Azure Provider

Terraform

Agenda

- Azure Authentication
- Import existing resources

Azure Authentication

Terraform

Azure Authentication

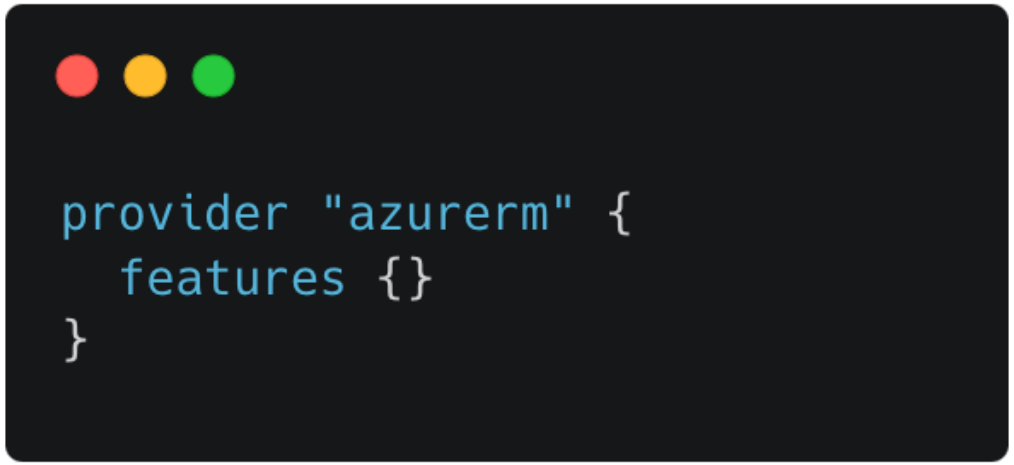
- Azure CLI Authentication
- Service Principal Authentication
- Managed Identity Authentication
- Workload Identity Federation

Azure CLI Authentication

- Terraform uses the credentials stored locally after running the **az login** command
- Use Case
  - Ideal for development or when running Terraform locally with access to the Azure CLI
- Pros
  - No need for explicit configuration in Terraform
  - Easy setup during local development
- Cons
  - Not suitable for automation (e.g., CI/CD pipelines)

Azure CLI Authentication

- Provider Configuration



Service Principal Authentication

- Use a Service Principal (SP) with a client ID, secret, and tenant ID for authentication.
- Use Case
  - Recommended for automation in CI/CD pipelines or infrastructure where local authentication isn't practical.
- Pros
  - Secure and automated
  - Granular control over permissions using Azure RBAC
- Cons
  - Requires explicit storage and management of SP credentials.

Service Principal Authentication

- Provider Configuration

```
provider "azurerm" {
  features {}

  client_id       = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
  client_secret   = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
  tenant_id       = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
}
```

Managed Identity Authentication

- Use Azure Managed Identity (System-Assigned or User-Assigned) to authenticate
- Use Case
  - Best for deployments running in Azure services (e.g., Azure VMs, AKS) without managing secrets.
- Pros
  - No credentials to manage
  - Seamless integration with Azure-hosted resources
- Cons
  - Only works within Azure-hosted environments (agent/runner running in Azure)

Managed Identity Authentication

- Provider Configuration

```
provider "azurerm" {
  features {}

  use_msi         = true
  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
}
```

Workload Identity Federation

- Authenticate with Azure Active Directory using external identity providers (e.g., GitHub Actions, AWS STS) without needing secrets.
- Use Case
  - Ideal for modern CI/CD systems leveraging external identity providers
- Pros
  - Secretless authentication.
  - Integrates seamlessly with identity providers like GitHub Actions or AWS
- Cons
  - Requires additional setup in Azure AD

Workload Identity Federation

- Configure Workload Identity Federation in Azure AD

```
az identity federated-credential create \
  --name "github-actions" \
  --identity-name "example-identity" \
  --resource-group "example-resource-group" \
  --issuer "https://token.actions.githubusercontent.com" \
  --subject "repo:<OWNER>/<REPO>:environment:<ENVIRONMENT>"
```

Workload Identity Federation

- Provider Configuration

```
provider "azurerm" {
  features {}

  use_oidc_auth = true
  subscription_id = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
}
```

Environment Variables

- On CI/CD pipelines using of environment variables can be a suitable way to inject privately these details
- On Terraform you can set all these needed values using environment variables on the format **ARM_VARIABLE**

```
export ARM_CLIENT_ID="xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
export ARM_CLIENT_SECRET="xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
export ARM_SUBSCRIPTION_ID="xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
export ARM_TENANT_ID="xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
```

Import existing resources

Terraform

Terraform and existing infrastructure

- When you start to work on a greenfield for Terraform, everything follows the flow we view before
- But on a brownfield?
- We need to have a tfstate updated and synced with real infrastructure
- Needed steps: authoring or generate Terraform code and populate tfstate

Generate Terraform Code

- For Azure, two tools are recommended to generate code:
  - Azure Terraform Resource Provider
  - Azure Export for Terraform
- But you need to use it carefully because this tooling can generate a massive amount of code
- When you create the code by yourself, you only set the properties that you want to set, everything else you rely on default values
- When you generate code by tooling you may get all properties to be set on the code
- On complex resources, like AKS, can generate big amount of code

Azure Terraform Resource Provider

- Native feature on Azure to export resources to Terraform (like before for ARM)
- This feature is in Public Preview and need to be explicitly enabled on your subscriptions
- Rely on Azure Export for Terraform

Azure Export for Terraform

- Azure Export for Terraform is a tool designed to help reduce friction in translation between Azure and Terraform concepts.
- Simplify migration to Terraform on Azure allowing you to migrate Azure resources to Terraform using a single command
- Export user-specified sets of resources to Terraform HCL code and state with a single command
- Azure Export for Terraform enables you to specify a predetermined scope to export
- The scope can be as granular as a single resource. You can also export a resource group and its nested resources. Finally, you can export an entire subscription.

Azure Export for Terraform

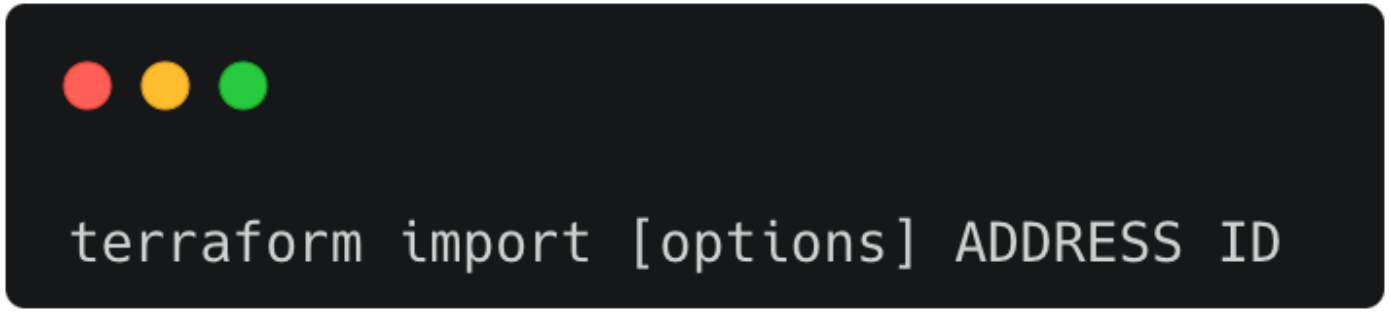
- Inspect preexisting infrastructure with all exposed properties
- Whether learning a newly released resource or investigating an issue in production, Azure Export for Terraform supports a read-only export with the option to expose all configurable resource properties
- Follow plan/apply workflow to integrate non-Terraform infrastructure into Terraform
- Export HCL code, inspect non-Terraform resources and easily integrate them into your production infrastructure and remote backends.

Authoring vs Generate

- For Azure, two tools are recommended to generate code:
  - Azure Terraform Resource Provider
  - Azure Export for Terraform
- But you need to use it carefully because this tooling can generate a massive amount of code
- When you create the code by yourself, you only set the properties that you want to set, everything else you rely on default values
- When you generate code by tooling you may get all properties to be set on the code
- On complex resources, like AKS, can generate big amount of code

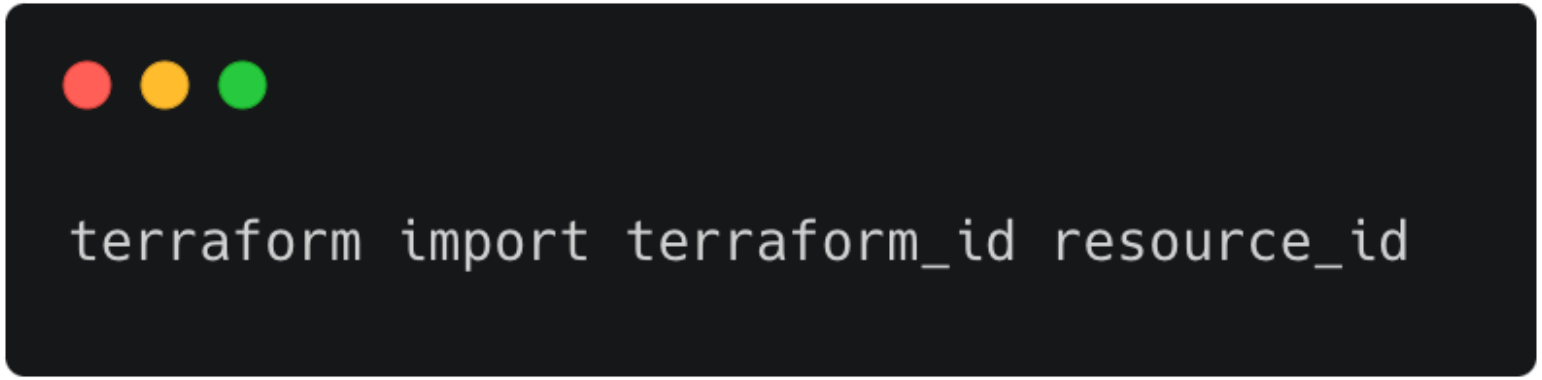
terraform import

- This command allow you to import existing infrastructure resources
- This functionality lets you bring existing resources under Terraform management



terraform import

- Import will find the existing resource from **resource_id** and import it into your Terraform state at the given **terraform_id**
- **terraform_id** must be a valid resource address
- **resource_id** is dependent on the resource type being imported



Demo – Import resources

Terraform

Lab 03 – Store your state in Azure

Terraform