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.

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
}
```

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
}
```

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.

Q&A

Thank You!

Questions & Discussion

Let's discuss your Terraform implementation questions

Fidelity International Limited - Infrastructure as Code Excellence