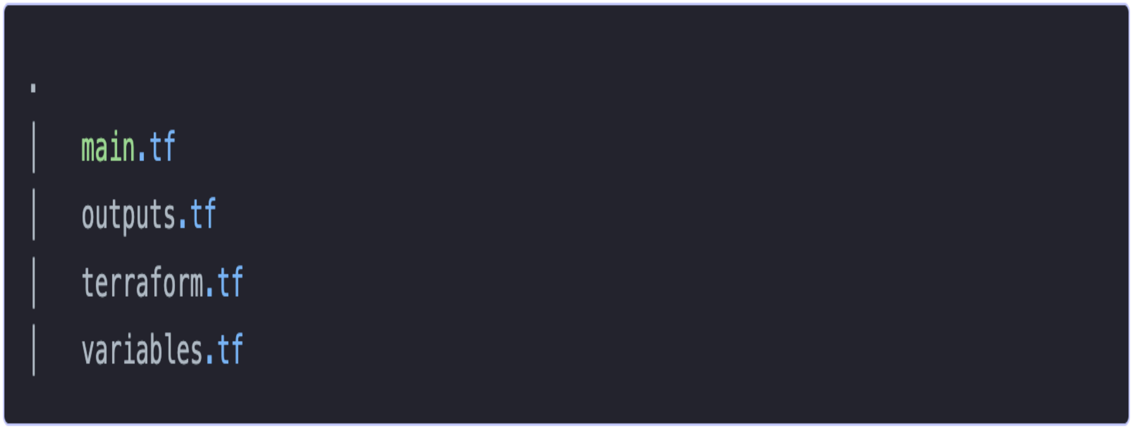
Terraform Modules Guide: Best Practices & Examples

When you start using HashiCorp Terraform to set up your infrastructure, you'll notice that some resources or setups are repeated in different places. Instead of creating the same setup again and again, Terraform lets you group these resources into reusable units called **modules**. Think of a module like a template that bundles resources together, making it easy to use them in multiple configurations without starting from scratch. This saves time and keeps your code organized and efficient.

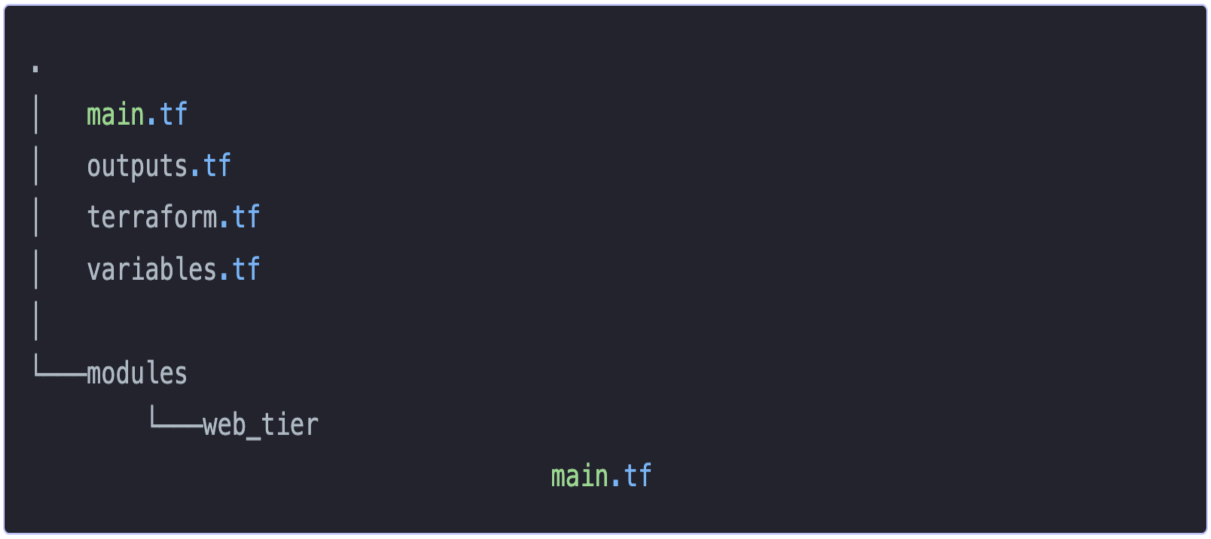
1. What are Terraform Modules?

A Terraform module is simply a group of resources defined in Terraform files (`.tf` or `.tf.json`) stored in the same folder. In fact, every time you've used Terraform, you've already been working with a module! If your project has a folder with Terraform configuration files, that's essentially a module.



```
.
│   main.tf
│   outputs.tf
│   terraform.tf
│   variables.tf
```

That configuration is called the **root module**. It usually includes things like resources, data sources, input variables, and outputs. The root module can also include **child modules**, which are like smaller, reusable modules. These child modules can also have the same components—resources, data sources, variables, and outputs—just like the root module.



```
.
│   main.tf
│   outputs.tf
│   terraform.tf
│   variables.tf
│
└───modules
        └───web_tier
                                main.tf
```

You might wonder what Terraform modules are commonly used for. Well, they can handle almost anything! Here are some examples:

- Setting up a complete AWS VPC with subnets, route tables, and an internet gateway.
- Creating a Microsoft SQL Always On cluster in Azure with Network Security Groups.
- Configuring a GCP project with the required APIs and permissions.

Once you notice repeated patterns in your setups, it's easy to see which parts can become reusable modules. But modules are more than just saving time—they help keep your setups organized and consistent.

2. Why use Terraform Modules?

There are three main reasons to use a Terraform module:

1. **Reusable Configurations**: Group resources into a single setup you can use again and again.
2. **Standardization**: Share consistent, pre-approved configurations with your team for common deployments.
3. **Avoid Repetition (DRY)**: Save time and reduce errors by not rewriting the same setup multiple times.

We'll dive deeper into these later, once you know how to build and use modules effectively.

3. Using Terraform Modules

Before you start creating your own modules, it's helpful to understand how **child modules** are used within a **root module** configuration. It's also important to know where you can find existing modules that you can use in your projects. Let's take a closer look at these points before diving into creating your own modules.

Module Sources

Terraform modules can be saved in different locations: on your local computer, in a Terraform registry, or in a source control system like GitHub. When you're just starting, you'll most likely store your modules in a subfolder of your root module, as shown in the example below. This keeps things simple and easy to manage while you're learning.

```
.
│   main.tf
│   outputs.tf
│   terraform.tf
│   variables.tf
│
└───modules
        └───web_tier
                                main.tf
```

Terraform only processes files in the current working directory, so files in subdirectories won't be included. Storing modules in a subdirectory is a good starting point because it lets you reuse common components, create multiple instances of a module, or copy it to other projects. However, storing modules locally makes sharing and collaboration with your team harder, and it's not easy for others to discover your modules.

To solve this, Terraform has a **public registry** managed by HashiCorp. It's free and contains modules, providers, and other resources. Anyone can publish modules for others to use. Organizations can also use **private registries** (e.g., with env0) to keep modules secure while sharing them internally.

Terraform registries make it easy to version, search, and document modules. Each module links to a source control repository that holds its code.

Alternatively, you can store modules directly in a source control repository like GitHub. While this provides a shared location, it lacks the versioning and discovery features of a registry.

Module Usage

Adding a module to your Terraform configuration is straightforward. You just need to define a `module` block and specify the **source location** of the module you want to use. This could be a local path, a registry, or a repository.

```
module "web_app" {
  source = "./modules/web_tier/"


}
```

After adding a `module` block, run `terraform init` to load the module into your configuration. For remote modules, Terraform downloads the module's contents into the `.terraform/modules` folder. For local modules, Terraform just references their location in the `.terraform/modules/modules.json` file.

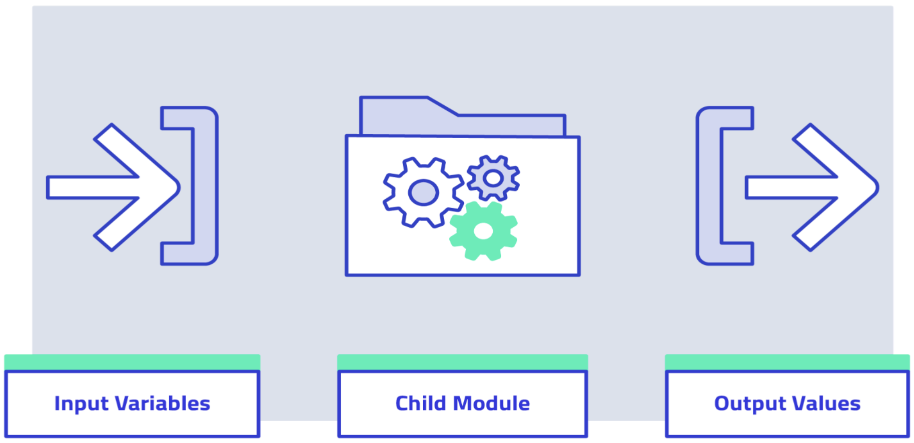
You can pass data to the module by providing arguments that match the **input variables** defined in the module. This allows you to customize the module's behavior and configuration.

```
module "web_app" {
  source = "./modules/web_tier/"
  name = "web-app-a"
  size = "medium"
  min_count = 2
}
```

The outputs defined in a module can be accessed using standard Terraform addressing syntax. For example, if the `web_tier` module has an output called `app_dns_fqdn`, you can reference it in your root module like this:

```
locals {
    web_app_address = module.web_app.app_dns_address
}
```

Objects defined within a module cannot be accessed directly by the root module, and child modules cannot directly reference objects in the root module. The only way they interact is through the **input variables** and **output values** defined in the child module. This acts like an **API or function library**, creating a clear contract for how the root module and child module communicate. This separation ensures modularity and keeps configurations clean and reusable.



Terraform Modules Diagram

By limiting access to objects this way, the root module doesn't need to worry about how the child module works internally. As long as the child module accepts the defined inputs and delivers the expected outputs, the root module remains unaffected by the implementation details. This allows the child module to evolve or improve without impacting the root module, as long as it follows the same input-output "contract." If a newer version of the module introduces breaking changes, Terraform provides ways to manage versioning and updates to handle such changes gracefully.

Versioning Modules

In the `module` block, you can use the `version` argument to specify which version of a module the root module should use. This works only with modules stored in a Terraform registry, not those from local files or Git repositories. For example, to use version `3.19.0` of the AWS VPC module from the public registry, you can specify:

```
module "primary_vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.19.0"
}
```

When you run `terraform init`, Terraform will download version `3.19.0` to the `.terraform/modules` directory. You can also define version ranges, and Terraform will select the newest version within the specified range.

Using version constraints is highly recommended. It prevents unexpected breaking changes from newer versions and lets you control upgrades after testing them. This is one of the key benefits of using a Terraform registry.

4. Creating Terraform Modules

As mentioned earlier, every Terraform project you've worked on has been a root module. While creating a child module shares many similarities with a root module, there are some unique considerations and best practices to keep in mind. These practices ensure that the child module is reusable, modular, and easy to maintain. Let's explore these further.

Module Structure

When creating a Terraform module, it will consist of one or more Terraform configuration files. Here's a common layout for organizing your module:

1. `variables.tf`: Contains all the input variables for the module.
   - These variables allow users of the module to pass in specific values to customize how the resources are created.

2. `versions.tf`: Specifies the required version of Terraform and the necessary providers.
   - This ensures compatibility between the module and the Terraform version or provider versions used.

3. `main.tf`: The core configuration file where the resources and data sources are defined.
   - This is where you describe the infrastructure the module will create, like EC2 instances, VPCs, subnets, etc.

4. `outputs.tf`: Defines all the output values for the module.
   - Outputs are used to return useful information about the resources created, such as an IP address or resource ID, back to the root module.

README.md

It's a good practice to include a `README.md` file that explains the module's purpose and how to use it. The README typically contains:

- The purpose of the module (e.g., what it does).
- A list of input variables and output values.
- Details about the resources the module creates.
- Examples of how to use the module in a root module.

When you look at modules in the public Terraform registry, you might see additional files and folders for advanced use cases (e.g., examples, documentation, or CI/CD workflows). However, for now, it's best to focus on the basics: **inputs, outputs, and resources**.

Example Module: AWS Linux Virtual Machine

Let's create a module that deploys a **Linux virtual machine** (VM) on AWS with a public IP address. This setup is common and involves three core resources:

1. **`aws_instance`**: The virtual machine itself.
2. **`aws_vpc`**: The Virtual Private Cloud (VPC) to define the network.
3. **`aws_subnet`**: A subnet within the VPC to place the VM

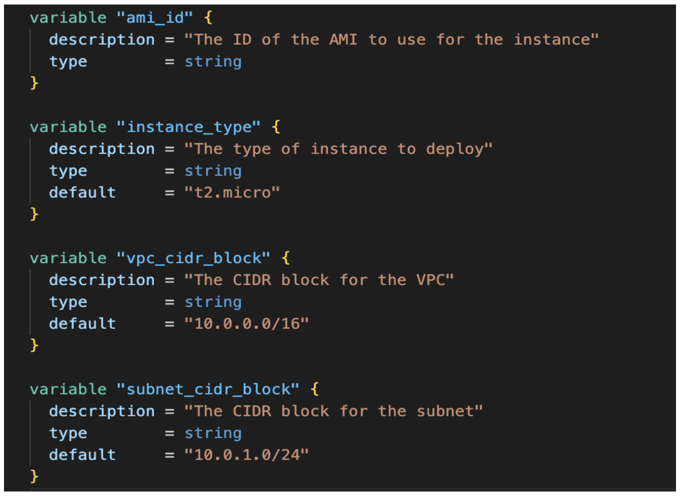
Why Use Variables?

Each resource (like `aws_instance`, `aws_vpc`, and `aws_subnet`) has arguments that require values, such as the AMI ID for the VM, the CIDR block for the VPC, and so on. Using **input variables** allows us to make the module reusable by letting users pass in their own values instead of hardcoding everything.

Step-by-Step Breakdown of the Module

● variables.tf

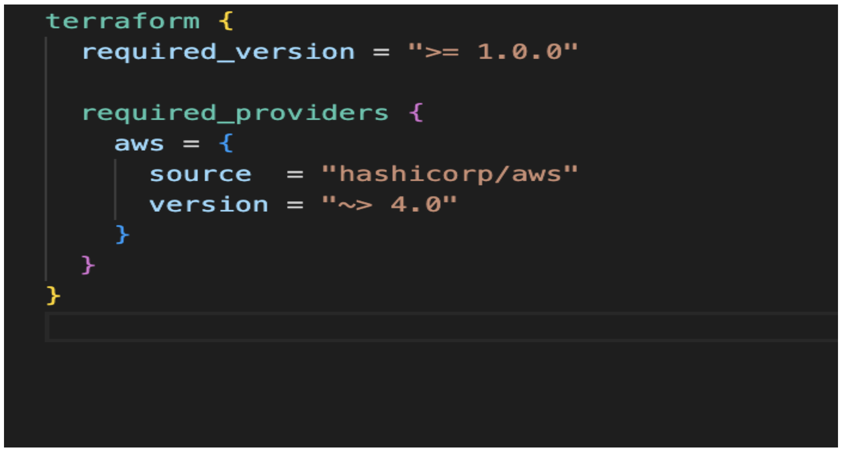
Define the input variables for the module, allowing users to customize the deployment:



```
variable "ami_id" {
  description = "The ID of the AMI to use for the instance"
  type        = string
}

variable "instance_type" {
  description = "The type of instance to deploy"
  type        = string
  default     = "t2.micro"
}

variable "vpc_cidr_block" {
  description = "The CIDR block for the VPC"
  type        = string
  default     = "10.0.0.0/16"
}

variable "subnet_cidr_block" {
  description = "The CIDR block for the subnet"
  type        = string
  default     = "10.0.1.0/24"
}
```

- **versions.tf**

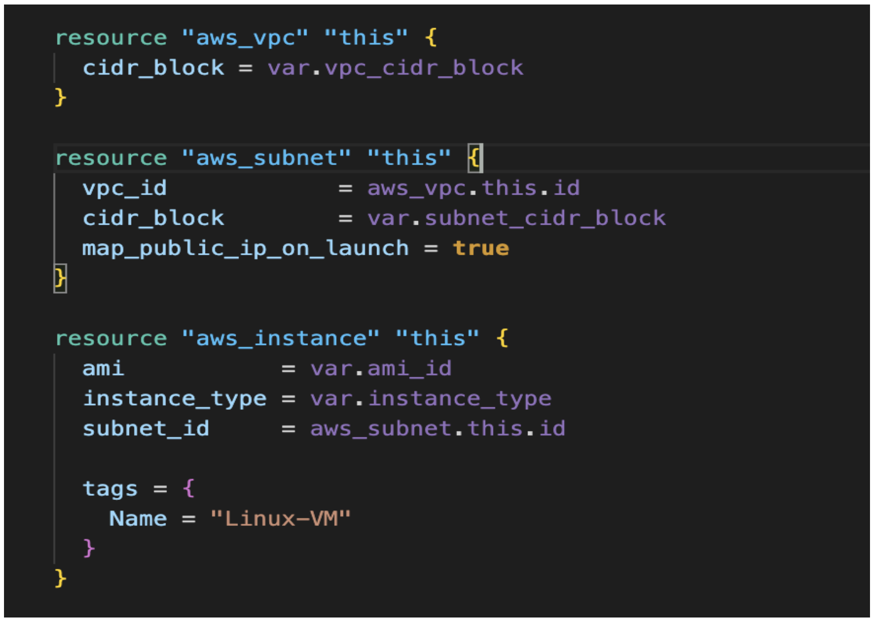
Specify the required versions for Terraform and the AWS provider to ensure compatibility:



```
terraform {
  required_version = ">= 1.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

- **main.tf**

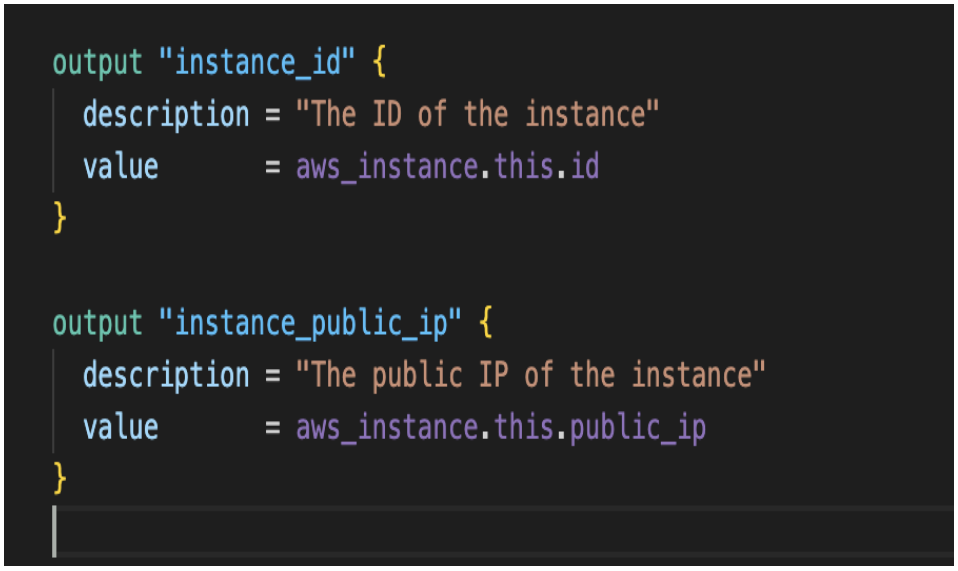
Define the core resources for the module:



```
resource "aws_vpc" "this" {
  cidr_block = var.vpc_cidr_block
}

resource "aws_subnet" "this" {
  vpc_id                  = aws_vpc.this.id
  cidr_block              = var.subnet_cidr_block
  map_public_ip_on_launch = true
}

resource "aws_instance" "this" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = aws_subnet.this.id

  tags = {
    Name = "Linux-VM"
  }
}
```

- **outputs.tf**

Define outputs to return useful information to the root module:



```
output "instance_id" {
  description = "The ID of the instance"
  value       = aws_instance.this.id
}

output "instance_public_ip" {
  description = "The public IP of the instance"
  value       = aws_instance.this.public_ip
}
```

How It Works

1. **Users Provide Inputs**:
   - Users pass their own values for variables like the AMI ID or VPC CIDR block when calling this module.
2. **Resources Are Created**:
   - Terraform uses these inputs to configure the VPC, subnet, and instance as defined in `main.tf`.
3. **Outputs Are Returned**:
   - After deployment, outputs like the instance ID and public IP are passed back to the root module for further use.

Why Use This Approach?

- **Reusability**: The module can be reused across different projects by changing the input values.

- **Scalability**: You can modify or expand the module to add more features without affecting users who rely on it.

- **Clarity**: By separating inputs, resources, and outputs, the module is easy to understand and maintain.

This approach ensures your module is well-structured, reusable, and easy to integrate into any Terraform configuration.

Resource Configuration

Now let's focus on the most important part of the module: the **resources** we want to deploy in our target environment. Resources and their attributes are defined **within the scope of the child module**, meaning they are accessible only inside the module they are defined in.

Key Rules for Using Resources in Modules

1. **Local Scope**:
    - You can only use objects (like variables, data sources, or other resources) that are **defined within the same module**.
    - You cannot reference values or objects defined in the parent (root) module directly. For example, a local value in the parent module cannot be accessed by the child module.
2. **Sources for Resource Arguments**:
    - **Input Variables**: Users of the module pass these values into the child module.
    - **Data Sources**: External data, like AWS AMIs, fetched dynamically.
    - **Other Resources in the Module**: Outputs or attributes from other resources defined in the same module.
3. **Main File**:
    - Resources are typically defined in a single `main.tf` file within the module.
    - Larger modules can break resources into multiple files for better readability (e.g., `network.tf`, `compute.tf`).

Designing Resource Arguments

When defining resource arguments, we often hardcode some values for consistency and reliability. However, you can also create **input variables** to allow users flexibility in customizing the module.

Example AWS Module

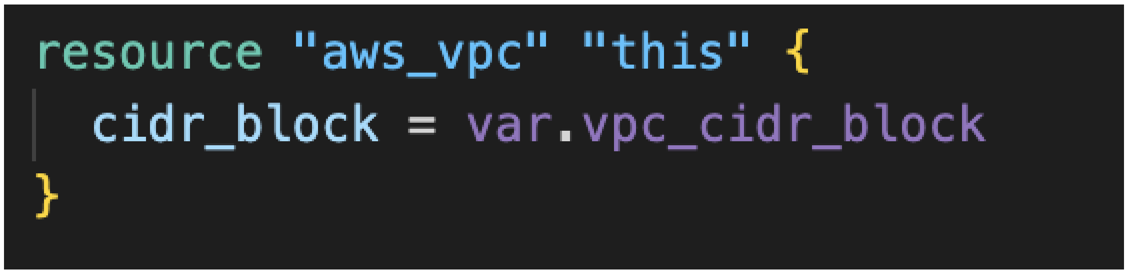
Let's consider our AWS module with three resources:

1. **`aws_vpc`**: Defines the network for the environment.
2. **`aws_subnet`**: Places a subnet inside the VPC.
3. **`aws_instance`**: Launches an EC2 instance within the subnet.

All three resources will be placed in the same `main.tf` file for simplicity.

Resource Definitions

aws_vpc



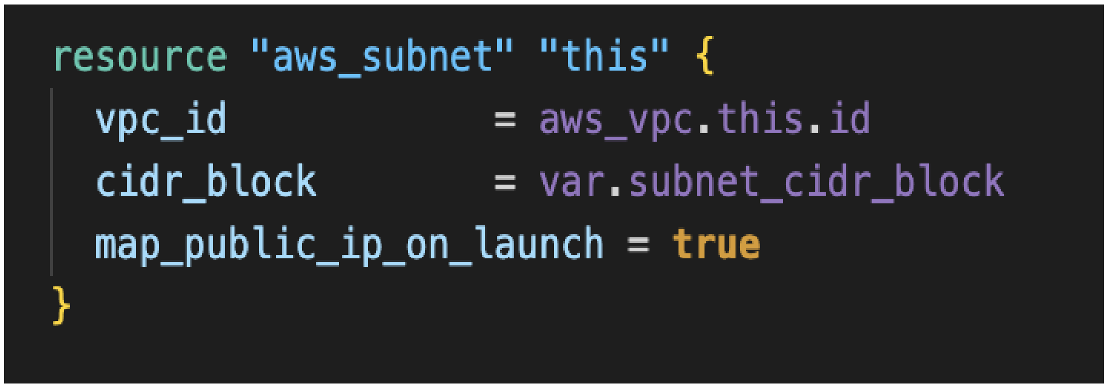
```
resource "aws_vpc" "this" {
  cidr_block = var.vpc_cidr_block
}
```

The VPC defines the overall network space for the environment:

- **`cidr_block`**: Accepts input from the `vpc_cidr_block` variable.

aws_subnet

The subnet defines a smaller network inside the VPC:

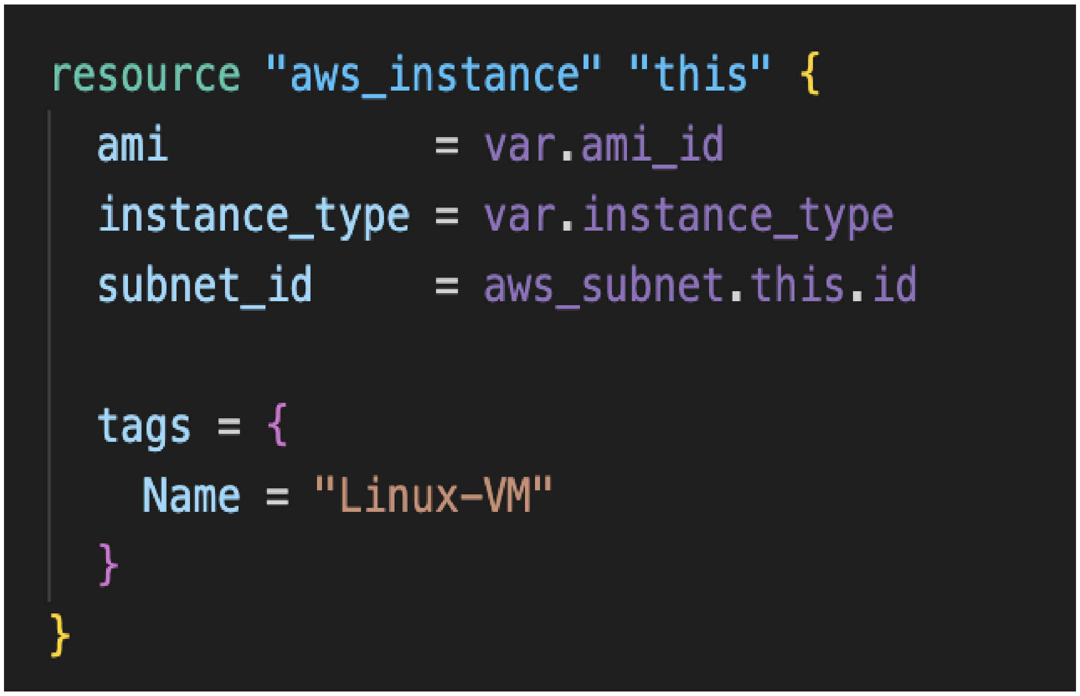


```
resource "aws_subnet" "this" {
  vpc_id                  = aws_vpc.this.id
  cidr_block              = var.subnet_cidr_block
  map_public_ip_on_launch = true
}
```

- **`vpc_id`**: References the VPC resource defined in the same module.
- **`cidr_block`**: Uses the `subnet_cidr_block` variable as input.
- **`map_public_ip_on_launch`**: Hardcoded to `true` to always enable public IP assignment.

aws_instance

The EC2 instance launches the virtual machine:



```
resource "aws_instance" "this" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = aws_subnet.this.id

  tags = {
    Name = "Linux-VM"
  }
}
```

- **`ami` and `instance_type`**: Use input variables (`ami_id` and `instance_type`) for flexibility.
- **`subnet_id`**: References the subnet resource from within the module.
- **`private_ip`**: Hardcoded to `Dynamic` to ensure no static IPs are used, following organizational best practices.

Hardcoding vs. Flexibility

1. Why Hardcode Some Values?
    - Hardcoding values ensures consistency and adherence to organizational policies.
    - For example, by setting `private_ip` to `Dynamic`, you prevent users of the module from assigning static IPs, which might not align with your standards.
2. When to Use Input Variables?
    - Use input variables for values that users need to customize, like the **AMI ID**, **instance type**, or **VPC CIDR block**.
    - This makes the module reusable for different environments.

Improving the Module Over Time

- Start with hardcoded values for simplicity.
- Gradually replace hardcoded values with input variables as you identify areas where users need flexibility.
- Retain hardcoding for values that should always follow organizational best practices.

5. Benefits of Terraform Modules

We previously touched on the benefits of using Terraform modules, but now that you've seen how to construct and use one, let's revisit those benefits in more detail. Terraform modules aren't just about organizing code—they offer significant advantages that improve the efficiency, scalability, and maintainability of your infrastructure.

1. Reusability

- Modules allow you to package and reuse configurations across multiple environments (e.g., dev, staging, and production).
- Instead of rewriting the same code, you can simply call the module with different input variables.
- *Example*: A module for setting up an AWS VPC can be reused for all your environments by changing the CIDR block and region.

2. Consistency

- By standardizing configurations into modules, you ensure that deployments follow the same structure and adhere to best practices.
- This reduces the risk of errors caused by inconsistencies or manual changes in configurations.
- *Example*: A module for deploying EC2 instances always includes proper security group rules, ensuring compliance across projects.

3. Simplified Collaboration

- Teams can share modules, making it easier to collaborate on infrastructure projects.
- Shared modules act as a "blueprint" that everyone can use, reducing onboarding time and errors.
- *Example*: A team can use a shared module to deploy Kubernetes clusters in different cloud providers, without needing to understand the underlying complexities.

4. Flexibility

- Modules let you customize configurations using input variables, while still maintaining standardization.
- This allows for tailored deployments without compromising on core best practices.
- *Example*: A load balancer module can accept inputs for the number of backend servers or target group configurations, making it adaptable for different applications.

5. Improved Maintainability

- Changes made to a module are automatically reflected wherever the module is used.
- Centralizing configurations in modules makes updates easier and reduces duplication of effort.
- *Example*: If you need to update how logging is configured in your application, you can modify the module once and apply it across all environments.

6. Next Steps

We've only scratched the surface of Terraform modules. There's still much more to explore, including:

- **Refactoring existing code** to use modules for better organization and reusability.
- **State management** when introducing modules into existing infrastructure.
- **Testing module updates** before releasing them for use to ensure stability and compatibility.

We'll dive into these topics in future posts.

As you build a catalog of modules for internal use, consider storing them in a **Terraform-supported registry**. While the public registry is an option, you might prefer to keep your modules and their supporting code private.

7. Key Takeaways

Infrastructure as Code (IaC) is fundamentally about writing code, and a key principle of good software development is using abstractions to make your code **reusable**, **scalable**, and **consistent**. In Terraform, **modules** serve as the abstraction, allowing you to group related resources and package them into reusable units.

Terraform modules can be stored locally, in source control, or in a Terraform registry. Using a registry offers benefits like built-in versioning and better discoverability for teams and organizations. By creating internal modules for your company, you can enforce sensible defaults and integrate industry best practices, making them easily reusable for both infrastructure and application teams.

You can start using modules right away by exploring the **public Terraform registry** hosted by HashiCorp.