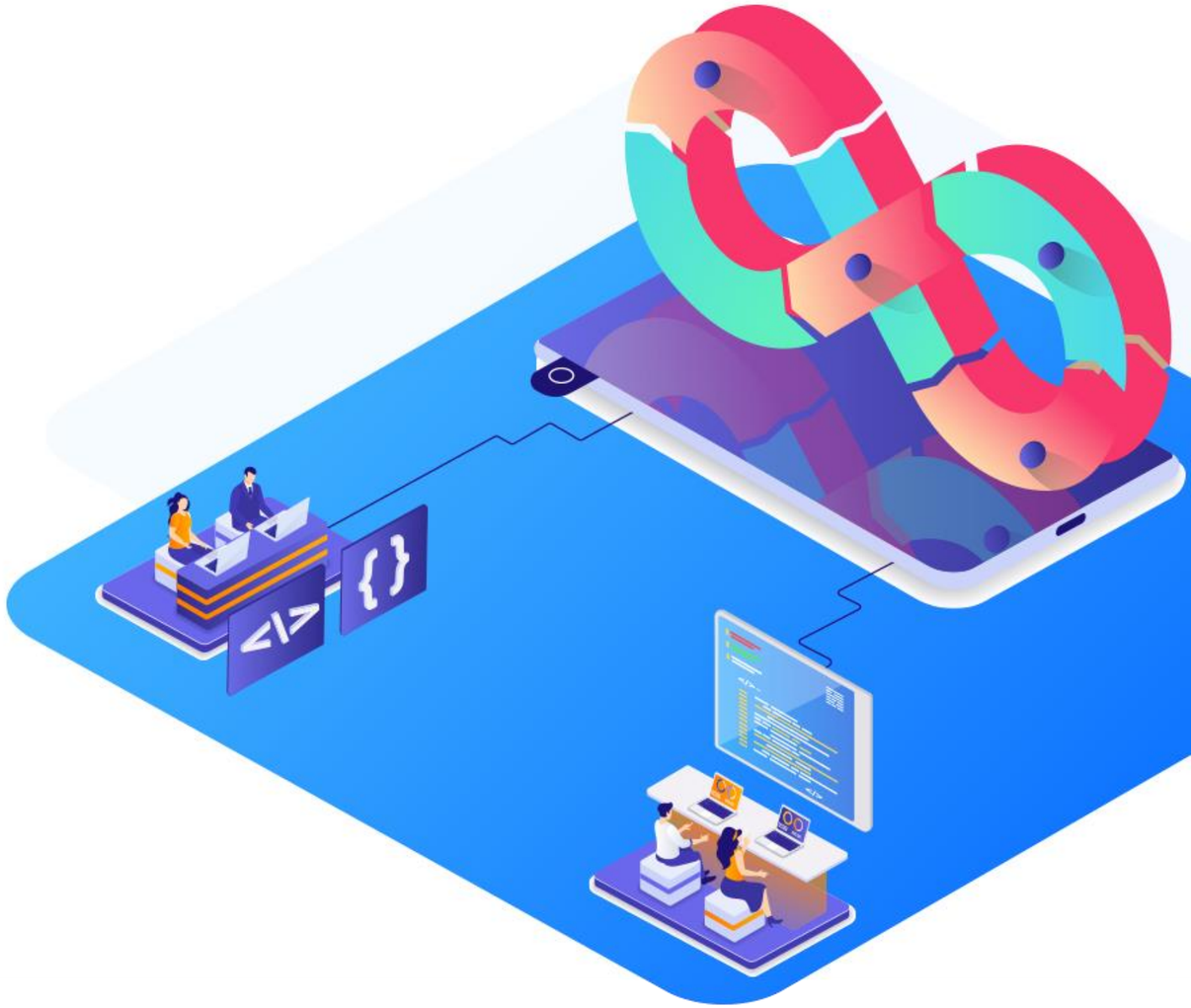
Configuration Management with Ansible and Terraform



Terraform State

Learning Objectives

By the end of this lesson, you will be able to:

- Classify various core components within Terraform state files to understand the structure and contents of state files
- Compare the differences between local and remote Terraform backends to choose the best backend solution
- Implement state locking mechanisms in different backends to ensure consistency and prevent conflicts during state file updates
- Build the necessary configuration to migrate Terraform state files between different backends

Learning Objectives

By the end of this lesson, you will be able to:

- Review post-migration steps to ensure successful state file migration and validate infrastructure consistency
- Develop best practices for handling and securing sensitive data in Terraform state files to protect confidential information and ensure compliance

Deep Dive into Terraform State

What Is Terraform State?

It is a file that tracks the state of infrastructure managed by Terraform.

Terraform state files contain the details of all resources along with their status, such as **ACTIVE**, **DELETED**, or **PROVISIONING**.

Terraform State File: Example

The following example shows how a basic Terraform state file describes a single resource:

```
{
  "version": 4,
  "terraform_version": "1.0.11",
  "resources": [
    {
      "type": "aws_instance",
      "name": "example",
      "instances": [
        {
          "attributes": {
            "id": "i-0123456789abcdef0",
            "ami": "ami-0c55b159cbfafe1f0",
            "instance_type": "t2.micro"
          }
        }
      ]
    }
  ]
}
```

Purpose of Using Terraform State

The following are the key reasons why Terraform state is essential for infrastructure management:

01 It acts as a database to map configurations to real-world resources.

02 It ensures that each resource in the configuration corresponds to a specific remote object.

03 It enables accurate tracking and management of infrastructure changes.

Purpose of Using Terraform State

The following are the key reasons why Terraform state is essential for infrastructure management:

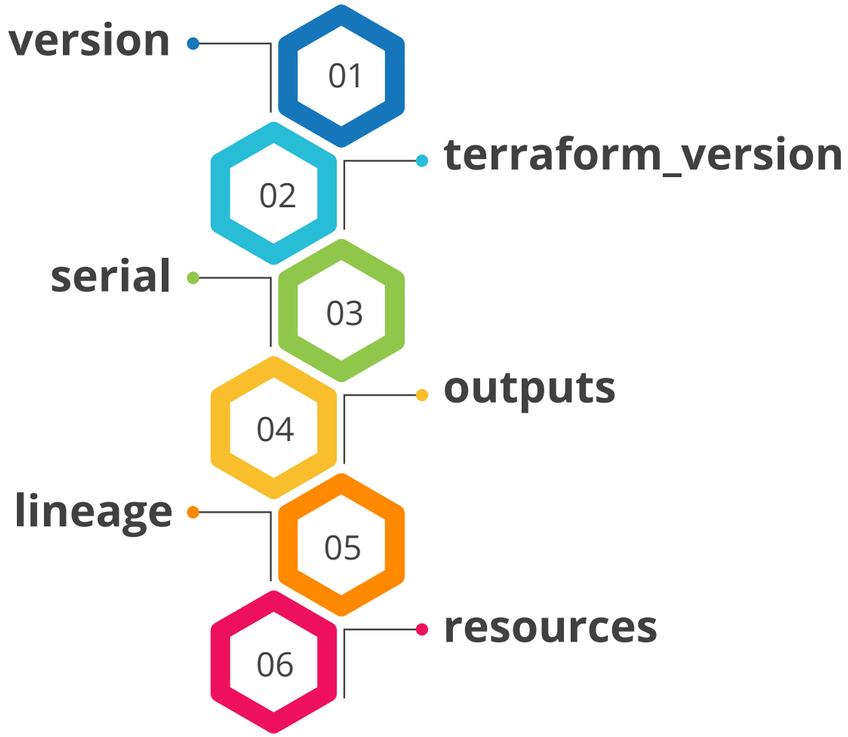
04 It tracks dependencies between resources to ensure correct order of operations.

05 It improves planning and application speeds by caching the attribute values of all resources.

06 It reduces the need for querying all resources, which is beneficial for large infrastructures.

Core Components of Terraform State Files

The following are the core components of a terraform state file:



Core Components of Terraform State File

- **version:** Specifies the version of the state file format

Example:

```
"version": 4
```

- **terraform_version:** Indicates the version of Terraform used to manage the state

Example:

```
"terraform_version": "1.0.11"
```

- **serial:** A number that increments with every state change

Example:

```
"serial": 1
```

Core Components of Terraform State File

- **outputs:** Stores output values defined in the Terraform configuration

Example:

```
"outputs": {
  "instance_ip": {
    "value": "203.0.113.1"
  }
}
```

- **lineage:** Unique identifier for the state file lineage

Example:

```
lineage": "e12345f6-7890-1234-5678-9abcdef01234"
```

Core Components of Terraform State File

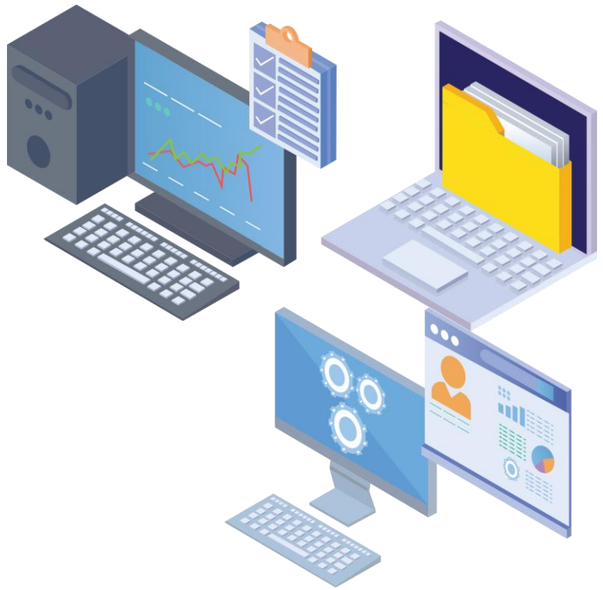
- **resources:** Lists of all managed resources, including their types, names, and attributes

Example:

```
"resources": [
  {
    "type": "aws_instance",
    "name": "example",
    "instances": [
      {
        "attributes": {
          "id": "i-0123456789abcdef0",
          "ami": "ami-0c55b159cbfafe1f0",
          "instance_type": "t2.micro"
        }
      }
    ]
  }
]
```

Resources in State File

It contains detailed information about each managed resource, including its type, name, and provider.



This section helps Terraform track and manage the current state of the infrastructure resources.

Key Elements in Resource Section

- **type:** Specifies the type of resource

Example:

```
"type": "aws_instance"
```

- **name:** Indicates the name of the resource as defined in the Terraform configuration

Example:

```
"name": "example_instance"
```

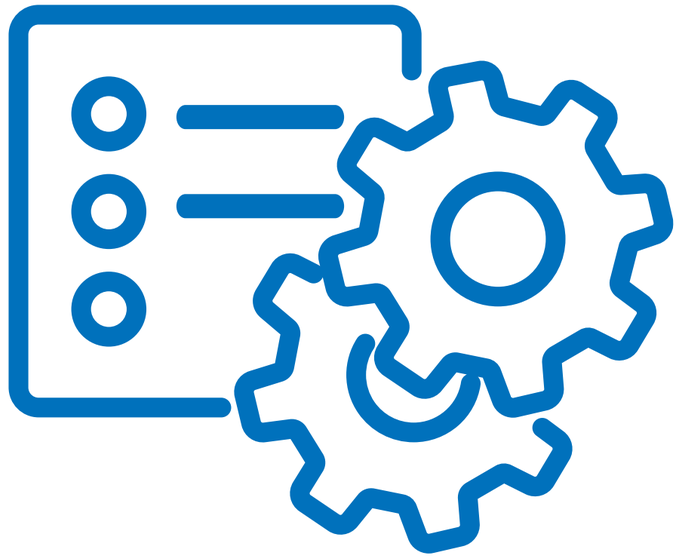
- **provider:** Identifies the provider used for managing the resource

Example:

```
"provider": "provider[\"registry.terraform.io/hashicorp/aws\"]"
```

Instances in State File

It provides the current state attributes and dependencies for each resource, ensuring accurate tracking and management.



Each resource can have multiple instances.

Key Elements in Instance Section

- **attributes:** States attributes of the resource instance, such as ID, AMI, and instance type

Example:

```
"attributes": {
  "id": "i-0123456789abcdef0",
  "ami": "ami-0c55b159cbfafe1f0",
  "instance_type": "t2.micro"
}
```

- **dependencies:** Lists other resources that the current resource instance depends on

Example:

```
"dependencies": [
  "module.vpc.aws_vpc.main"
]
```

Quick Check

You are reviewing a Terraform state file to understand how your infrastructure is represented. Which section of the state file would you look at to find detailed information about its current state and attributes?

A. root_module section

B. resources section

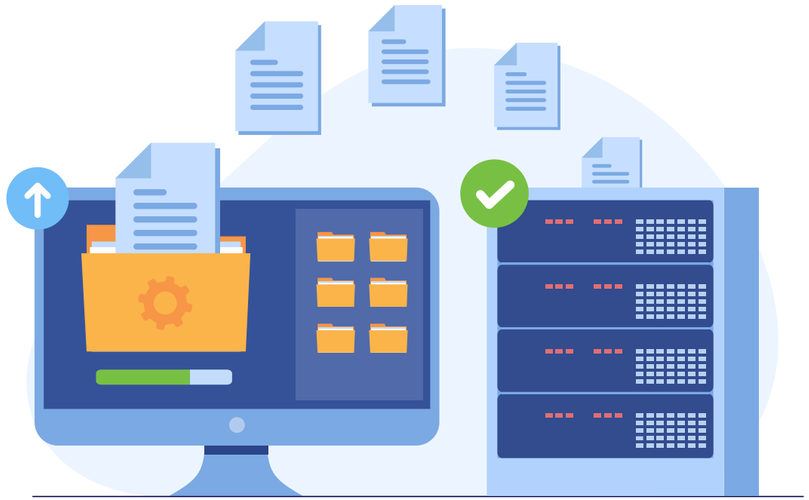
C. instances section under the resources section

D. outputs section

Terraform Backends

What Is a Terraform Backend?

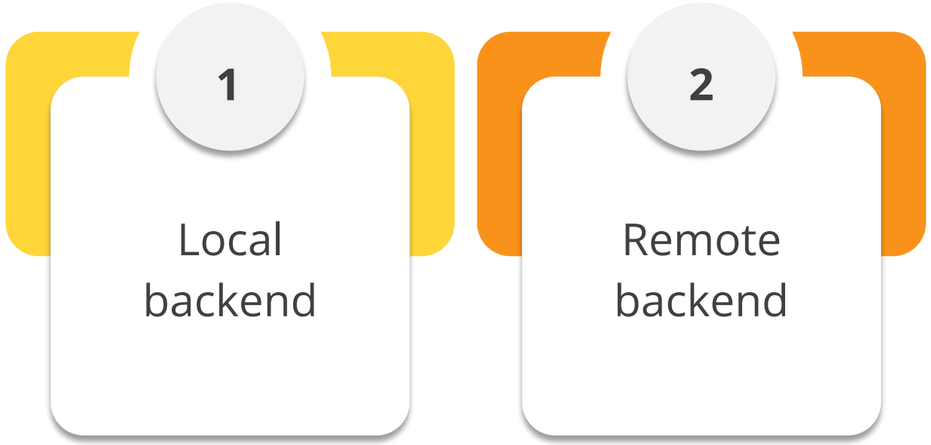
It defines where and how Terraform stores state data. Configuring a backend to store the state file for the user infrastructure is crucial.



Without a backend, the user will have to manually manage the state file, which can lead to errors and inconsistencies.

Types of Terraform Backends

The following are the main types of Terraform backends, each suited to different project needs:



Local Backend

It stores the state file on the local disk. This type of backend is suitable for individual projects.

Example:

```
terraform {
  backend "local" {
    path = "terraform.tfstate"
  }
}
```

Remote Backend

It stores the state file in a remote, shared store. Several remote backend providers are available, such as Amazon S3, Azure Storage, Google Cloud Storage, and HashiCorp Consul.

Example with Amazon S3:

```
terraform {
  backend "s3" {
    bucket = "my-bucket"
    key    = "path/to/my/key"
    region = "us-west-2"
  }
}
```

Example with Google Cloud Storage:

```
terraform {
  backend "gcs" {
    bucket = "my-terraform-state"
    prefix = "terraform/state"
  }
}
```

Comparison of Local and Remote Backends

The table below compares the features of local and remote backends in Terraform:

Features	Local backend	Remote backend
State storage	Stored on local disk	Stored in a remote, shared store (for example, AWS S3, GCS)
Suitability	Suitable for individual projects	Suitable for team and collaborative projects
Collaboration	Limited to single user	Allows multiple users to access the same state
State locking	No built-in locking	Supports state locking to prevent concurrent operations
Security	Relies on local file system security	Supports encryption and secure access controls
Scalability	Limited to local storage capacity	Scales with remote storage solutions

Assisted Practice

Managing Terraform state using default local backend **Duration: 10 Min.**

Problem Statement:

You've been assigned a task to illustrate the steps involved in managing Terraform state using the default local backend, making infrastructure changes, and verifying the updated state.

Assisted Practice: Guidelines

Steps to be followed:

1. Show current state
2. Show state file location
3. Modify, plan, and execute changes
4. Show new state and state backup

ASSISTED PRACTICE

Assisted Practice

Authenticating Terraform state backend **Duration: 15 Min.**

Problem Statement:

You've been assigned a task to authenticate and manage Terraform state using two different backend types, S3 standard backend and remote enhanced backend.

Assisted Practice: Guidelines

Steps to be followed:

1. Authenticate S3 standard backend
2. Authenticate remote enhanced backend

ASSISTED PRACTICE

Assisted Practice

Configuring Terraform state backend storage **Duration: 20 Min.**

Problem Statement:

You've been assigned a task to configure and manage Terraform state using the AWS S3 backend for ensuring reliable state storage and management.

Assisted Practice: Guidelines

Steps to be followed:

1. Enable versioning on the S3 bucket
2. Enable encryption on the S3 bucket
3. Enable locking for the S3 backend
4. Remove existing resources with the terraform destroy command

ASSISTED PRACTICE

Assisted Practice

Configuring Terraform remote state backend **Duration: 20 Min.**

Problem Statement:

You've been assigned a task to configure and manage Terraform state using the remote enhanced backend with Terraform Cloud for ensuring efficient and collaborative state management.

Assisted Practice: Guidelines

Steps to be followed:

1. Sign in to Terraform Cloud platform
2. Update the Terraform configuration to use remote enhanced backend
3. Re-initialize Terraform and validate the remote backend with Terraform Cloud
4. Provide secure credentials for remote runs
5. Remove existing resources with the terraform destroy command

Quick Check

You are setting up Terraform to manage your infrastructure and need to decide between using a local backend and a remote backend. Which of the following scenarios best justifies the use of a remote backend over a local backend?

A. You are working on a personal project with a small, static infrastructure.

B. You need to collaborate with a team and ensure state consistency across multiple team members.

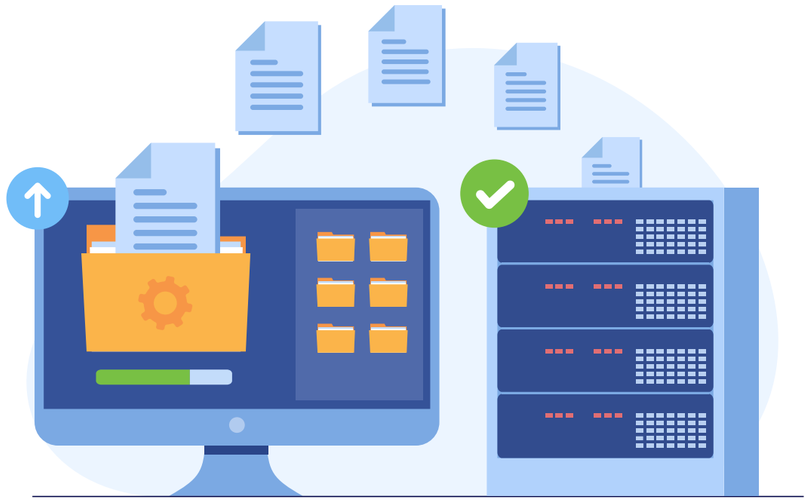
C. You want to quickly prototype changes without worrying about state management.

D. You have limited access to cloud services and prefer to keep all files local.

Implementing State Lock

What Is State Locking?

It is a mechanism to prevent concurrent operations on the same state file. It ensures that only one operation can modify the state at a time, avoiding conflicts and potential corruption.



When a Terraform operation begins, it attempts to acquire a lock on the state file. If the lock is acquired, the operation proceeds; if not, the operation waits until the lock is available or fails.

Purpose of Using State Locking

The following are the key reasons why state locking is essential in Terraform:

Consistency

Ensures that all changes to the infrastructure are applied sequentially, maintaining the correct order of operations

Integrity

Prevents multiple users or processes from making simultaneous changes, which could lead to inconsistent state and infrastructure drift

Purpose of Using State Locking

The following are the key reasons why state locking is essential in Terraform:

Reliability

Manages automatically for supported backends, enhancing reliability without additional configuration

Safety

Reduces the risk of state corruption by ensuring that only one Terraform process can write to the state file at a time

Steps to Configure State Locking in Terraform

- Define the backend configuration in your Terraform configuration file
- Specify the necessary parameters for state locking, such as a DynamoDB table for AWS S3

Enable State Locking

State locking can be enabled by configuring a backend that supports locking. Many backends support state locking, such as:

AWS S3 with DynamoDB

Azure Blob Storage

Google Cloud Storage

Enable State Locking: Example

The following is an example of how state locking is enabled in **AWS S3**, where you need to specify the **dynamodb_table** that will be used for state locking:

```
terraform {
  backend "s3" {
    bucket         = "my-bucket"
    key            = "path/to/my/key"
    region         = "us-west-2"
    dynamodb_table = "my-lock-table"
  }
}
```

Enable State Locking: Example

Azure Blob Storage and **Google Cloud Storage** can automatically handle state locking when configuring the backend.

Azure Blob Storage:

```
terraform {
  backend "azurerm" {
    storage_account_name = "mystorageaccount"
    container_name       = "mycontainer"
    key                  = "terraform.tfstate"
  }
}
```

Google Cloud Storage:

```
terraform {
  backend "gcs" {
    bucket = "my-terraform-state"
    prefix = "terraform/state"
  }
}
```

Assisted Practice

Implementing Terraform state locking **Duration: 10 Min.**

Problem Statement:

You've been assigned a task to implement state locking in Terraform, ensuring concurrent operations on the Terraform state file are managed to prevent conflicts and corruption.

Assisted Practice: Guidelines

Steps to be followed:

1. Update the Terraform configuration
2. Generate a Terraform state lock
3. Specify a Terraform lock timeout

ASSISTED PRACTICE

Quick Check

You are configuring Terraform to manage your infrastructure and want to ensure that only one person can make changes to the state file at any given time to avoid conflicts. Which feature should you enable, and how can you configure it?

A. Enable state encryption and set up encryption keys in your backend configuration

B. Enable state locking and configure your backend with appropriate locking mechanisms

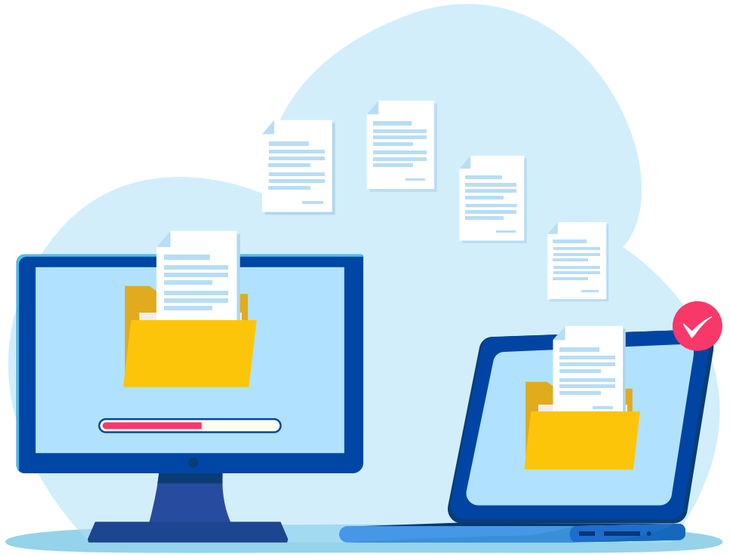
C. Enable remote state storage and use a cloud provider to store the state file

D. Enable state consistency checks and run periodic validations of the state file

Implementing State Migration

What Is State Migration?

It refers to the process of moving the Terraform state file from one backend to another.



This involves changing the storage location from a local file to a remote storage service or moving between different remote storage services, depending on the user’s requirements.

Purpose of Using State Migration

The following are the key reasons why state migration is essential in Terraform:

Improved scalability

- Migrating the state to a more scalable backend can better handle larger infrastructures and frequent updates.

Enhanced security

- Moving the state to a backend with superior security features, such as encryption and access controls, ensures data protection and compliance with security standards.

Centralized management

- Consolidating the state files to a central backend facilitates easier team collaboration and management, streamlining workflow and enhancing productivity.

Where to Use State Migration?

The following are some real-world scenarios where state migration is used:

Local to remote backend

When a project outgrows local state management, it requires shifting to a remote backend like AWS S3, Azure Blob Storage, or Google Cloud Storage.

Reorganizing resources

When significant infrastructure changes, it necessitates reorganizing resources and their state files.

Changing cloud providers

When migrating infrastructure from one cloud provider to another, it maintains consistency and continuity.

Ways to Migrate Terraform State

Using terraform init with a new backend:

- Update the backend configuration in your Terraform files
- Run **terraform init** to initialize the new backend

Step 1: Example

```
terraform {
  backend "s3" {
    bucket = "new-bucket"
    key    = "new-path/to/state"
    region = "us-west-2"
  }
}
```

Step 2:

```
terraform init
```

Ways to Migrate Terraform State

Manually migrating state files:

- Copy the state file from the old backend to the new backend
- Verify that the state file is accessible from the new location

Example:

```
aws s3 cp s3://old-bucket/old-path/to/state s3://new-bucket/new-path/to/state
```

Ways to Migrate Terraform State

Using terraform state commands:

- **terraform state pull** to download the state file
- **terraform state push** to upload the state file to the new backend

Example:

```
terraform state pull > state.tfstate
terraform state push state.tfstate
```

Post Migration Steps

Verifying the migration:

- Run **terraform plan** to ensure that Terraform can read the state from the new backend
- Check for any discrepancies or errors in the state

Example:

```
terraform plan
```

Post Migration Steps

Updating terraform configurations:

- Update your Terraform configuration files to reference the new backend
- Ensure that all team members update their configurations accordingly

Example: original configuration

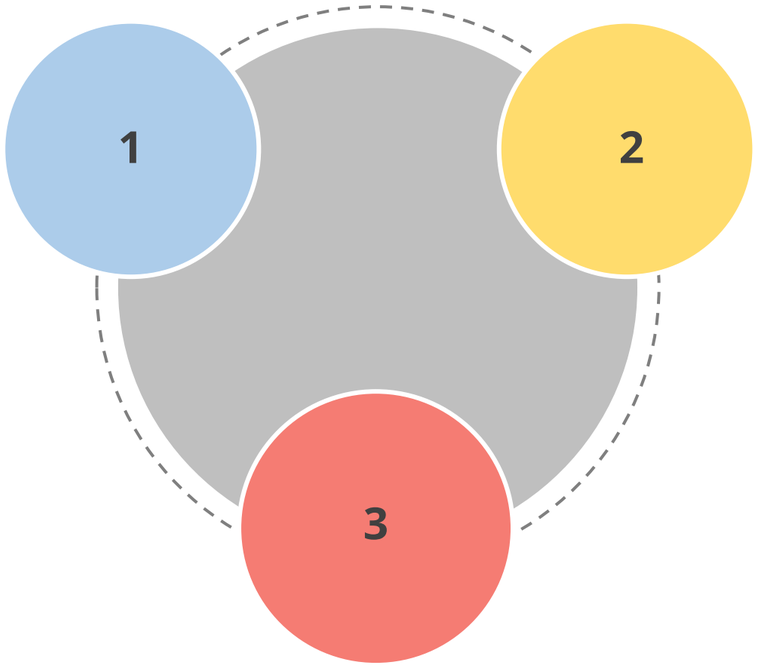
```
terraform {
  backend "local" {
    path = "terraform.tfstate"
  }
}
```

Example: Updated configuration

```
terraform {
  backend "s3" {
    bucket = "new-bucket"
    key    = "new-path/to/state"
    region = "us-west-2"
  }
}
```

Best Practices for State Migration

Regularly back up state files to ensure data integrity and prevent data loss



Thoroughly test the new backend before completing the migration

Document the migration process and any issues encountered

Assisted Practice

Migrating Terraform state **Duration: 15 Min.**

Problem Statement:

You've been assigned a task to migrate Terraform state between different backends, including the default local backend, AWS S3 backend, and Terraform Cloud remote backend for achieving seamless state management across various environments.

Assisted Practice: Guidelines

Steps to be followed:

1. Use Terraform default local backend
2. Migrate state to S3 backend
3. Migrate state to remote backend
4. Migrate back to local backend

ASSISTED PRACTICE

Quick Check

You need to migrate your Terraform state to a new backend to improve state management. What is the best practice to perform this migration while ensuring the state file remains consistent and accurate?

A. Manually edit the state file to update the backend configuration

B. Use terraform state pull to download the state file and then upload it to the new backend

C. Update the Terraform configuration with the new backend and run terraform init to handle the migration

D. Create a new state file in the new backend and manually import all resources

Handling Sensitive Data in Terraform State

What Is Sensitive Data in Terraform State?

It refers to information that needs protection due to its confidential nature.

This includes database passwords, user passwords, private keys, and other credentials.

Types of Sensitive Data

The following are various types of sensitive data in Terraform state:

1. Resource IDs and attributes
2. Initial passwords
3. Private keys
4. Other sensitive configurations

Types of Sensitive Data

Resource IDs and attributes

- Unique identifiers and metadata for resources that may include sensitive information
- **Example**: AWS instance IDs, IP addresses

Initial passwords

- Default or initial passwords for services and databases
- **Example**: Initial database passwords set during resource creation

Types of Sensitive Data

Private keys

- Cryptographic keys used for securing communications and data
- **Example**: SSH keys, TLS certificates

Other sensitive configurations

- Any other configuration data that should remain confidential
- **Example**: API keys, tokens, and environment variables

Recommendations for Handling Sensitive Data

Treating state as sensitive data:

- Always treat the Terraform state file as sensitive data because it contains detailed information about your infrastructure
- Ensure that state files are backed up regularly to prevent data loss

Using remote backend:

- It provides better security features such as encryption at rest and access control.
- It also allows multiple team members to access the state file securely without having to share it manually.

Verifying backend configurations:

- Regularly test and verify your backend configurations to ensure that they are set up correctly and securely
- This includes checking that encryption is enabled and access controls are in place.

Recommendations for Handling Sensitive Data

Configuring encryption:

- Choose a backend that supports encryption
- Enable encryption in the backend configuration

Example: Configuration for AWS S3 with encrypt option enabled

```
terraform {
  backend "s3" {
    bucket = "my-bucket"
    key    = "path/to/my/key"
    region = "us-west-2"
    encrypt = true
  }
}
```

Assisted Practice

Managing sensitive data in Terraform state **Duration: 10 Min.**

Problem Statement:

You've been assigned a task to manage sensitive data within Terraform state files by viewing them in raw format and suppressing sensitive information.

Assisted Practice: Guidelines

Steps to be followed:

1. View Terraform state in raw format
2. Suppress sensitive information
3. View the Terraform state file

ASSISTED PRACTICE

Quick Check

You are managing sensitive data within your Terraform state files and want to ensure that this data is protected. What is the best practice for handling sensitive data in Terraform state files?

A. Store the state file in a public repository with limited access

B. Manually remove sensitive data from the state file after each apply

C. Enable encryption for the state file at rest using the backend encryption features

D. Use plain text files for storing sensitive data and exclude them from version control

Key Takeaways

- Terraform state is a file that tracks the state of infrastructure managed by Terraform.
- Terraform state files contain every detail of any resources along with their status, such as ACTIVE, DELETED, or PROVISIONING.
- The Terraform backend defines where and how Terraform stores state data. Configuring backend to store the state file for the infrastructure is crucial.
- State locking is a mechanism to prevent concurrent operations on the same state file. It ensures that only one operation can modify the state at a time, avoiding conflicts and potential corruption.

Key Takeaways

- State migration refers to the process of moving the Terraform state file from one backend to another.
- The Terraform state file should always be treated as sensitive data because it contains detailed information about your infrastructure.
- Regular backups of state files should be performed to prevent data loss.

Managing Terraform State Using Different Backends

Duration: 25 Min.

Project agenda: To perform Terraform state management using different backends for storing and managing the state file securely and efficiently

Description: You work as a junior DevOps engineer in an IT firm. Your company is undertaking a project that involves migrating the Terraform state between various backends for better state management and collaboration. The project aims to leverage Amazon S3 for state storage and DynamoDB for state locking, followed by a migration to Terraform Cloud for enhanced team collaboration.

LESSON-END PROJECT

Managing Terraform State Using Different Backends

Duration: 25 Min.

Perform the following:

1. Configure S3 backend and DynamoDB
2. Update the Terraform configuration for S3 backend
3. Migrate state to remote backend with Terraform Cloud
4. Update the Terraform configuration for remote backend

Expected deliverables: An operational Terraform state management mechanism across S3 and Terraform Cloud backends.

LESSON-END PROJECT

Thank You