Using Resources

Terraform

Agenda

- Data Sources
- Explicit dependencies
- Dynamic Blocks
- Resource Lifecycle

Data Sources

Terraform

Data Sources

- Data sources in Terraform allow you to query existing resources or external information and use it in your configuration
- They are particularly useful when you want to reference existing resources instead of creating new ones or fetch dynamic information from providers (e.g., AMI IDs, VPC IDs, or environment-specific values).

Data Sources: Why?

- **Reuse Existing Resources**: Avoid duplicating infrastructure already managed outside your Terraform configuration
- **Dynamic Updates**: Always fetch the latest state of resources, such as the newest AMI or updated VPC information.
- **Environment-Specific Configuration**: Query resources based on environment-specific tags or attributes.
- **Integration with External Systems**: Data sources enable integration with systems like AWS, Terraform Cloud, external files, or third-party APIs.

Data Sources: Syntax

- **provider_resource_type**: The resource type you are querying.
- **name**: A unique name to reference the data source.
- Arguments: Used to filter or query specific data.

```
data "provider_resource_type" "name" {
  # Arguments to query the resource
}
```

Data Sources: Syntax

- To reference you use the same concept as resources

```
data.provider_resource_type.name
```

Data Sources: Example

```
# Data Source: Query Existing Resource Group
data "azurerm_resource_group" "example" {
  name = "example-resource-group"
}

# Resource: Create a Storage Account Using the Resource Group Tags
resource "azurerm_storage_account" "example" {
  name                     = "examplestorageacct"
  resource_group_name      = data.azurerm_resource_group.example.name
  location                 = data.azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
  tags                     = data.azurerm_resource_group.example.tags
}
```

Data Sources: Key Points to Remember

- Data sources only fetch data; they do not create or manage resources
- Combine data sources with resources or outputs for dynamic and reusable configurations
- Data sources execute queries during the planning phase (**terraform plan**).

Data Sources: Best Practices

- Use filters to precisely define the data you need (e.g., by tags, names, or other identifiers)
- Limit usage of data sources to avoid fetching unnecessary data, which can slow down execution
- Document data source usage to ensure clarity for future updates

Explicit Dependencies

Terraform

Auto dependencies

- Terraform always try to generate automatically a dependency graph between all resources
- This graph is generated based on the relationship between the resources on your code and not based on the provider
- For example, Terraform Azure Provider doesn't make an explicit relationship between an Azure App Service Plan and an App Services. That relationship happens because you refer the first in the definition of the latter

Explicit dependencies

- But sometimes you need to handle hidden resource dependencies that Terraform cannot automatically infer
- You only need to explicitly specify a dependency when a resource or module relies on another resource's behavior but does not access any of that resource's data in its arguments
- For that you use the meta-argument **depends_on**

Explicit dependencies: **depends_on**

- The **depends_on** meta-argument instructs Terraform to complete all actions on the dependency object before performing actions on the object declaring the dependency
- You should use **depends_on** as a last resort because it can cause Terraform to create more conservative plans that replace more resources than necessary
- For example, Terraform may treat more values as unknown “(known after apply)” because it is uncertain what changes will occur on the upstream object

depends_on
example

```
resource "aws_iam_role_policy" "example" {
  name   = "example"
  role   = aws_iam_role.example.name
  policy = jsonencode({
    "Statement" = [{
      "Action" = "s3:*",
      "Effect" = "Allow",
    }],
  })
}

resource "aws_instance" "example" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  depends_on = [
    aws_iam_role_policy.example
  ]
}
```

Dynamic Blocks

Terraform

dynamic Blocks

- Within top-level block constructs like resources, expressions can usually be used only when assigning a value to an argument using the **name = expression** form
- This covers many uses, but some resource types include repeatable nested blocks in their arguments, which typically represent separate objects that are related to (or embedded within) the containing object
- You can dynamically construct repeatable nested blocks like setting using a special **dynamic** block type, which is supported inside **resource** or **data**

dynamic Blocks

```
resource "aws_elastic_beanstalk_environment" "tfenvtest" {
  name = "tf-test-name" # can use expressions here

  setting {
    # but the "setting" block is always a literal block
  }
}
```

dynamic Blocks

```
resource "aws_elastic_beanstalk_environment" "tfenvtest" {
  name = "tf-test-name"

  dynamic "setting" {
    for_each = var.settings
    content {
      namespace = setting.value["namespace"]
      name      = setting.value["name"]
      value     = setting.value["value"]
    }
  }
}
```

Properties

- The label of the dynamic block ("**setting**" in the example above) specifies what kind of nested block to generate
- The **for_each** argument provides the complex value to iterate over
- The **iterator** argument (optional) sets the name of a temporary variable that represents the current element of the complex value. If omitted, the name of the variable defaults to the label of the dynamic block ("**setting**" in the example above).

Benefits

- **Flexibility**: Allows you to create nested blocks dynamically based on a list, map, or other input data.
- **Reusability**: Simplifies configurations by avoiding hardcoded repetition.
- **Conditionals**: Can include or exclude blocks based on conditions.

Resource Lifecycle

Terraform

lifecycle block

- When you define a resource block, Terraform handles automatically its lifecycle
- But in some situations, you need to handle manually specific parts of the resource lifecycle

```
resource "azurerm_resource_group" "example" {
  # ...

  lifecycle {
    create_before_destroy = true
  }
}
```

lifecycle block: **create_before_destroy**

- By default, when Terraform must change a resource argument that cannot be updated in-place due to remote API limitations
- Terraform will instead destroy the existing object and then create a new replacement object with the new configured arguments
- The **create_before_destroy** meta-argument changes this behavior so that the new replacement object is created first, and the prior object is destroyed after the replacement is created

lifecycle block: **prevent_destroy**

- This meta-argument, when set to **true**, will cause Terraform to reject with an error any plan that would destroy the infrastructure object associated with the resource, as long as the argument remains present in the configuration
- This can be used as a measure of safety against the accidental replacement of objects that may be costly to reproduce, such as database instances
- However, it will make certain configuration changes impossible to apply, and will prevent the use of the terraform destroy command once such objects are created, and so this option should be used sparingly.

lifecycle block: **ignore_changes**

- By default, Terraform detects any difference in the current settings of a real infrastructure object and plans to update the remote object to match configuration
- The **ignore_changes** feature is intended to be used when a resource is created with references to data that may change in the future, but should not affect said resource after its creation
- In some rare cases, settings of a remote object are modified by processes outside of Terraform, which Terraform would then attempt to "fix" on the next run
- In order to make Terraform share management responsibilities of a single object with a separate process, the **ignore_changes** meta-argument specifies resource attributes that Terraform should ignore when planning updates to the associated remote object.

lifecycle block: ignore_changes

```
resource "aws_instance" "example" {
  # ...

  lifecycle {
    ignore_changes = [
      # Ignore changes to tags,
      tags,
    ]
  }
}
```