### ****Setup a Workspace on GitHub****

#### ****1. Create a Repository on GitHub****

* Go to [GitHub](https://github.com/" \t "_new) and log in.
* Click on **New Repository** and configure it as needed.

#### ****2. Open the Workspace in VS Code****

* Click on the **Code** button in your repository.
* Select **Open with Codespaces** (if available) or manually open it in **VS Code**.

#### ****3. Install Required Tools****

* Click on the **Search** button in VS Code.
* Search for(right arrow) > **Dev Containers ->modified Dev container** and install the extension **terraform, aws cli.**

#### ****5. Rebuild the Codespace****

* Open the **Command Palette** (Ctrl + Shift + P or Cmd + Shift + P on Mac).
* Search for **Rebuild Codespace** and execute the command.

#### ****6. Configure Credentials on the Workspace****

* Set up authentication and credentials (e.g., AWS,.).
* Use the terminal to configure them securely.

Link for all documents: https://registry.terraform.io/providers/hashicorp/aws/latest/docs

Architecture of terraform file

Generally terraform project main.tf(contain fundamental instruction), , variables.tf(contains all variable), output.tf(contain output)

Modules/nom\_module ->main.tf, variables.tf, output.tf

In Modules lavel another main.tf(for assign values)

Principle commande of terraform:

Terraform init -> terraform plan -> terraform apply -> terraform destroy

Modules

The advantage of using Terraform modules in your infrastructure as code (IaC) projects lies in improved organization, reusability, and maintainability. Here are the key benefits:

Modularity: Terraform modules allow you to break down your infrastructure configuration into smaller, self-contained components. This modularity makes it easier to manage and reason about your infrastructure because each module handles a specific piece of functionality, such as an EC2 instance, a database, or a network configuration.

Reusability: With modules, you can create reusable templates for common infrastructure components. Instead of rewriting similar configurations for multiple projects, you can reuse modules across different Terraform projects. This reduces duplication and promotes consistency in your infrastructure.

Simplified Collaboration: Modules make it easier for teams to collaborate on infrastructure projects. Different team members can work on separate modules independently, and then these modules can be combined to build complex infrastructure deployments. This division of labor can streamline development and reduce conflicts in the codebase.

Versioning and Maintenance: Modules can have their own versioning, making it easier to manage updates and changes. When you update a module, you can increment its version, and other projects using that module can choose when to adopt the new version, helping to prevent unexpected changes in existing deployments.

Abstraction: Modules can abstract away the complexity of underlying resources. For example, an EC2 instance module can hide the details of security groups, subnets, and other configurations, allowing users to focus on high-level parameters like instance type and image ID.

Testing and Validation: Modules can be individually tested and validated, ensuring that they work correctly before being used in multiple projects. This reduces the risk of errors propagating across your infrastructure.

Documentation: Modules promote self-documentation. When you define variables, outputs, and resource dependencies within a module, it becomes clear how the module should be used, making it easier for others (or your future self) to understand and work with.

Scalability: As your infrastructure grows, modules provide a scalable approach to managing complexity. You can continue to create new modules for different components of your architecture, maintaining a clean and organized codebase.

Security and Compliance: Modules can encapsulate security and compliance best practices. For instance, you can create a module for launching EC2 instances with predefined security groups, IAM roles, and other security-related configurations, ensuring consistency and compliance across your deployments.

**Terraform Backend:**

In Terraform, the **backend** is a configuration that determines how Terraform stores its state files.

However, using a backend allows you to store the state remotely, which is essential for collaboration and production environments.

 By default, Terraform stores the state locally in a file named terraform.tfstate.

The state file keeps track of the resources Terraform manages. Later, if you want to modify resources or change the infrastructure, the state file ensures that Terraform understands the current state of your infrastructure. Without a state file, Terraform would not know about the existing resources and might create new infrastructure instead of updating the existing one. With the state file, Terraform maintains the same infrastructure and only makes the desired changes to align with your configuration

**To see terraform stateful file:**

**Command : terraform show**

How Does Locking Work?

* When a Terraform operation (e.g., apply, plan, destroy) starts, Terraform attempts to acquire a lock on the state file.
* If another operation is already in progress, Terraform will wait for the lock to be released or fail with an error, depending on the configuration.
* Once the operation is complete, the lock is released.

**DynamoDB**is commonly used as a locking solution for Terraform when using an S3 backend to store the state file. This combination ensures that:

1. State File Storage: The Terraform state file is stored in an S3 bucket.
2. State Locking: DynamoDB is used to manage locks, preventing multiple users or processes from modifying the state file simultaneously.

**file Provisioner:**

The file provisioner is used to copy files or directories from the local machine to a remote machine. This is useful for deploying configuration files, scripts, or other assets to a provisioned instance.