Fidelity Terraform Assets

Project Scope: Personal Banking Management Service

Fidelity International Limited

Infrastructure as Code with Terraform

Personal Banking Management Service

Professional Training Session

Agenda

Fidelity Terraform Assets-python.zip

Terraform Fundamentals

- Infrastructure as Code (IaC) Concepts
- Core Components: Providers, Resources, State, Variables, Outputs
- Managing Environments with Workspaces
- Remote State Management with Backends
- Handling Secrets in Terraform

Fidelity's Innersource Repository

- Structure & Key Modules for Personal Banking
- Contribution Model & Best Practices

Hands-On Exercises

- Exercise 1: Provisioning a Simple Application Server
- Exercise 2: Utilizing an Innersource Module

Conclusion & Key Takeaways

- Best Practices Recap
- Pro-Level Tips
- Q&A

Introduction to Infrastructure as Code (IaC)

What is IaC?

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

Why IaC for Personal Banking?

Consistency

Ensures that development, testing, and production environments are identical, reducing bugs.

Speed & Efficiency

Automates infrastructure deployment, allowing developers to get resources faster.

Version Control

Track changes to your infrastructure just like you track changes to your application code (e.g., using Git).

Security & Compliance

Codified infrastructure makes it easier to enforce security standards and audit for compliance.

Terraform Core Concepts

Terraform is a tool that allows you to build, change, and version infrastructure safely and efficiently.

Providers

Plugins that let Terraform interact with cloud providers (AWS, Azure, GCP), SaaS providers, and other APIs.
Example: AWS provider to create EC2 instances, S3 buckets, etc.

Resources

The infrastructure components you create. This could be a virtual machine, a database, or a DNS record.
Example: aws_instance, aws_db_instance.

State

A file (usually terraform.tfstate) where Terraform stores the current state of your managed infrastructure. This is crucial for Terraform to know what it manages.

Variables & Outputs

Variables: Used to parameterize your configurations, making them reusable and flexible.
Outputs: Return values from your Terraform configuration that can be used by other configurations.

One-stop course for Terraform

- Learn it quickly, easily and effectively
- Hands-on and practical
- Short course = results quicker

Links for VS Code and Plugins

Link to download VS Code: https://code.visualstudio.com
Link for the plugin: https://marketplace.visualstudio.com/items?itemName=HashiCorp.terraform

How to install chocolatey on window

https://chocolatey.org/

How does Terraform work?

How Does Terraform Work?

Core Mechanism

Terraform creates and manages resources on cloud platforms and other services through their **application programming interfaces (APIs)**. Providers enable Terraform to work with virtually any platform or service with an accessible API.

Available Providers

HashiCorp and the Terraform community have written **thousands of providers** to manage many different types of resources and services. All publicly available providers can be found on the **Terraform Registry**.

AWS | Azure | Google Cloud Platform | Kubernetes | Helm | GitHub | Splunk | DataDog

And Many More

Core Terraform Workflow

Three Essential Stages for Infrastructure as Code

Write

Define resources across multiple cloud providers and services. For example, create a configuration to deploy an application on virtual machines in a VPC network with security groups and a load balancer.

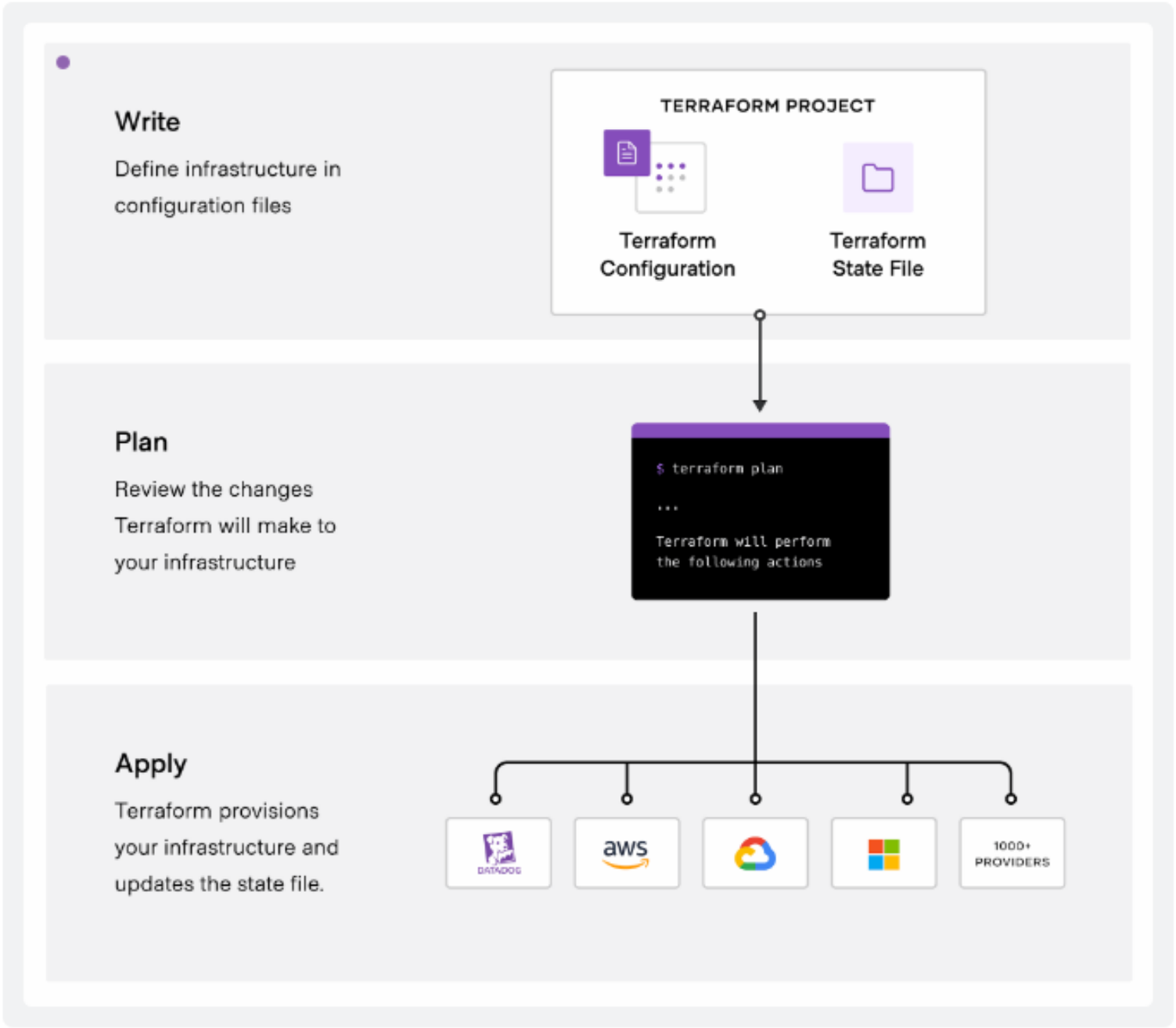
Plan

Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.

Apply

On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies. For example, if you update VPC properties and change the number of virtual machines, Terraform will recreate the VPC before scaling the virtual machines.

Core Terraform Workflow



🎯 Terraform Commands – Your CLI Toolkit

Infrastructure as Code Command Reference

Master the essential commands for managing your infrastructure lifecycle

From initialization to destruction – Your complete CLI guide

🧩 Why These Commands Matter

Terraform is a command-line-driven tool. Every step — from setting up to destroying infrastructure — is performed through commands.

> **Key Insight:** Understanding these commands means knowing the lifecycle of Infrastructure as Code.

- All infrastructure operations are CLI-based
- Commands follow a logical workflow pattern
- Each command has a specific purpose in the IaC lifecycle
- Mastering these commands is essential for DevOps engineers

🚀 Main Workflow Commands

Core Lifecycle

Command	Purpose	Example
`terraform init`	Initializes working directory, downloads providers & modules	`terraform init`
`terraform validate`	Checks syntax & configuration errors	`terraform validate`
`terraform plan`	Shows what Terraform will create/update/destroy	`terraform plan`
`terraform apply`	Executes the plan and builds the infrastructure	`terraform apply`
`terraform destroy`	Removes the infrastructure managed by Terraform	`terraform destroy`

> 🌀 **Lifecycle Tip:** These 5 commands form the standard Terraform workflow used by every DevOps engineer.

⚙️ Other Useful Commands (Part 1)

Command	Description
`terraform fmt`	Formats code to Terraform's standard style
`terraform output`	Displays output values (e.g., public IPs)
`terraform show`	Displays the current state or plan in detail
`terraform providers`	Lists all providers used in configuration
`terraform workspace`	Manages multiple environments (e.g., dev, test, prod)

⚙️ Other Useful Commands (Part 2)

Command	Description
`terraform state`	Inspects or modifies the Terraform state file
`terraform import`	Brings existing infrastructure under Terraform's control
`terraform graph`	Visualizes dependencies between resources

🧠 Advanced / Diagnostic Commands

Command	Description
`terraform console`	Opens an interactive shell to test Terraform expressions
`terraform taint / untaint`	Marks or unmarks a resource for recreation
`terraform login / logout`	Manage authentication for remote Terraform Cloud
`terraform test`	Runs integration tests for Terraform modules
`terraform version`	Shows installed Terraform version
`terraform refresh`	Updates local state to match cloud reality

🌍 Global Options

Option	Description
`-chdir=DIR`	Run Terraform from a different directory
`-help`	Shows command help
`-version`	Prints Terraform version

🧭 Terraform Workflow

Visual Representation

> init ➜ validate ➜ plan ➜ apply ➜ (optional: show/output) ➜ destroy

- 💬 **Key Points to Remember**
- Every Terraform project starts with `terraform init`
- `plan` does not make changes — it's a preview
- Always `validate` and `fmt` before `apply`
- `destroy` is irreversible — always confirm twice!

🧩 Check Your Understanding (Part 1)

Q1: What does terraform init do?
Ans: It initialises the working directory and downloads provider plugins

Q2: Which command checks if your Terraform configuration has syntax errors?
Ans: terraform validate

Q3: What is the purpose of terraform plan?
Ans: To preview the actions Terraform will perform before applying them

Q4: How do you actually create resources in the cloud?
Ans: Using terraform apply

🧩 Check Your Understanding (Part 2)

Q5: What is the difference between terraform show and terraform output?
Ans: show displays full state details; output only shows defined outputs

Q6: When should you run terraform fmt?
Ans: Before committing code, it formats your Terraform files

Q7: What command destroys all the resources managed by Terraform?
Ans: terraform destroy

Q8: How can you test Terraform expressions interactively?
Ans: Using terraform console

💻 Hands-On Classroom Exercise

🎯 **Goal:** Practice the core Terraform workflow end-to-end

Step 1: Setup

mkdir terraform-basics
cd terraform-basics

Step 2: Create Configuration File

Create `main.tf` with the configuration

Step 3: Run Commands in Order

```
terraform init      # Initialise
terraform validate  # Validate syntax
terraform plan      # Preview actions
terraform apply     # Create the bucket
terraform show      # Display state details
terraform destroy   # Clean up
```

Coding exercise and practices

AWS Infrastructure as Code

First-resource: Terraform Configuration for VPC Setup

Implementation Code

```
provider "aws" {
    region = "ap-south-1"
}

resource "aws_vpc" "myvpc" {
    cidr_block = "10.0.0.0/16"
    tags = {
        Name = "myvpc"
    }
}
```

Region Configuration

Deployed in Mumbai (ap-south-1) region

Network Setup

VPC with 10.0.0.0/16 CIDR block

Clean Syntax

Properly structured Terraform code

Resource Tagging

Named resources for easy identification

Terraform Variables

String Variable

```
variable "vpcname" {
type = string
default = "myvpc"
}
```

Boolean Variable

```
variable "enabled" {
default = true
}
```

Tuple Variable

```
variable "mytuple" {
type = tuple([string, number,
string])
default = ["cat", 1, "dog"]
}
```

Input Variable

```
variable "inputname" {
type = string
description = "Set the name of the VPC"
}
```

Number Variable

```
variable "sshport" {
type = number
default = 22
}
```

List Variable

```
variable "mylist" {
type = list(string)
default = ["Value1", "Value2"]
}
```

Map Variable

```
variable "mymap" {
type = map
default = {
Key1 = "Value1"
Key2 = "Value2"
}
}
```

Object Variable

```
variable "myobject" {
type = object({ name =
string, port = list(number)
})
default = {
name = "TJ"
port = [22, 25, 80]
}
}
```

Resource Configuration & Outputs

AWS VPC Resource

```
resource "aws_vpc" "myvpc" {
cidr_block = "10.0.0.0/16"
tags = {
Name = var.inputname
}
}
```

Output: VPC ID

```
output "vpcid" {
value = aws_vpc.myvpc.id
}
```

Output: Owner ID

```
output "owner" {
value = aws_vpc.myvpc.owner_id
description = "owner of the vpc"
}
```

Key Takeaways

✓ Provider Configuration

AWS provider set to `ap-south-1` region

✓ Variable Type Coverage

String, Number, Boolean, List, Map, Tuple, and Object types demonstrated

✓ Best Practices

Use type constraints, default values, and descriptions for maintainability

✓ Output Values

Expose important resource attributes like VPC ID and Owner ID

Pro Tip: Always use explicit type declarations and descriptions to improve code readability and team collaboration.

💻 Hands-On Classroom Exercise

💻 Configuration File (main.tf)

```
provider "aws" {
  region = "eu-west-1"
}

resource "aws_s3_bucket" "demo_bucket" {
  bucket = "terraform-demo-bucket-${random_id.rand.hex}"
  tags = {
    Purpose = "Training Demo"
  }
}

resource "random_id" "rand" {
  byte_length = 4
}
```

> **Note:** This creates an S3 bucket with a randomly generated name to avoid conflicts

💻 Discussion Questions

After completing the exercise, discuss:

- What files were created during `init`?
- What happened to the `.tfstate` file after `apply`?
- How did the bucket name differ each time (due to `random_id`)?
- What's the effect of `destroy`?

Terraform Core Concepts - Code Example

Basic example in HCL
(HashiCorp Configuration Language)


```
# 1. Provider Configuration
provider "aws" {
  region = "eu-west-1"
}

# 2. Input Variable
variable "instance_type" {
  description = "The EC2 instance type for our banking app server."
  type        = string
  default     = "t3.micro"
}

# 3. A Resource block to define an EC2 instance
resource "aws_instance" "banking_app_server" {
  ami           = "ami-0c55b159cbfafe1f0" # Amazon Linux 2 AMI
  instance_type = var.instance_type      # Using the variable

  tags = {
    Name    = "Banking-App-Server"
    Project = "Personal Banking Management Service"
  }
}

# 4. An Output to display the public IP
output "server_public_ip" {
  value = aws_instance.banking_app_server.public_ip
}
```

Understanding Your First Terraform Script

A Complete Guide for Freshers

Building Cloud Infrastructure with Code

Part 1: The Provider Block

The "Where" - Choosing Your Cloud Platform

```
# 1. Provider Configuration
provider "aws" {
  region = "eu-west-1"
}
```

What it is:

The provider block tells Terraform which cloud platform we will be working with.

> 💡 **Analogy:**
>
> Imagine you want to build a house. The first thing you do is hire a construction company. The provider is your construction company (in this case, AWS - Amazon Web Services).

region = "eu-west-1": This specifies the exact location where you want to build. AWS has data centers all over the world, and eu-west-1 is the code for their Ireland region. This is like choosing the city to build your house in.

Part 2: Input Variable

The "What If?" - Making Code Flexible

```
# 2. Input Variable
variable "instance_type" {
  description = "The EC2 instance type for our banking app server."
  type        = string
  default     = "t3.micro"
}
```

What it is:

A variable makes our code flexible and reusable. Instead of hard-coding a value, we create a placeholder.

> 💡 **Analogy:**
>
> Think of this as a part of your house blueprint that you can easily change. Instead of fixing the window size in the main blueprint, you have a note that says "Window Size: see default". Here, our variable is `instance_type` and its default value is `"t3.micro"` (a small server size).

Why use it?

We can now easily change the server size later without touching the main resource code. This is great for creating different environments (like a small server for testing and a large one for production).

Part 3: The Resource Block (1/2)

The "What" - Defining Your Infrastructure

```
# 3. A Resource block to define an EC2 instance
resource "aws_instance" "banking_app_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type

  tags = {
    Name    = "Banking-App-Server"
    Project = "Personal Banking Management Service"
  }
}
```

What it is:

This is the most important block. It describes the actual piece of infrastructure we want to build.

resource "aws_instance" "banking_app_server"

aws_instance: This is the type of resource we want (an AWS virtual server, also known as an EC2 instance).

banking_app_server: This is our local name for this resource. We use this name to refer to this server in other parts of our code.

Part 3: The Resource Block (2/2)

Understanding Resource Properties

Inside the block:

ami: This is the Amazon Machine Image. It's a template that defines the operating system. Think of it as choosing between Windows, macOS, or Linux for a new laptop.

instance_type: This sets the server's hardware (CPU, RAM). Notice we are using **var.instance_type** to get the value from the variable we defined earlier!

tags: These are simple labels. They help you organize and identify your resources in the AWS console, just like labeling folders in a filing cabinet.

> 🔑 **Key Point:**
>
> By using `var.instance_type` instead of hardcoding "t3.micro", we make our code flexible and maintainable!

Part 4: The Output Block

The "Show Me" - Displaying Important Information

```
# 4. An Output to display the public IP
output "server_public_ip" {
  value = aws_instance.banking_app_server.public_ip
}
```

What it is:

An output block is used to display useful information after your resources are created.

> 💡 **Analogy:**
>
> After the construction company builds your house, they hand you the key and tell you the address. This output is doing the same thing.

value = aws_instance.banking_app_server.public_ip: This line tells Terraform: "After you're done, go to the aws_instance we named banking_app_server and show me its public_ip address." This IP address is what you would use to connect to your new server.

📝 Knowledge Check (1/2)

Test Your Understanding

Question 1: What is the purpose of the provider block in this script?

Answer: It tells Terraform that we are building resources in Amazon Web Services (AWS) and specifically in the eu-west-1 (Ireland) region.

Question 2: Why is it better to use var.instance_type instead of just writing "t3.micro" directly in the resource block?

Answer: Using a variable makes the code more flexible. We can easily change the instance type for different environments without editing the core resource logic.

📝 Knowledge Check (2/2)

Test Your Understanding

Question 3: In the resource block, what do "aws_instance" and "banking_app_server" represent?

Answer: "aws_instance" is the resource type (what we are building), and "banking_app_server" is the local name we use to refer to this specific resource within our Terraform code.

Question 4: What information will we see on our screen after Terraform successfully creates the server?

Answer: We will see the server's public IP address, because of the output "server_public_ip" block.

💻 Hands-On Lab (1/3)

Modify and Redeploy!

Objective:

Get comfortable with the plan/apply cycle by modifying the existing code.

Prerequisites:

Terraform installed on your machine

AWS account credentials configured for Terraform

Setup:

Create a new folder named **terraform-practice**

Inside the folder, create a file named **main.tf**

Copy and paste the entire Terraform code from the slide into main.tf

💻 Hands-On Lab (2/3)

Initialize & Deploy

Steps to Deploy:

Open your terminal in the **terraform-practice** folder

Run the command **terraform init**. This downloads the necessary AWS provider plugin.

Run **terraform plan**. Review the output to see what Terraform intends to create.

Run **terraform apply**. Type **yes** when prompted to create your first EC2 instance.

Note the **server_public_ip** that is displayed at the end.

💻 Hands-On Lab (3/3)

The Challenge - Modify the Code

Your Tasks:

Task 1: In the tags section of the resource block, change the Name from "Banking-App-Server" to "[Your-Name]-Server" (e.g., "Priya-Server")

Task 2: Add a new tag to the resource block: Environment = "Training"

Task 3: Change the default value in the variable block from "t3.micro" to "t2.micro"

See Your Changes:

Save the main.tf file

Run **terraform plan** again. Notice how Terraform shows it will modify the existing resource

Run **terraform apply** and type yes to apply your changes

⚠️ Clean Up! Run `terraform destroy` to avoid AWS charges!

Summary

Key Takeaways

What You've Learned:

Provider: Defines which cloud platform and region to use

Variable: Makes code flexible and reusable across environments

Resource: The core block that defines actual infrastructure components

Output: Displays important information after resource creation

> 🎯 Remember:
>
> Terraform is Infrastructure as Code. You write it once, version control it, and deploy it consistently across environments!

Challenge-01

1. Create one folder with the name Challenge-01
2. Create a VPC named “Terraform_VPC”
3. CIDR Range: 192.168.0.0/24

EC2 Instance

Hands-On Exercise 1: Provision a Simple App Server

Goal

Create a Terraform configuration to launch a basic EC2 instance that could host our banking application.

Scenario

Your development team needs a standard server for testing a new transaction processing microservice.

Ready to Get Started?

Let's build our first Terraform configuration together!

Exercise 1 - Steps & Solution

1. **Create a file:** main.tf
2. **Add Provider Configuration:** Add the AWS provider block and specify a region.
3. **Define a Resource:** Add an aws_instance resource with appropriate AMI and instance type.
4. **Initialize Terraform:** Run `terraform init`
5. **Plan the changes:** Run `terraform plan`
6. **Apply the changes:** Run `terraform apply`
7. **Clean up:** Run `terraform destroy`

Solution Code (main.tf):

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "transaction_service_dev" {
  ami           = "ami-0c55b159cbfafe1f0" # Amazon Linux 2 AMI (us-east-1)
  instance_type = "t2.micro"

  tags = {
    Name = "Transaction-Service-Dev"
  }
}
```

Terraform Workspaces & Backends

Workspaces

Workspaces allow you to use the same configuration to manage multiple distinct sets of infrastructure resources.

Use Case: Managing separate environments like dev, staging, and production for the Personal Banking app without copying code.

Commands:

```
terraform workspace new <name>
terraform workspace select <name>
```

Backends

A backend determines how Terraform loads and stores state. By default, it's a local file (terraform.tfstate).

Why use a Remote Backend?

- Collaboration: Teams can access the same state file
- State Locking: Prevents corruption
- Security: Keeps sensitive info off local machines

Example Backend Configuration (S3):

```
terraform {
  backend "s3" {
    bucket         = "fidelity-personal-banking-tfstate"
    key            = "global/s3/terraform.tfstate"
    region         = "eu-west-1"
    dynamodb_table = "terraform-state-lock" # For state locking
  }
}
```

Managing Secrets in Terraform

Problem:
How do you handle database passwords, API keys, and other secrets for the Personal Banking service without committing them to Git?

Solutions

HashiCorp Vault: The gold standard. Terraform has a Vault provider to read secrets dynamically.

Cloud Provider's Secret Manager:

- AWS Secrets Manager
- Azure Key Vault
- Google Cloud Secret Manager

Pro-Level Tip: Never hardcode secrets!

Example: Using AWS Secrets Manager Data Source

```
# Data source to fetch a secret from AWS Secrets Manager
data "aws_secretsmanager_secret_version" "db_credentials" {
  secret_id = "personal-banking/database/credentials"
}

# Parse the JSON secret string
locals {
  db_creds = jsondecode(data.aws_secretsmanager_secret_version.db_credentials.secret_string)
}

# Use the fetched secret in a resource
resource "aws_db_instance" "banking_db" {
  # ... other configuration
  username = local.db_creds.username
  password = local.db_creds.password
}
```

Fidelity Innersource Repository

What is Innersource?

Applying open-source principles and practices to our internal software development. We collaborate on shared code to build better software, faster.

Walkthrough of Important Modules

For the Personal Banking Management Service, you'll frequently use:

```
/modules
  /vpc
    main.tf
    variables.tf
    outputs.tf
  /rds-postgres
    ...
/examples
  /complete-app-setup
    ...
README.md
CONTRIBUTING.md
```

modules/vpc: Creates a standard, compliant Virtual Private Cloud for our services.

modules/rds-postgres: Provisions a PostgreSQL database with our standard configuration (backups, encryption, etc.).

modules/ecs-service: Deploys a containerized application (Java/Python) as a service on ECS.

modules/iam-role: Creates standardized IAM roles with the principle of least privilege.

Fidelity Innersource Repository

What is Innersource?

Applying open-source principles and practices to our internal software development. We collaborate on shared code to build better software, faster.

Walkthrough of Important Modules

For the Personal Banking Management Service, you'll frequently use:

```
/modules
  /vpc
    main.tf
    variables.tf
    outputs.tf
  /rds-postgres
    ...
/examples
  /complete-app-setup
    ...
README.md
CONTRIBUTING.md
```

modules/vpc: Creates a standard, compliant Virtual Private Cloud for our services.

modules/rds-postgres: Provisions a PostgreSQL database with our standard configuration (backups, encryption, etc.).

modules/ecs-service: Deploys a containerized application (Java/Python) as a service on ECS.

modules/iam-role: Creates standardized IAM roles with the principle of least privilege.

Hands-On Exercise 2: Use an Innersource Module

Goal

Use the rds-postgres innersource module to provision a database for the user profile service.

Scenario

Your team is building a new "User Profile" microservice and needs a standard, secure PostgreSQL database.

Time to Use Shared Modules!

Let's leverage Fidelity's innersource repository

Exercise 2 - Steps & Solution

- **Create a new folder:** for your project (e.g., user-profile-service-db)
- **Create a main.tf file**
- **Reference the module:** Use a module block. The source will point to the innersource Git repository path
- **Provide required variables:** The module's README.md will list required inputs like db_name, instance_class, etc.
- **Initialise, Plan, Apply:** terraform init → terraform plan → terraform apply

Solution Code (main.tf):

```
provider "aws" {
  region = "eu-west-1"
}

# Call the innersource module
module "user_profile_db" {
  # Example source URL. Use the actual Fidelity Git URL.
  source = "git::https://git.fidelity.com/terraform-modules/" +
           "rds-postgres.git?ref=v1.2.0"

  # Pass required variables to the module
  db_name               = "user_profiles"
  engine_version        = "13.4"
  instance_class        = "db.t3.small"
  allocated_storage     = 20
  vpc_security_group_ids = ["sg-012345abcdef"]
  db_subnet_group_name   = "my-db-subnet-group"

  # Example of passing project-specific tags
  tags = {
    Project = "Personal Banking Management Service"
    Service = "User Profile Service"
  }
}

# Output the database endpoint address
output "db_endpoint" {
  value = module.user_profile_db.db_instance_address
}
```

Questions and Answers (1-2)

Q1: What is the purpose of the provider block in this Terraform file?

A: The provider block specifies which cloud platform the resources will be created on (in this case, `aws`) and in which geographical region (`eu-west-1`).

Q2: In your own words, what is a Terraform Module? Why is it useful?

A: A Terraform Module is a reusable package of code, like a template or a function. It's useful because it saves time, reduces errors, enforces best practices, and keeps configurations consistent across different projects

Questions and Answers (3-5)

Q3: What does ?ref=v1.2.0 mean and why is it important?

A: It's called version pinning. It tells Terraform to use a specific version (v1.2.0) of the module. This ensures our infrastructure build is predictable and won't break if a newer, incompatible version is released.

Q4: Which variable would you change to make the database more powerful?

A: You would change the `instance_class`. For example,
from `db.t3.small` to `db.t3.medium` or `db.m5.large`.

Q5: What is the goal of the output block?

A: To display valuable information after infrastructure is created - in this case, the database's endpoint address which applications need to connect to it.

Contributing to the Innersource Repository

Want to fix a bug or add a feature to a module?
Follow the standard contribution model.

1. **Fork:** Create a personal copy (fork) of the central module repository
2. **Clone:** Clone your forked repository to your local machine
3. **Branch:** Create a new feature branch for your changes (e.g., feature/add-read-replica-support)
4. **Commit:** Make your changes, commit them with a clear message
5. **Push:** Push your branch to your forked repository
6. **Pull Request (PR):** Open a PR from your branch to the main branch of the central repository
7. **Code Review:** Your PR will be reviewed by the module maintainers. They may request changes
8. **Merge:** Once approved, your changes are merged!

Best Practice: Always discuss significant changes in an issue before starting work.

Conclusion & Key Takeaways

What We've Learned

- How to define infrastructure as code using Terraform
- The importance of remote state and secrets management for teamwork and security
- How to leverage Fidelity's innersource modules to build faster and more consistently
- The process for contributing back to our shared modules

Key Takeaways

Automate Everything

Use IaC for all infrastructure. No manual changes in the console!

Don't Reinvent

Always check the innersource repository for an existing module before building your own

Security is Paramount

Never hardcode secrets. Use Vault or AWS Secrets Manager

Collaborate

Treat infrastructure code like application code. Use PRs and code reviews to maintain quality

Pro-Level Tips

Use terraform fmt: Automatically formats your code to the standard style. Run it before committing.

Use terraform validate: Checks your syntax before you run a plan or apply.

Keep Modules Focused: A good module does one thing well (e.g., creates a database). Avoid monolithic modules.

Understand count and for_each: Learn these meta-arguments to dynamically create multiple resources from a list or map. This is powerful for creating similar resources (e.g., multiple IAM users).

Most Important Tip:

Read the Plan: Always carefully read the terraform plan output before applying. It is your best defense against accidental, destructive changes.

Challenge-02

1. Create a DB server and output the private IP
2. Create a web server and ensure it has a fixed public IP
3. Create a security Group for the web server, opening ports 80 and 443 (HTTP, HTTPS)
4. Run the provided script on the web server

Challenge-03

Make your code as modular as possible

1. Create a DB server and output the private IP
2. Create a web server and ensure it has a fixed public IP
3. Create a security Group for the web server, opening ports 80 and 443 (HTTP, HTTPS)
4. Run the provided script on the web server

Q&A

Thank You!

Questions & Discussion

Let's discuss your Terraform implementation questions

Fidelity International Limited - Infrastructure as Code Excellence