Infrastructure as Code

Terraform & Azure

The Problem with "ClickOps"

You've Been Clicking Around the Portal...

What happens when you need to:

- Recreate your environment?
- Deploy to multiple regions?
- Set up dev, test, and prod environments?
- Document what you built?
- Audit history?

Manual approaches lead to:

- Inconsistency between environments
- "Configuration drift" over time
- No version history
- Human errors during deployment
- Knowledge trapped in someone's head

What is Infrastructure as Code?

> *Managing and provisioning infrastructure through **code** instead of manual processes*

- Write configuration files describing your infrastructure
- Store them in version control (Git)
- Run tools to create/update infrastructure automatically
- Track changes just like application code

Benefits:

- ✅ Repeatable and consistent
- ✅ Version controlled
- ✅ Documented by default
- ✅ Testable
- ✅ Fast to deploy
- ✅ Easy to destroy and recreate

Infrastructure as Code Tools

Offerings

Tool	Type	Best For
Terraform	Declarative, Multi-cloud	Cloud-agnostic infrastructure
Azure ARM Templates	Declarative, Azure-only	Azure-native deployments
Bicep	Declarative, Azure-only	Modern Azure IaC
Pulumi	Imperative, Multi-cloud	Using programming languages
CloudFormation	Declarative, AWS-only	AWS infrastructure

Why Terraform?

The Industry Standard

- **Multi-cloud:** Works with Azure, AWS, GCP, and 1000+ providers
- **Declarative:** Describe what you want, not how to build it
- **Large community:** Extensive documentation and examples
- **Mature:** Battle-tested in production worldwide
- **Plan before apply:** See changes before they happen
- **State management:** Knows what currently exists

How Terraform Works

```
1. Write configuration (.tf files)
   ↓
2. terraform init (download providers)
   ↓
3. terraform plan (preview changes)
   ↓
4. terraform apply (create/update resources)
   ↓
5. State file updated (tracks reality)
```

Terraform File Structure

Typical project layout:

```
my-infrastructure/
├── main.tf           # Main resources
├── variables.tf      # Input variables
├── outputs.tf        # Output values
├── providers.tf      # Provider configuration
├── env.tfvars  # Variable values
└── modules/          # Reusable modules
    └── networking/
        ├── main.tf
        └── variables.tf
```

Azure Provider Setup

Connecting Terraform to Azure

The Azure Provider enables Terraform to manage Azure resources

- **Subscription ID:** Which Azure subscription
- **Tenant ID:** Your Azure AD tenant
- **Authentication:** How Terraform logs in

Variables and Outputs

Making Configuration Flexible

Variables = Inputs

- Parameterize your infrastructure
- Different values for dev/test/prod
- Avoid hardcoding values
- Can have defaults and validation

Outputs = Results

- Export important values
- Pass data between modules
- Display information after deployment
- Use in other tools

> *Variables go in, outputs come out!*

Terraform Modules

Why use modules?

- ✅ Reusability across projects
- ✅ Consistency and standards
- ✅ Abstraction of complexity
- ✅ Easier testing
- ✅ Team collaboration

Module Best Practices

Good modules are:

1. **Single-purpose:** Do one thing well
2. **Well-documented:** Clear README and variable descriptions
3. **Tested:** Verified to work correctly
4. **Versioned:** Use version tags in Git
5. **Flexible:** Configurable via variables
6. **Opinionated:** Encode best practices

Module Sources

Where to get modules:

Public Registry:

- registry.terraform.io
- Thousands of modules
- Official Azure modules from Microsoft
- Community contributions

Private Registry:

- Terraform Cloud
- Your company's internal modules (or one you've written!)

Git repositories:

- GitHub, GitLab, Bitbucket
- Version control with tags
- Good for custom modules

State File Management

💾 Terraform's Memory

State file (`terraform.tfstate`) tracks:

- What resources exist
- Current configuration
- Resource metadata
- Dependencies between resources

Critical importance:

- Terraform compares desired state (your .tf files) vs actual state (state file)
- Determines what changes are needed
- **Losing state file = disaster!**

State File Challenges

Local state problems:

- ❌ Not shared between team members
- ❌ No locking (concurrent changes = corruption)
- ❌ **Contains sensitive data!**
- ❌ No versioning or backup
- ❌ Hard to collaborate

For production: NEVER use local state!

Remote State Backends

Store state remotely for team collaboration:

Azure Storage (Recommended for Azure):

- Blob storage container
- Built-in encryption
- Access controls via Azure AD
- Automatic blob versioning
- State locking via blob lease

Other backends:

- Terraform Cloud (managed service)
- AWS S3
- Google Cloud Storage
- Consul, etcd (for advanced setups)

🔒 State Locking

Prevents you and I from destroying each others work

Without locking:

- Two people run `terraform apply` simultaneously
- State file gets corrupted
- Infrastructure becomes inconsistent

With locking:

- First person acquires lock
- Second person waits or gets error
- Safe sequential execution

Azure Storage provides automatic locking!

State Best Practices

1. **Always use remote state** for team projects
2. **Enable state locking**
3. **Enable versioning** on storage backend
4. **Encrypt state** at rest and in transit
5. **Restrict access** to state files (contain secrets!)
6. **Backup state** regularly
7. **Use separate states** for different environments

The Development Cycle

Step 1: Write

- Create or modify .tf files
- Define resources and configuration
- Use AI assistance for syntax and best practices

Step 2: Style!

- Terraform format

Step 3: Initialize

- Run `terraform init`
- Downloads providers and modules
- Configures backend

Step 4: Plan

- Run `terraform plan`
- See what will change
- Review before applying

Step 5: Apply

- Run `terraform apply`
- Confirm changes
- Terraform creates/updates resources
- State file updated

Step 6: Verify

- Check Azure portal
- Test deployed resources
- Review outputs

Useful Terraform Commands

Command	Purpose
`terraform init`	Initialize working directory
`terraform plan`	Preview changes
`terraform apply`	Create/update infrastructure
`terraform destroy`	Delete all resources
`terraform fmt`	Format code nicely
`terraform validate`	Check syntax
`terraform output`	Show output values
`terraform state list`	List resources in state

Resource Dependencies

Managing Relationships

Terraform automatically handles dependencies:

Implicit dependencies:

- Terraform detects when one resource references another
- Creates resources in the correct order
- Example: Container App needs Container App Environment

Explicit dependencies:

- Use `depends_on` when implicit isn't enough
- Forces creation order
- Useful for non-obvious dependencies

Dependency graph:

- Terraform builds internal graph
- Parallel creation when possible
- Sequential when required

Version Control for IaC

Repository structure:

```
infrastructure/
├── .gitignore           # Ignore state and secrets
├── README.md            # Documentation
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   └── terraform.tfvars
│   ├── test/
│   └── prod/
└── components/
    └── container-app/
```

.gitignore must include:

- `*.tfstate`
- `*.tfstate.backup`
- `.terraform/`
- `terraform.tfvars` (if contains secrets)

Importing Existing Resources

What if you already created resources manually?

Terraform import:

- Brings existing Azure resources into Terraform state
- You write the Terraform configuration
- Import command links it to existing resource
- Future changes managed by Terraform

Process:

1. Write Terraform config matching existing resource
2. Run `terraform import` with resource ID
3. Run `terraform plan` (should show no changes)
4. Now managed by Terraform!

AI can help: Generate Terraform from Azure resource properties!

Terraform Best Practices Summary

Code organization:

- Use modules for reusability
- Meaningful resource and variable names
- Consistent naming conventions
- Document with comments

State management:

- Always use remote backend for teams
- Enable state locking
- Version and backup state
- Separate states for environments

Security:

- Never commit secrets
- Use managed identities
- Encrypt state files
- Review all changes

Workflow:

- Always plan before apply
- Use version control
- Code review via pull requests
- Automate with CI/CD