SKILLS | DevOps and Cloud Computing

Terraform Provisioners, Workspaces, Remote State, and CI/CD Integration

Objective

- Use Terraform provisioners to execute scripts on infrastructure resources.
- Manage multiple environments (dev, staging, prod) using Terraform workspaces.
- Configure remote state storage and state locking for collaborative workflows.
- Integrate Terraform into CI/CD pipelines to automate infrastructure provisioning and management.

PW SKILLS

Explaining what provisioners are and their role in Terraform.

Let’s see

- In Terraform, provisioners are used to execute scripts or commands on a local or remote machine after a resource is created. They help with bootstrapping, such as installing software, configuring services, or copying files.

Role:

- Execute tasks post-deployment.
- Set up resources after they’re created (e.g., install Apache on an EC2 instance).
- Debug infrastructure issues by running diagnostic scripts.

PW SKILLS

Discuss common provisioners

Let’s discuss

Common Terraform Provisioners (Short & Simple):

1. **Remote-exec**
    - Runs scripts/commands on a remote machine (e.g., an EC2 instance).
    - Requires SSH or WinRM access.
    - Example use: Install packages, configure services after the server is up.
2. **Local-exec**
    - Runs scripts/commands on the local machine (where Terraform is executed).
    - Useful for tasks like sending notifications, calling APIs, or triggering local scripts after resource creation.

PW SKILLS

Demonstrating using a provisioner to install software on a virtual machine

Let’s do it

Example: Using ‘remote-exec’ to Install Software on a VM

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  key_name      = "my-key"

  provisioner "remote-exec" {
    inline = [
      "sudo apt update",
      "sudo apt install -y nginx"
    ]
  }

  connection {
    type        = "ssh"
    user        = "ubuntu"
    private_key = file("~/.ssh/my-key.pem")
    host        = self.public_ip
  }
}
```

PW SKILLS

Explaining why provisioners can break idempotency in Terraform configurations.

Let’s see

- Provisioners can break idempotency in Terraform because they run outside Terraform's state tracking.

Why:

- Terraform can't track or verify what a provisioner does.
- If you reapply, Terraform may rerun the provisioner, causing duplicate actions (e.g., reinstalling software).
- Failures can leave resources in a partially configured state.

PW SKILLS

Discussing when to use provisioners vs. when to rely on configuration management tools (Ansible, Chef, etc.).

Let’s discuss

Use Provisioners when:

- You need quick, simple setup after resource creation.
- Tasks are one-time (e.g., install a single package).
- You don’t have a config management tool in place.

Use Configuration Management Tools (Ansible, Chef, etc.) when:

- You need complex, repeatable configuration.
- You manage multiple servers or roles.
- You want better error handling, version control, and idempotency.

PW SKILLS

Explaining Terraform workspaces and their role in managing environments.

Let’s see

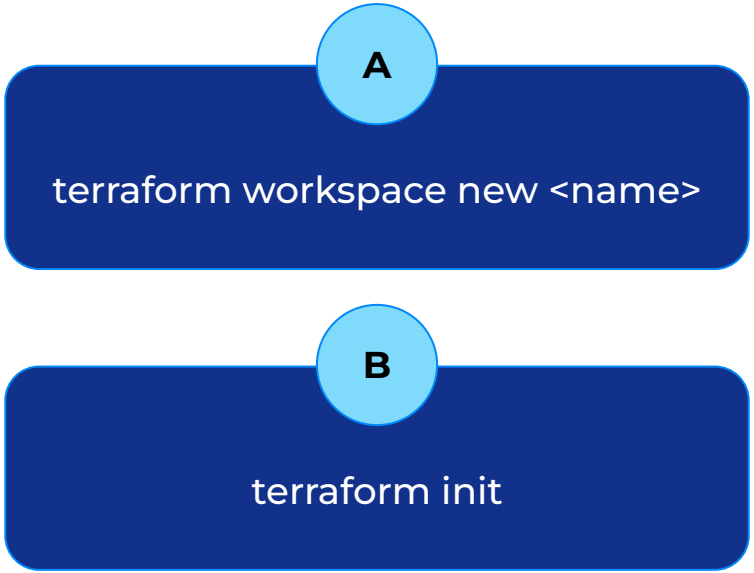
Workspaces allow you to use the same Terraform configuration to manage multiple environments (like dev, staging, prod) with separate state files.

Role:

- Isolate environments using different states.
- Avoid managing multiple copies of config files.
- Helpful for testing changes in dev before applying to prod.

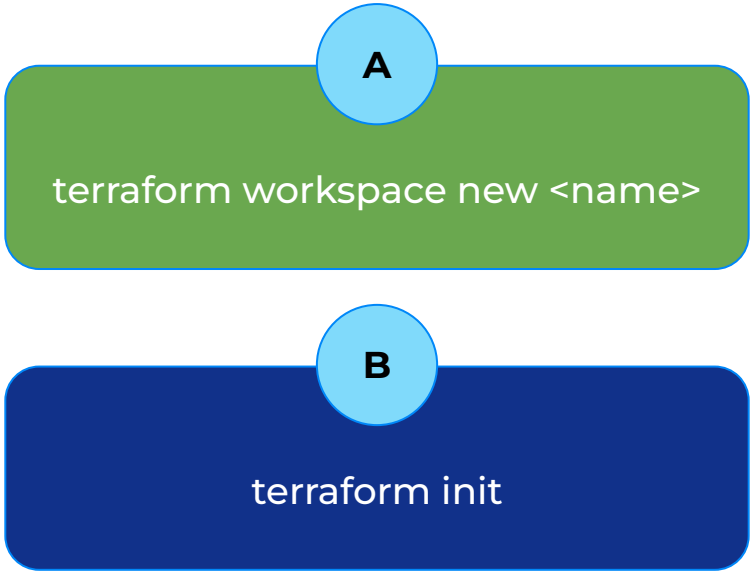
Pop Quiz

Q. Which command is used to create a new Terraform workspace?



Pop Quiz

Q. Which command is used to create a new Terraform workspace?



PW SKILLS

Discussing how workspaces help separate dev, staging, and production environments without duplicating code.

Let’s discuss

How Workspaces Help Separate Environments:

Workspaces let you manage dev, staging, and prod using the same code, but with separate state files.

Benefits:

- No code duplication – reuse the same .tf files.
- Isolated states – changes in one environment don’t affect others.
- Easier to test and promote changes across environments.

PW SKILLS

Demonstrating Terraform workspace commands:

Let’s do it

Terraform Workspace Commands (Short & Simple):

1. **Create a new workspace**

```
terraform workspace new dev
```

2. **Switch to another workspace**

```
terraform workspace select staging
```

Let’s do it

3. **List all workspaces**

```
terraform workspace list
```

4. **Show current workspace**

```
terraform workspace show
```

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Explaining why remote state storage is necessary for team collaboration.

Let’s see

Why Remote State Storage is Needed for Teams:

- Centralizes the state file so everyone works with the same infrastructure snapshot.
- Prevents conflicts from multiple people changing the state locally.
- Enables locking to avoid simultaneous updates.
- Supports versioning and backups.

PW SKILLS

Discussing state locking and how it prevents multiple users from modifying infrastructure at the same time.

Let’s discuss

State locking prevents multiple users from making changes to infrastructure at the same time.

How it works:

- When a user runs terraform apply, Terraform locks the state.
- Others must wait until the lock is released to make changes.
- Prevents conflicts, corruption, and inconsistent state.

Pop Quiz

Q. How does Terraform handle state locking failure in a supported backend?

A

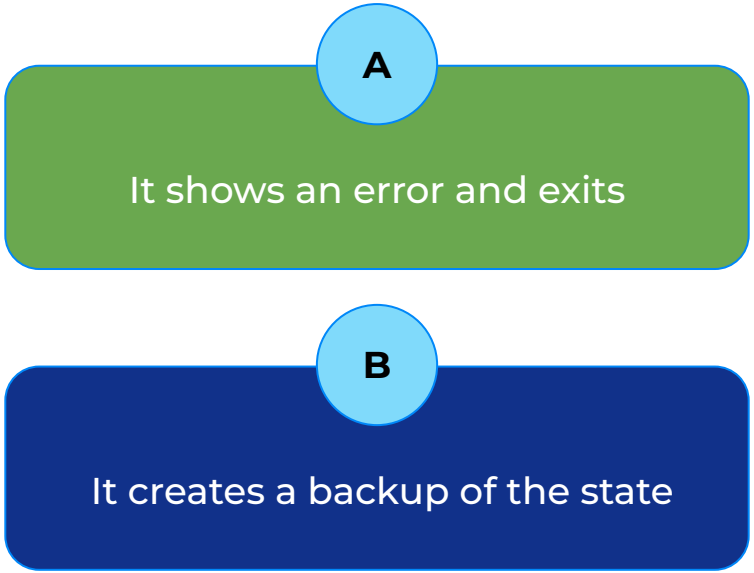
It shows an error and exits

B

It creates a backup of the state

Pop Quiz

Q. How does Terraform handle state locking failure in a supported backend?



PW SKILLS

Demonstrating setting up S3 or GCS as a remote backend with state locking.

Let’s do it

Example: S3 Remote Backend with State Locking

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "envs/dev/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-locks"   # Enables state locking
    encrypt        = true
  }
}
```

PW SKILLS

Discussing state locking mechanisms in Terraform Cloud and remote backends.

Let’s discuss

Terraform Cloud:

- Automatic state locking is built-in.
- Prevents multiple runs at the same time.
- Manages locks and state versioning for you.

Other remote backends (e.g., S3 + DynamoDB):

- Use DynamoDB for locking.
- Only one user can update state at a time.
- Blocks conflicting apply or plan actions.

PW SKILLS

Explaining the impact of concurrent state modifications and how to prevent conflicts.

Let’s see

Impact of Concurrent State Modifications:

- Can cause state file corruption
- Leads to infrastructure drift or unexpected changes
- Makes Terraform runs unreliable

How to Prevent Conflicts:

- Use remote backends with state locking (e.g., Terraform Cloud, S3 + DynamoDB)
- Avoid running apply from multiple machines at once
- Coordinate team actions with version control and automation tools

PW SKILLS

Explaining why Terraform fits into CI/CD workflows for infrastructure automation.

Let’s see

Terraform fits into CI/CD workflows for infrastructure automation because it enables Infrastructure as Code (IaC), allowing you to:

- Automate provisioning
- Ensure consistency
- Enable collaboration
- Support testing and validation
- Improve speed and reliability

PW SKILLS

Discussing using Terraform in GitHub Actions, GitLab CI, or Jenkins to deploy infrastructure.

Let’s discuss

Using Terraform in GitHub Actions, GitLab CI, or Jenkins lets you automate infrastructure deployment by integrating it into your CI/CD pipelines:

- **GitHub Actions:** Use workflows to run ‘terraform init’, ‘plan’, and ‘apply’ on code push or PR.
- **GitLab CI:** Define Terraform steps in ‘.gitlab-ci.yml’ to validate and deploy infrastructure with version control.
- **Jenkins:** Set up jobs or pipelines to execute Terraform commands, often triggered by Git commits.

PW SKILLS

Demonstrating best practices for automated Terraform runs in CI/CD:

Best practices

- **Run terraform plan:** Generate and show planned changes for review before deployment.
- **Use approval workflows:** Require manual approval (e.g., GitHub PR review or GitLab manual job) before terraform apply.
- **Store state securely:** Use remote backends like Terraform Cloud, AWS S3 with DynamoDB, or Azure Blob Storage with proper encryption and locking.

PW SKILLS

Discussing best practices for cost optimization in Terraform-managed cloud resources.

Best practices

- Use cost-effective instance types
- Tag resources
- Automate resource cleanup
- Use variables for scaling
- Monitor and budget

PW SKILLS

Explaining how to automate resource cleanup using scheduled Terraform runs.

Let’s see

You can automate resource cleanup using scheduled Terraform runs by:

- Scheduling CI jobs (e.g., with GitHub Actions, GitLab CI, or Jenkins) to run terraform apply at set times.
- Using lifecycle rules or variables to mark resources (e.g., ttl, auto destroy = true) for cleanup.
- Tagging temporary resources and using logic in .tf files to destroy them on schedule.

PW SKILLS

Demonstrating using tags and policies to track and manage cloud costs.

Let’s do it

Apply tags (e.g., Project, Owner, Environment) to all Terraform-managed resources.

- Use tagging policies in cloud platforms (like AWS Organizations) to enforce consistent tagging.
- Track costs by tags in tools like AWS Cost Explorer, Azure Cost Management, or GCP Billing.
- Automate tagging with modules or Terraform default tags for consistency.

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessio
- Review your lecture notes
- Note down questions and queries regarding this session ar
  consult the teaching assistants

Thanks !