Working in a Team

Working in a Team

- Not always easy or transparent
  - Whatever the group is doing is opaque to every other group
  - Result in unstable or inconsistent environment
- Fear of adopting automation tools
  - Especially tools that apply changes to existing systems
  - Not sure what might break
- Making changes easily and safely (IaC context)
  - Conventional wisdom is to sacrifice one for other

Terraform Modules

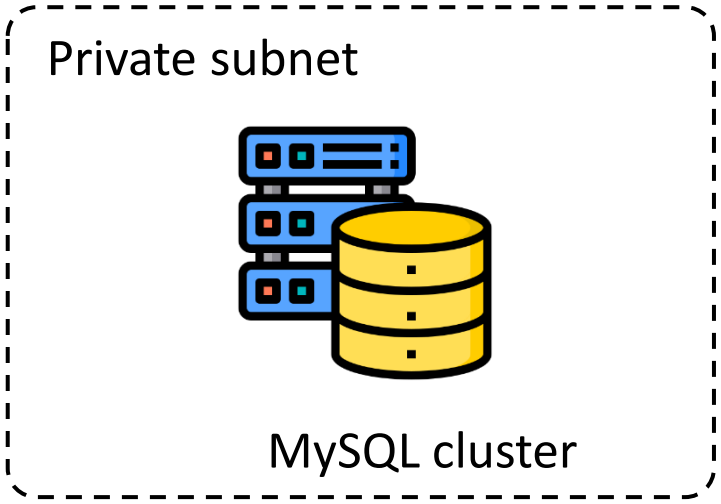
- Reusable infrastructure/architectural pieces
  - Any Terraform project is a module
- Module has
  - A set of parameters - eg the number of server instance to provision
  - A set of outputs - eg IP address of all the servers
  - Internal states - local variables, defined and used by the module
- Modules can be shared by publishing to public repository
- Terraform files inside a sub-directory becomes a module
- Modules require their own providers
  - Even if the root module is using the same provider

Example - Module

```
resources.tf

resource digitalocean_vpc vpc {
  name = "${var.name}_vpc"
  region = var.region
}

resource digitalocean_database_cluster db {
  name = "${var.name}_db
  region = var.region
  size = var.db_size
  engine = "mysql"
  version = "8"
  node_count = var.node_count
  private_network_uuid = resource.digitalocean_vpc.vpc.ip
}
```



Module will also need a provider configuration. Not shown

Example - Module

inputs.tf

```
variable region {
  type = string
  description = “Region to ...”
}
variable name {
  type = string
  description = “VPC name”
}

variable db_size {
  type = string
  description = “Database instance”
  default = “db-s-1vpc-2gb”
}

...
```

outputs.tf

```
output vpc_id {
  value = digitalocean_vpc.vpc.id
  description = “VPC id”
}

output db_connection {
  value = digitalocean_database_cluster.db.uri
  description = “MySQL connection string”
}

output db_user {
  value = digitalocean_database_cluster.db.user
  description = “MySQL user”
}

output db_user {
  value = digitalocean_database_cluster.db.password
  description = “MySQL password”
  sensitive = true
}
```

Example - Module

myapp.tf

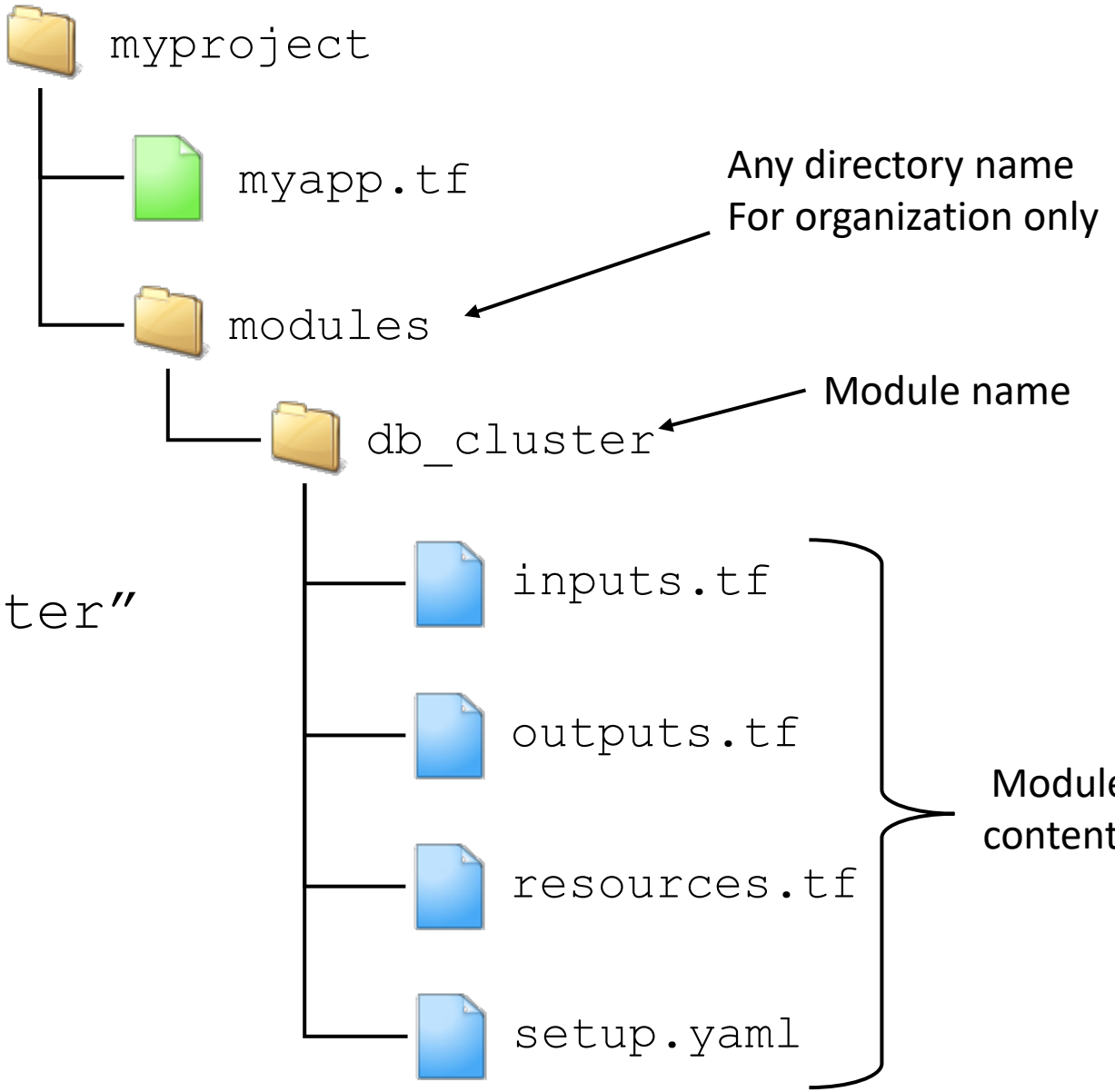
```
module myapp {
  source = "./modules/db_cluster"

  name = "myapp"
  region = "sgp1"
  node_count = 2
}
```

Module instance name

Location of the module directory

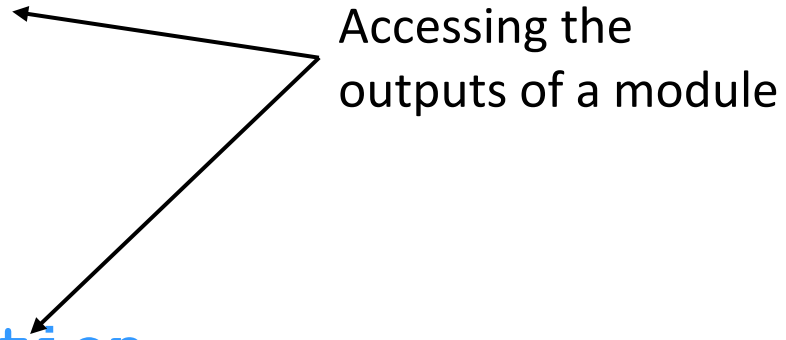
Module configurations input variables



Example - Module

```
resource digitalocean_droplet mydroplet {
  ...
  vpc_uuid = module.myapp.vpc_id
}


output db_connection {
  value = module.myapp.db_connection
}
```



Local Variables

- Local variables are used within the module
  - Not accessible by any resources outside of the module
- Local variables are access internally within the module with the `local` prefix

locals.tf

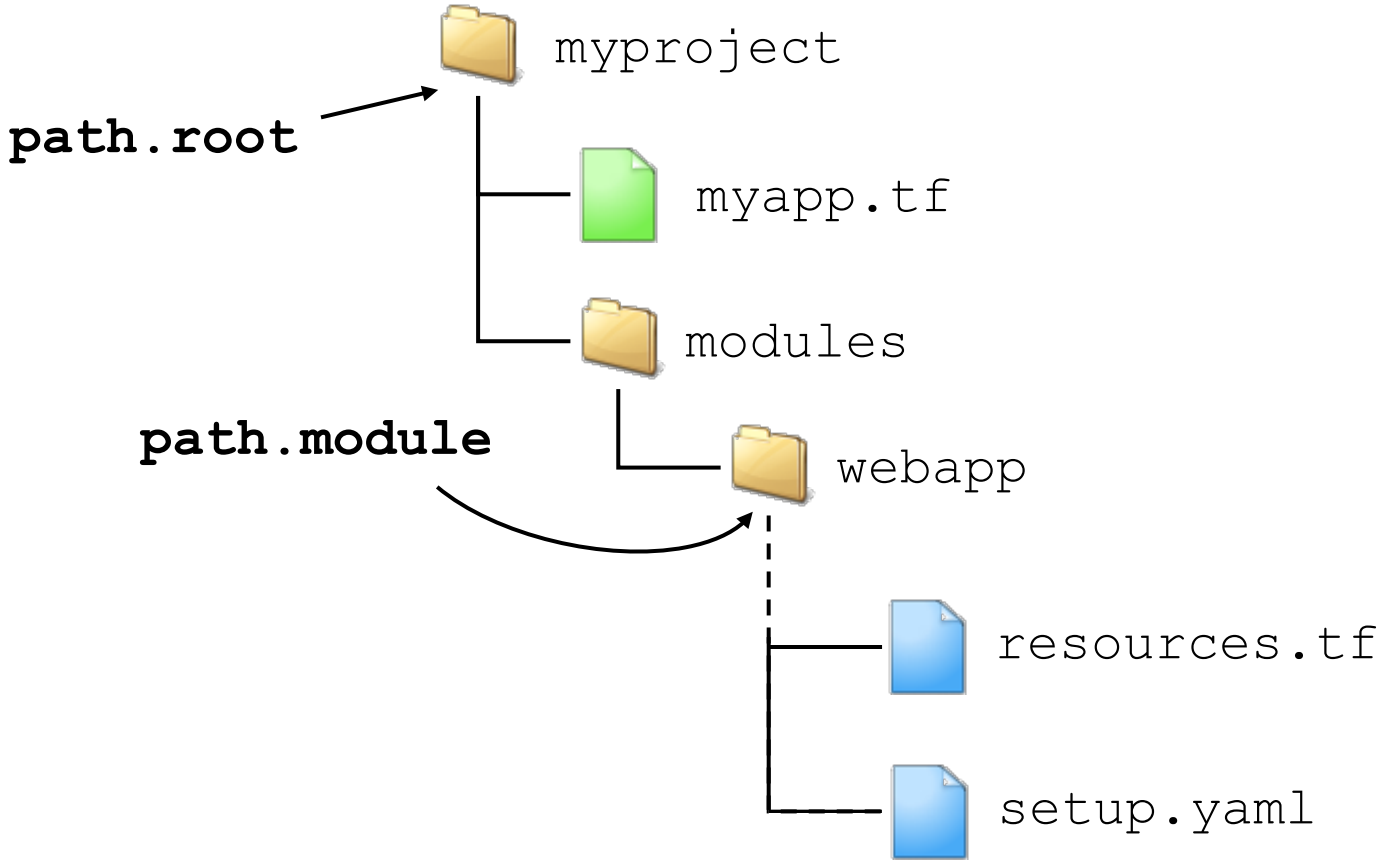
```
locals {
  db_username = “fred”
}
```

resources.tf

```
resource digitalocean_database_user dbuser {
  cluster_id = digitalocean_database_cluster.db.id
  name = local.db_username
}
```

File Path

- Modules provides the following method to transparently reference files relative to its location
  - `path.module` - the path to the module root
  - `path.root` - the path of the root module viz. the module that is using




```
resources.tf
resource digitalocean_droplet droplet {
  name = var.droplet_name
  image = var.droplet_size
  region = var.droplet_region
  size = var.droplet_size
  user_data = file("${path.module}/setup.yaml")
}
```

State File

- State file records the resources provisioned by Terraform
  - Created when you run `terraform apply`
- Maps the Terraform resource with the actual resource
  - Consult the state file whenever you run `terraform plan`
  - Reconcile what is defined in the `.tf` files vs the real world
  - Only provisioned those missing resources when apply is executed

State File

- Shared location of the state file
  - State files are created locally, so not accessible to another member in the group
  - Other team members can provision duplicate resource without access to a common shared state file
  - Simple solution is to save state file in a shared network accessible storage
- Corrupting state file with multiple simultaneous updates
  - Race condition, may cause data loss or corruption of the state file
- Single state files for multiple different environment
  - A state file only store the state of one environment eg. testing
  - Members in the same team cannot update the resources, may accidentally cause outage or failure in the test

Backends

- Pluggable state file storage
  - Can be configured to create, read and update state file from different backend storages
- Default backend is local
  - State file is created in the current directory, viz. `path.root`
- Other backends include S3, Postgres, Kubernetes, HTTP (WebDAV), EtcdV3,
  - Some backend supports locking
  - See https://www.terraform.io/docs/language/settings/backends/index.html
- Can only configure 1 backend per project

Example - local

```
terraform {
  required_version = ">= 0.15.0"
  required_providers = {
    digitalocean = { ... }
  }

  backend local {
    path = "path/to/file/terraform.tfstate"
  }
}
```

Specify the backend to use.
`local` is the default

Change the default location of the state file

The file can be stored in local hard disk or a shared directory like NFS or Gdrive
local does not provide file locking

S3 Backend

- Backend supports all AWS backend features including versioning
  - Best practice to enable versioning for S3, for recovering old state files in the event of an accident
- S3 bucket needs to have actions
  - `s3:ListBucket`, `s3:GetObject`, `s3:PutObject`
  - Either for a specific principal for the backend or all
- Supports locking by configuring a DynomoDB table name
- Table needs to have the following actions
  - `dynomodb:GetItem`, `dynomodb.PutItem`, `dynomodb.DeleteItem`

Example - S3 Compatible Storage

```
terraform {
  ...

  backend s3 {
    skip_credentials_validation = true
    skip_metadata_api_check = true
    skip_region_validation = true
    skip_requesting_account_id = true
    skip_s3_checksum = true
    endpoints {
      s3 =
"https://sgp1.digitaloceanspaces.com"
    }
    region = "sgp1"
    bucket = "mybucket"
    key = "states/terraform.tfstate"
  }
}
```

Skip all validation for S3 compatible storage

- Standard S3 parameters
- Access key and secret access key for S3 bucket
  - Can be configured as arguments in the backend (not recommended)
  - As environment variable `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
  - Read from `~/.aws/credentials`
- Need to disable validation if using S3 compatible storage
  - Eg. DigitalOcean, Cloudflare

Reconfigure Backend

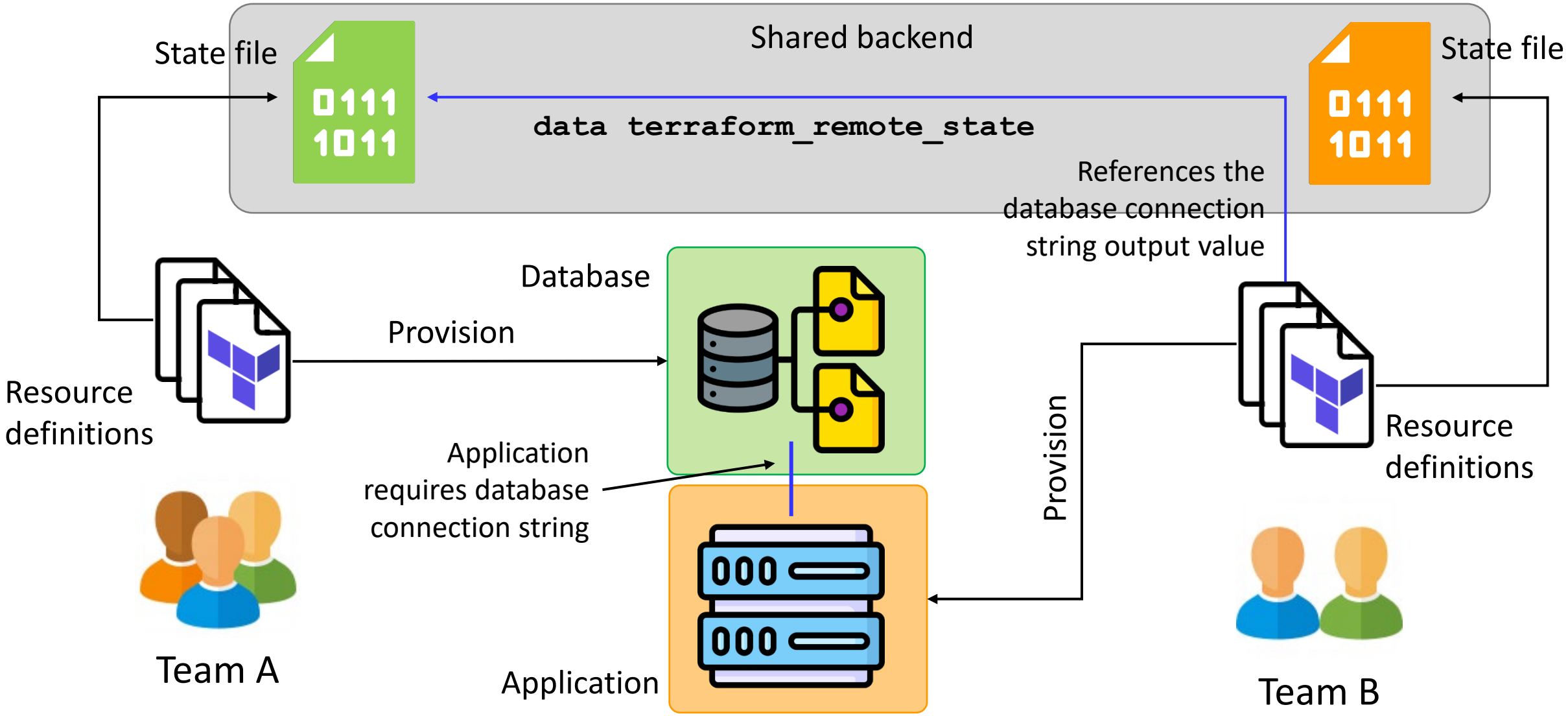
- Need to reconfigure the backend if it changes

```
terraform init -reconfigure
```

Sharing State File

- Remotely accessible state files' output can be read by other Terraform scripts
  - Need to configure access for others to use
  - Can only access root output values
- Accessed with the `terraform_remote_state` data source
  - See https://www.terraform.io/docs/language/state/remote-state-data.html
- Scripts using the data source will not be notified if the data source is deleted

Sharing and Accessing State File



Example - Sharing State File

```
data terraform_remote_state db {
  backend = "s3"
  config = {
    ...
  }
}
```

Type of backend

Configure the remote data source. Configuration should be the same as the backend

```
resource digitalocean_app web {
  spec {
    name = "web"
    region = var.region
    env {
      key = "DB_URL"
      value = data.terraform_remote_state.db.outputs.db_connection
      scope = "RUN_TIME"
      type = "GENERAL"
    }
    service { ... }
  }
}
```

Can only access root project outputs viz. not module outputs used within the project `db_connection` must be an output

The provisioned databased is used by another project

No Variables in Backend Block

- Backend blocks cannot reference variables
  - See https://www.terraform.io/docs/language/settings/backends/configuration.html#using-a-backend-block
- Alternative
  - Use environment variables
    - Eg S3 - `AWS_REGION`, but not for all arguments
  - Use a variable file to pass to Terraform
    - Partial configuration

Example - Partial Configuration

```
providers.tf
terraform {
  required_version = ">= 0.15.0"
  required_providers = {
    digitalocean = { ... }
  }

  backend s3 {
    key = "terraform.tfstate"
  }
}
```

```
s3_backend.hcl
bucket = "mybucket"
region = "sgp1"
access_key = "...."
secret_key = "...."
```

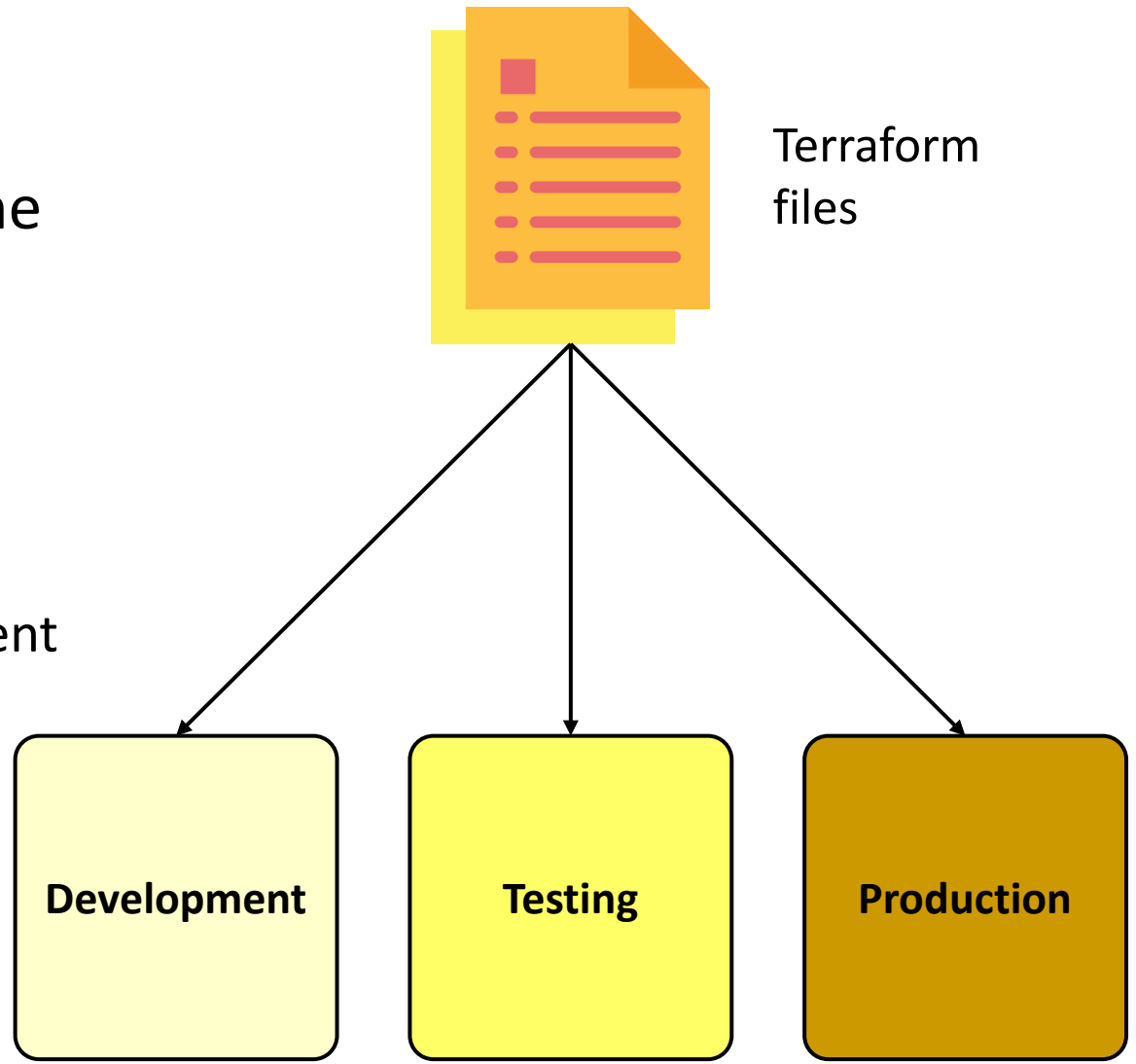
All arguments required by specific backend

```
terraform init -backend-config=s3_backend.hcl
```

Initialize Terraform (with backend) by providing `.hcl` file

Isolating Environment

- Sharing state file allows every one in the team to use the same environment
  - Share provisioned resources
- Separate environment so one environment does not interfere with another
  - Eg. testing a new OS version in development causes the production instances to be re-provisioned with the new version
- Two ways to isolate environment
  - Workspace
  - Directory structure



Isolating Environment with Workspaces

- Workspaces allow you to create different state files in different workspaces
  - Terraform will report that it will have to provision all resources in a newly created workspace
- Terraform CLI will read all the `.tf`, `.tfvars` files from the current directory
  - Unless you edit the files before running apply, you will get the same set of resources
  - Not way to pass to the CLI a subset of the files from your current directory
- Error prone because if you forget to switch workspace before performing apply, can destroy the existing provisioned resource from the current workspace
- Workspace name as variable `terraform.workspace`

- List workspace

```
terraform workspace list
```

- Create a new workspace and switch to it

```
terraform workspace new myspace
```

- Change to an existing workspace

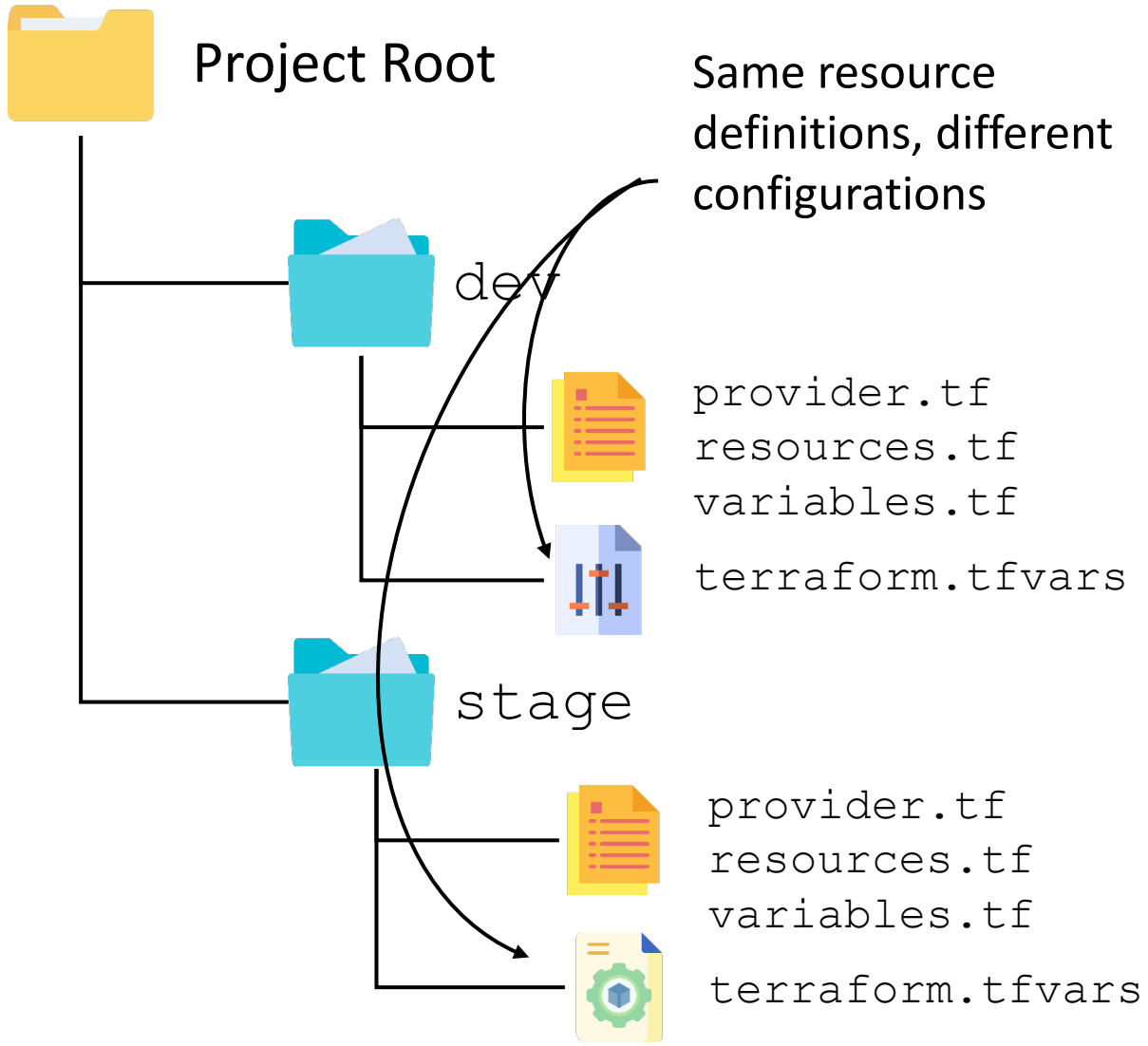
```
terraform workspace select otherspace
```

- Delete a workspace

```
terraform workspace destroy otherspace
```

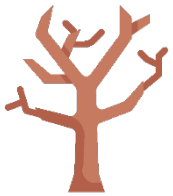
Isolating Environment with Directory Structure

- Create a directory for each environment eg. dev, test, prod
- Each directory will contain
  - Provider and resource definition files
  - Variable files specific to each environment
- Provider and resource definition are typically common across all the environment
- Better approach than workspaces but lots of duplication
- Reduce duplication by modularizing common and complex components

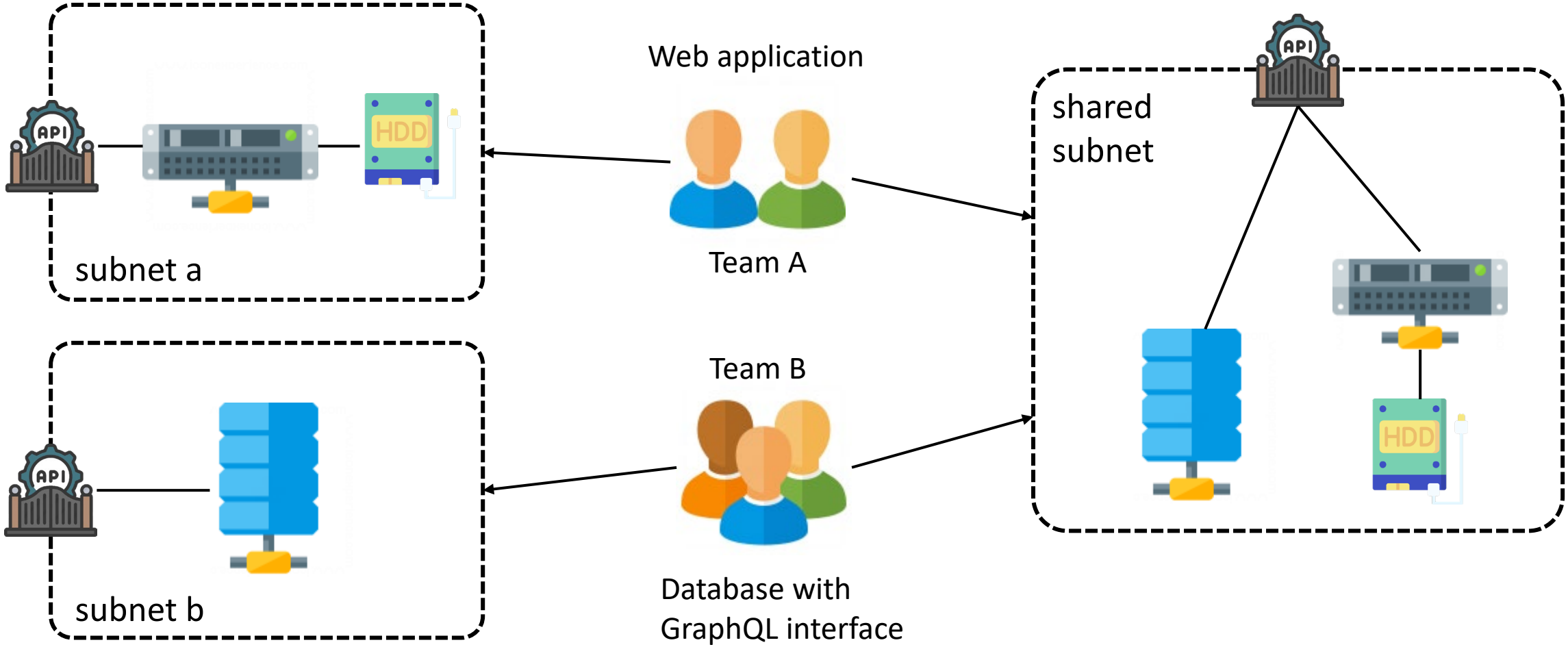


Shared Elements - Resource Definitions

- Configuration, resources and infrastructure definition
  - Eg database connection configs, VM details, layout of the infrastructure/stacks
  - Slightly different in each team
- Handle by duplicating the files or parameterizing the shared elements
- Duplicating avoids tight coupling
  - WET - Write Everything Twice
  - Resource definitions are simpler - do not need to handle multiple scenerios
  - May diverge from prescribed company policy for using and consuming cloud resources
    - Eg. only use company's encrypted images
- Parameterizing avoids duplication
  - DRY - Don't Repeat Yourself
  - Treat resource definitions like APIs
  - Best practices can be shared across the teams



Shared Elements - Environment



Shared Elements - Environments

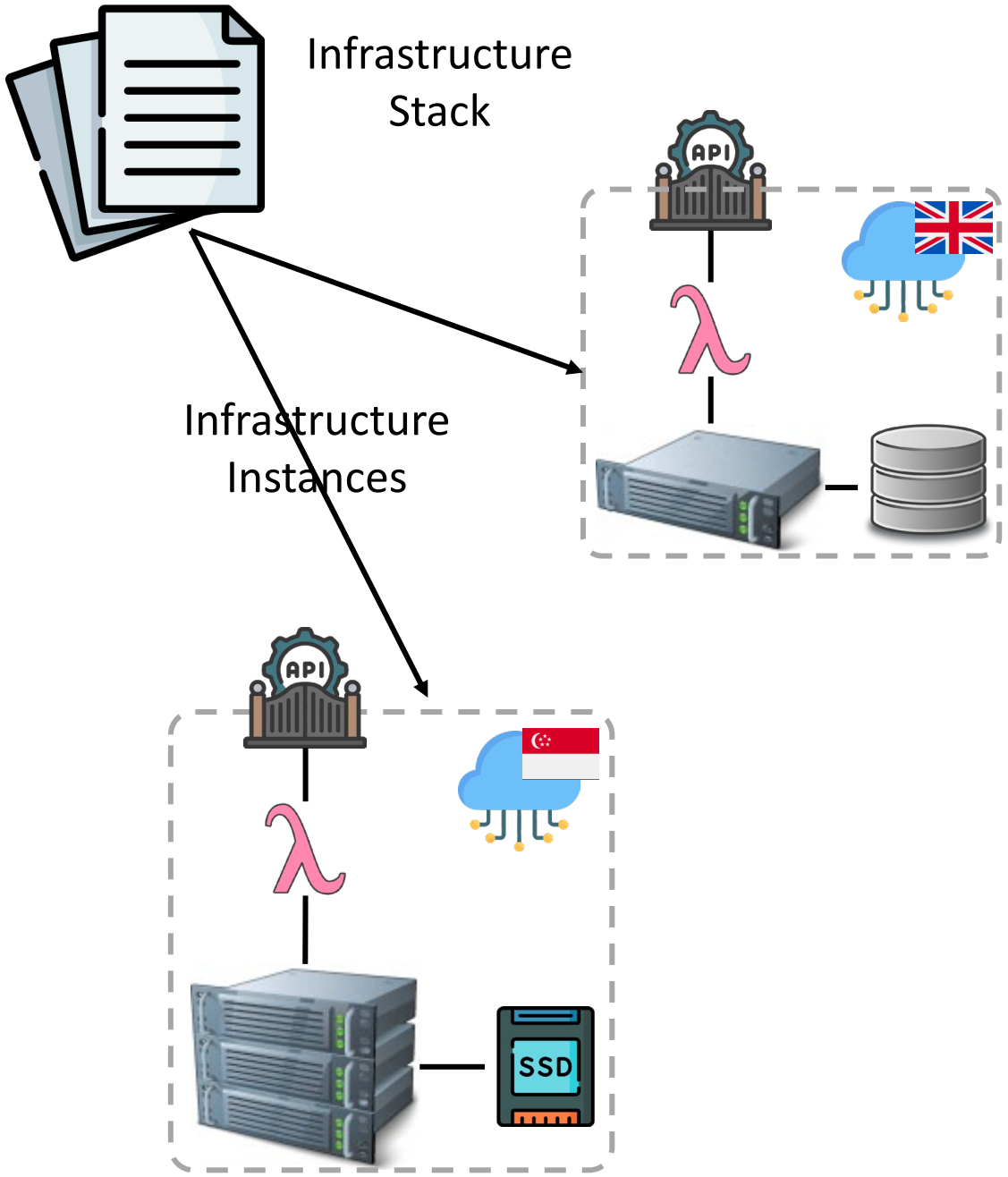
- Teams may have different environments for development/testing/deployment
  - No sharing, clean, do not interfere with other teams
  - Can be expensive, duplicate resources
- Provisioning into the same environment
  - Leverage share resources like internet gateways, bastion host, etc.
  - Consistent eg. security setting, routing rules, etc
- Sharing stateful resources
  - Servers - provisioning a new server or use an existing instance
    - Eg for deploying application - Docker or Kubernetes cluster

Shared Elements - Environments

- Sharing stateful resources
  - Within a network, partition into subnets
    - Eg. Network - 10.10.10.0/24
    - Eg. Subnet - 10.10.10.0/26 - 4 subtnets, 62 host per subnet

Infrastructure Stack

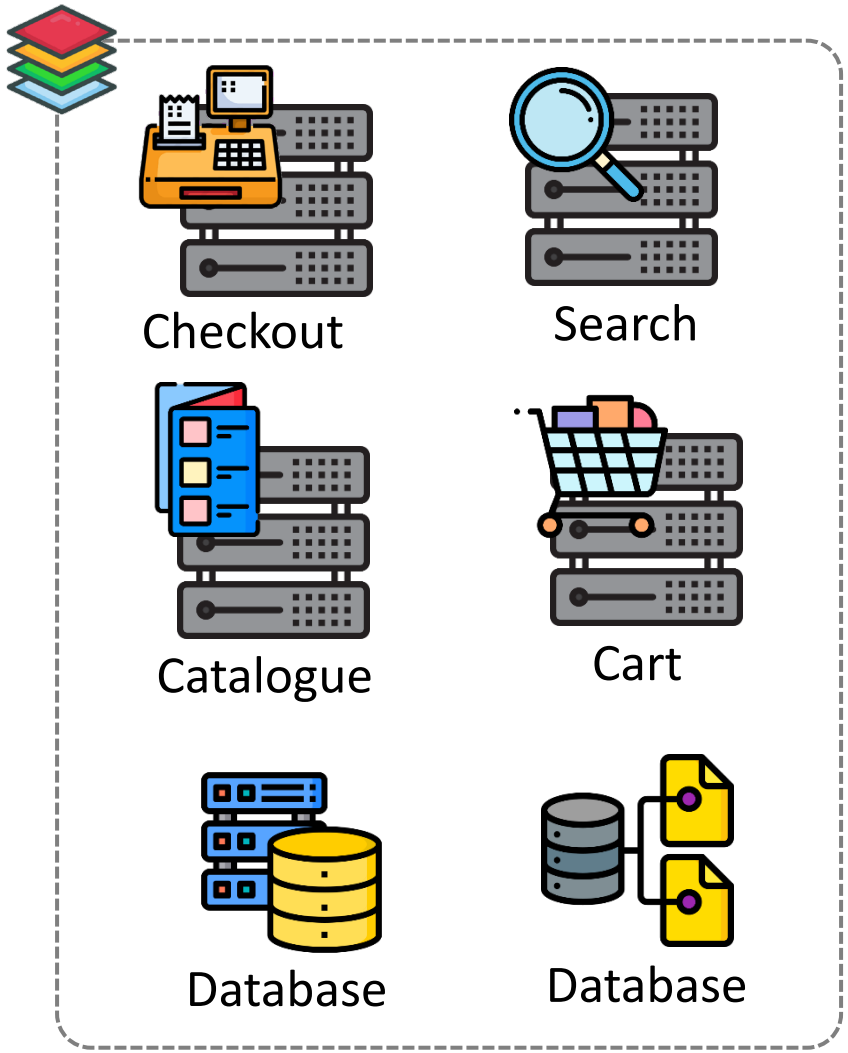
- Infrastructure stack includes
  - Images created from Packer
  - Provisioning with Terraform
  - Configured using Ansible
  - Templates for Terraform and Ansible
- Independent and parameterized to create the same infrastructure with different configuration
- Produces artefacts that can be used
  - Directly by the ‘root’ stack
  - As input to other stacks



Architectural Patterns

- Architectural patterns is common design used for infrastructure
  - Includes software installed and configurations
  - Similar in idea to design patterns for software design
- Infrastructure architectural patterns
  - Monolithic
  - Application group
  - Service
  - Micro

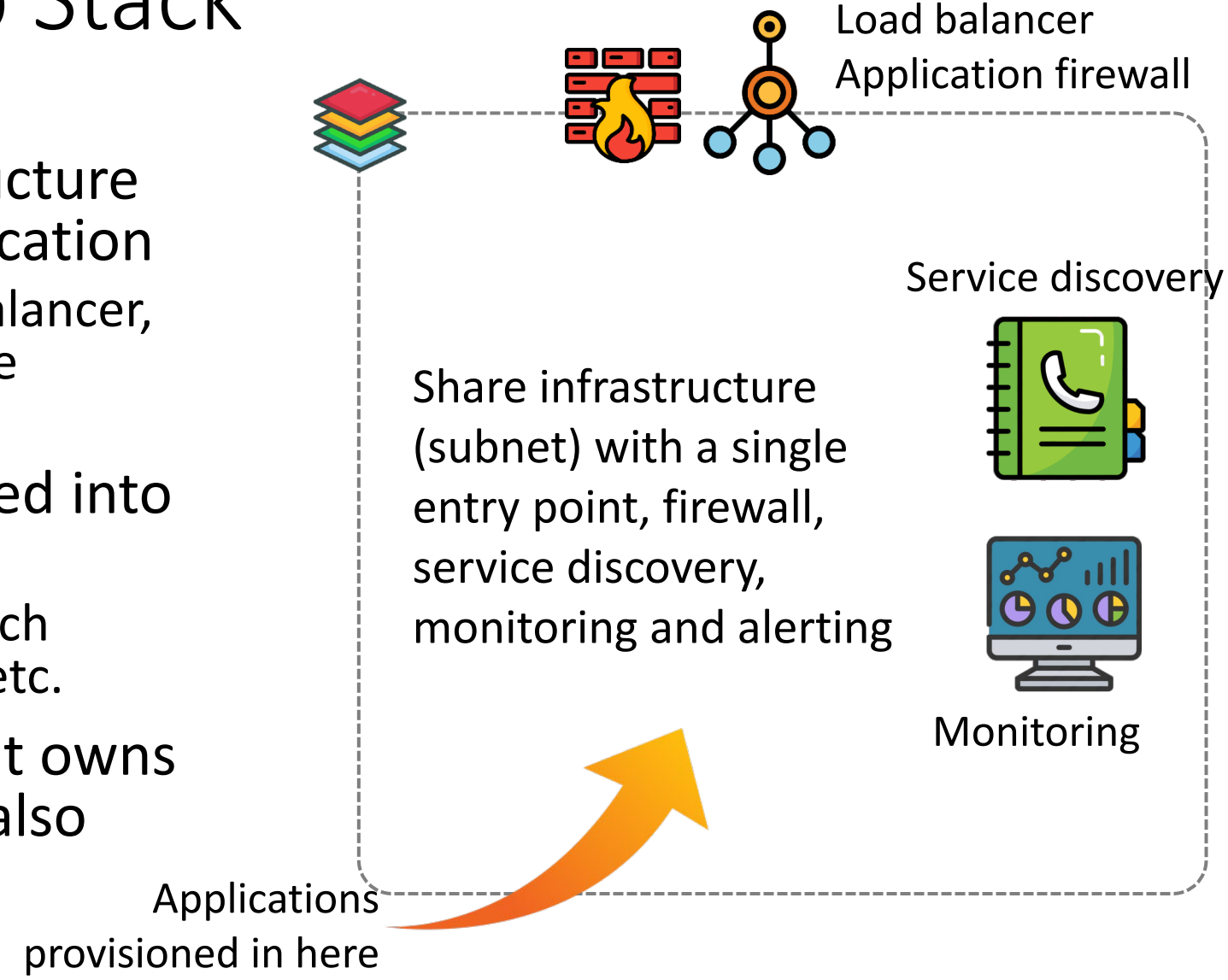
Monolithic Stack



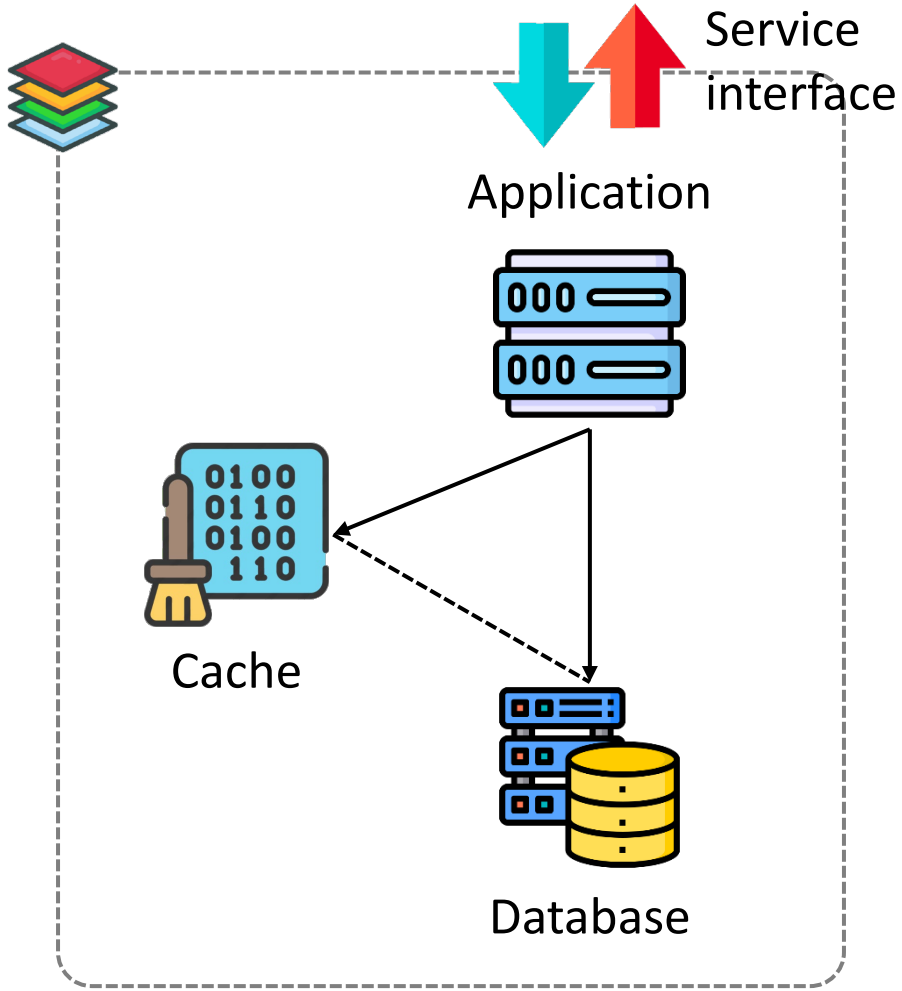
- All in one, easy to start
  - Easy to start, when grow organically will contain too many infrastructure elements
- When the stack grows
  - Difficult to separate the relationship within the
  - May take longer time to validate and provision
  - Can cause widespread changes with `terraform apply` when a change is made to an element
- Key indicator that a stack is monolithic is the manual coordination between members in the team

Application Group Stack

- Create a stack for infrastructure shared by a group of application
  - Resources like VPC, load balancer, application firewall, etc. are common and shared
- Applications are provisioned into the shared infrastructure
  - Eg. catalogue service , search service, checkout service, etc.
- Works well if the team that owns the shared infrastructure also owns the application



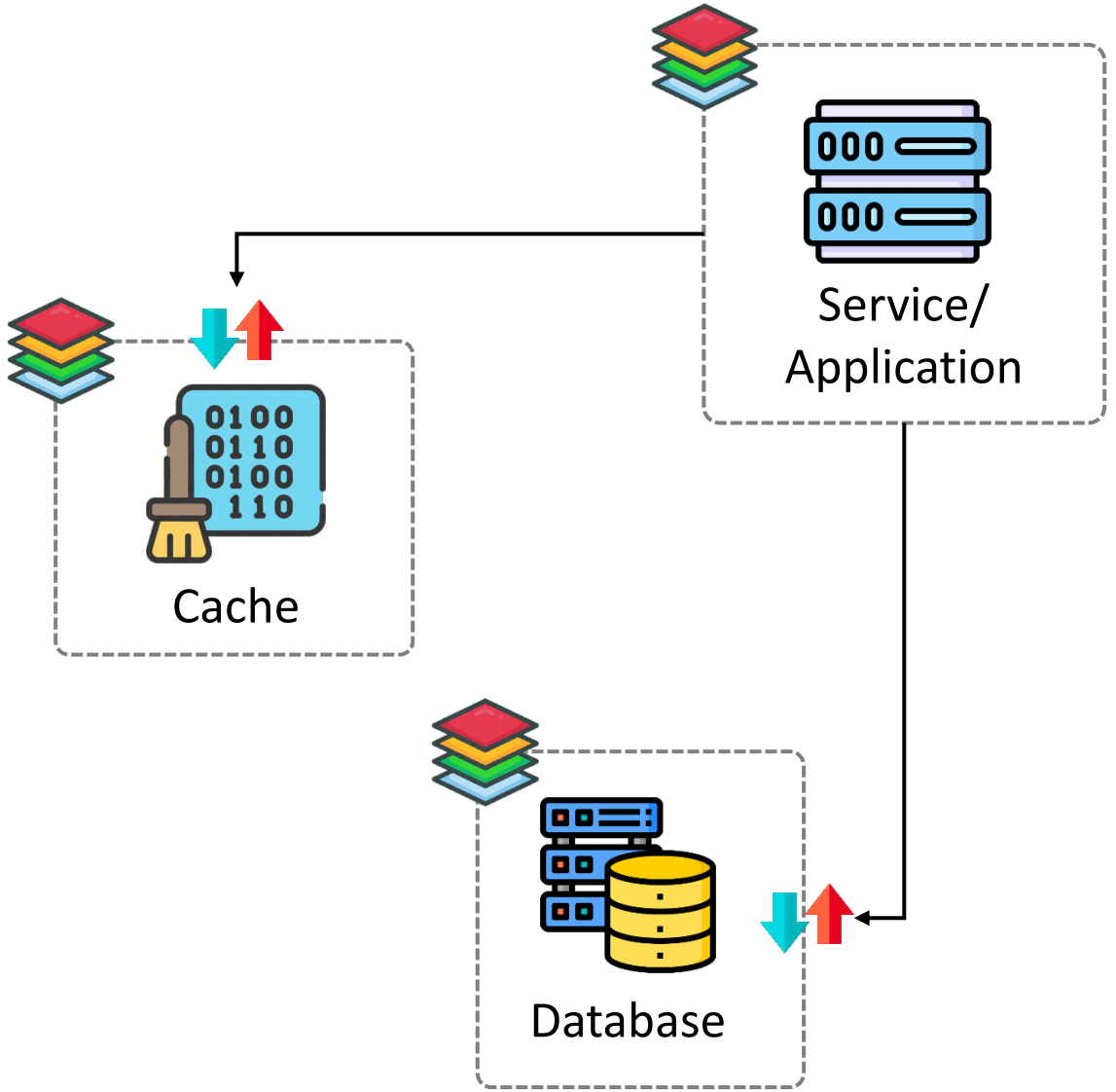
Service Stack



- Create a stack for each service
- Service stack may include
  - Compute (eg VM) for running the service/application
    - Service can be installed with Ansible or ‘burnt’ directly onto the image
  - Database to be used exclusively by the service
- Well define interface to access the service
- Works well with application group stack

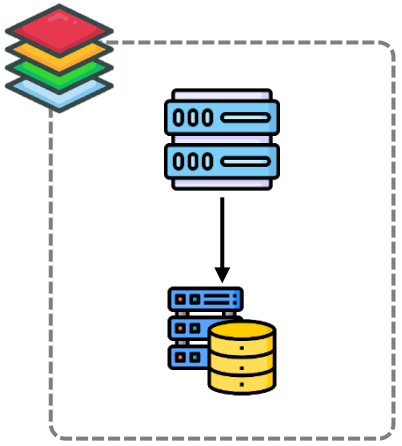
Micro Stack

- Divides a single service across multiple stacks
- More moving parts, more complex
- Better availability when service is important or complex
  - Each stack can be configured and managed separately by expertise in each of the domain
  - Allow for different lifecycles for each of the service's components

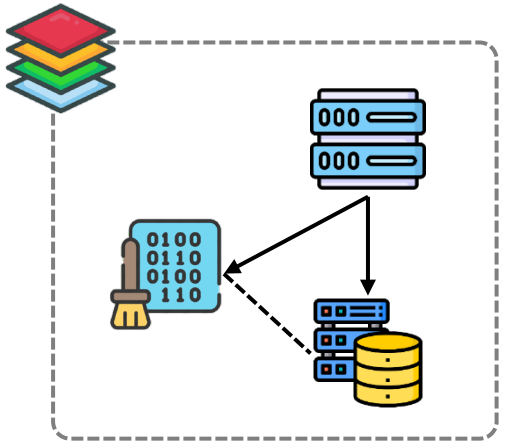


Environments

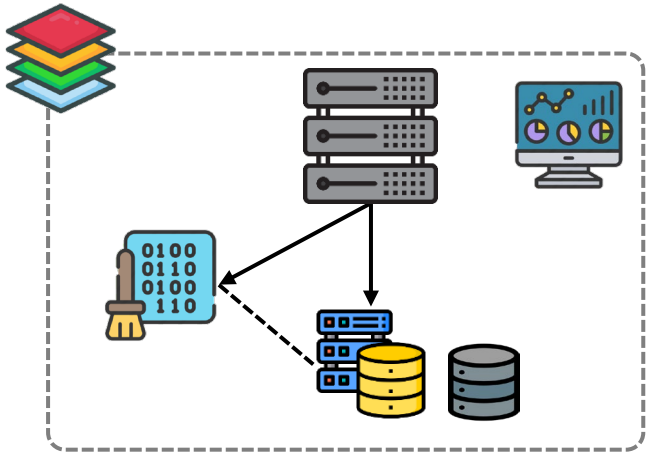
- Environments are resources and software used for a particular purpose
  - Eg. testing, development, staging, production
- Stack elements remains the same but the configuration of these elements may be slightly different
  - Eg. server size, clustered vs single instance databases, etc.
  - Eg. additional elements that may not be required eg. backups



Development



Testing



Production

Appendix

Reuse Common Task

- Keep common tasks in their separate file
  - Eg. upgrade all packages to the latest version
- Task file can be included in plays with `include_tasks`
- No explicit mechanism to pass parameters into the task
  - Task files can access all the variables defined in a play
- import_tasks is similar to include_tasks but
  - import_tasks is static - the tasks in the give YAML file is read and parsed when the playbook is parsed
  - include_tasks is dynamic - the tasks in the given YAML file is read and parsed when the playbook is executed
    - Allow different tasks to be include dynamically

Example - import_tasks

```
tasks:
- name: Update and install new packages
  import_tasks: tasks/apt_update_install.yaml
  var:
    package_list:
    - mysql-server
    - wordpress
    - php
    - libapache2-mod-php
    - php-mysql
```

List of task to be added to a playbook

```
tasks/apt_update_install.yaml
- name: Update cache
  apt:
    update_cache: yes
- name: Upgrade installed packages
  apt:
    upgrade: yes
- name: Install new packages
  apt:
    name: '{{ item }}'
    state: latest
  when: package_list is defined
  loop: '{{ package_list }}'
```

Variable from playbook can be accessed by the task file

Example - include_tasks

```
tasks:
- name: Update and install new packages
  include_tasks: 'tasks/{{ ansible_pkg_mgr}}.yaml'
  var:
    package_list:
    - mysql-server
    - wordpress
    - php
    - libapache2-mod-php
    - php-mysql
```

The expression is dynamically evaluated when the playbook is executed
Determine either to use `apt.yaml` or `yum.yaml` file to install the package depending on the package manager of the target host

Roles

- Playbook is used to configure a host for a particular purpose
  - Eg. database server, bastion host, CMS/WordPress, etc.
  - Eg. Need to configure MySQL server multiple times in different projects
- A role is a set of task applied to a host to configure the host for a specific purpose
  - Provide an intuitive role name to identify the purpose of the tasks
  - Hide complex configuration by providing a way to reuse playbooks
  - Share know-how by reusing tasks
- It is independent and can be incorporated into any playbooks

Using a Role

- Search from Ansible Galaxy, filter by roles
  - https://galaxy.ansible.com/
- Community, can publish the roles you have created
  - Require a GitHub account
- Documentation on how to use the role in your playbook
  - Version and OS platform
- Roles are executed before tasks
- Instructions to download and install a role
  - `ansible-galaxy`

```
ansible-galaxy install geerlingguy.docker
```

Install Docker role
from Ansible Galaxy

https://galaxy.ansible.com/geerlingguy/docker

Example - Using the Docker

```
site.yaml
- name: Install Docker
  host: docker_host
  pre_tasks:
  - name: Create user
    user:
      name: '{{ docker_user }}'
      password: '{{ docker_user_password }}'
  roles:
  - role: roles/docker
    vars:
      docker_users:
      - '{{ docker_user }}'
  tasks:
  - name: Enable remote access
    lineinfile:
      path: /lib/systemd/system/docker.service
      regexp: '^ExecStart='
      line: ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock-
H=tcp://0.0.0.0:2375
  - name: Restart Docker daemon
    ...
```

Like task but executed before roles or tasks

Role is downloaded to roles directory

Role variable(s) to configure Docker

```
- name: Restart Docker daemon
   systemd:
     name: docker
     daemon_reload: true
     state: restarted
```

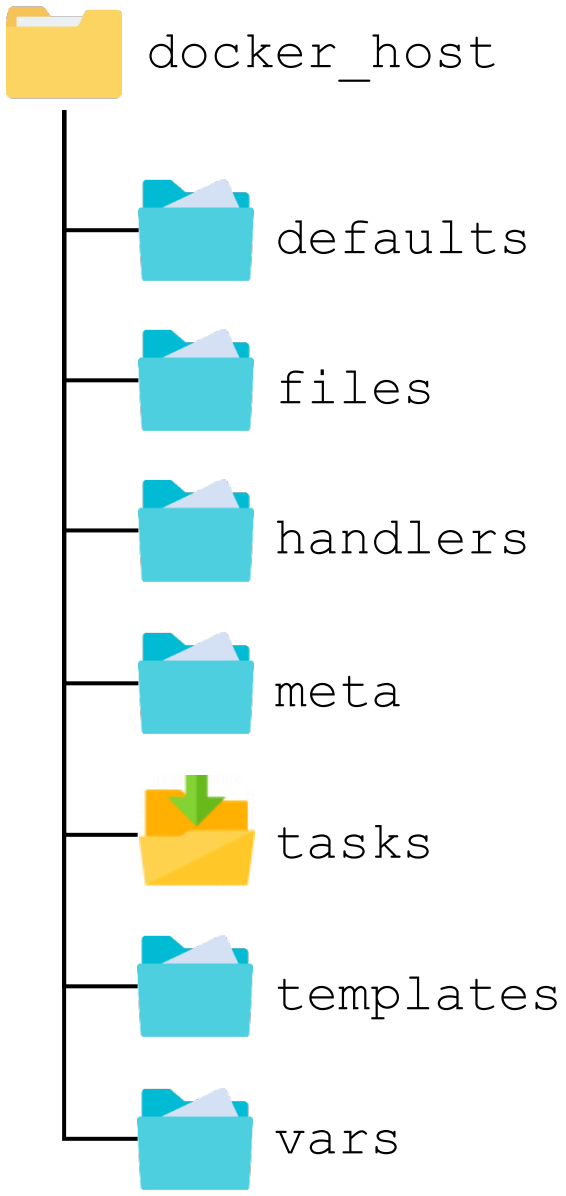
Note: an alternative way to provision Docker host is to use docker-machine https://docs.docker.com/machine

Creating Role

- A set of pre defined directory that can be generated with ansible-galaxy

```
ansible-galaxy init docker_host
```

- Directory uses
  - `defaults` - default variables
  - `files` - files, eg configuration, to be copied over to the target
  - `handlers` - for handlers
  - `tasks` - contains all the task to be performed by the role. When a role is applied, Ansible will perform the list of task listed here
  - `templates` - dynamically generated files
  - `vars` - other variables, override defaults
  - `meta` - metadata for the role



Example - Docker Host Role

```
default/main.yaml
---
package_list:
- apt-transport
- ca-certificates
- software-properties-common
- curl
- openssh-server
docker_gpg_key: https://download.docker.com/linux/ubuntu/gpg
docker_repo: “deb [arch=amd64] https://download.docker.com/linux/ubuntu
focal stable”
```

Example - Docker Host Role

tasks/main.yaml

```
---
- name: Install packages
  apt:
    update_cache: yes
    name: '{{ item }}'
    state: latest
  loop: '{{ package_list}}'
- name: Add Docker GPG key
  apt-key:
    url: '{{ docker_gpg_key }}'
    state: present
- name: Add Docker repository
  apt_repository:
    repo: '{{ docker_repo }}'
    state: present
```

User defined variables

```
- name: Install Docker
  apt:
    update_cache: yes
    name: docker-ce
    state: latest
- name: Create non-root Docker user
  user:
    name: '{{ docker_user }}'
    groups: [ 'docker' ]
  when: docker_user is defined
- name: Check if SSH key is present
  stat:
    path: '{{ docker_ssh_key }}'
  register: docker_ssh_key is defined
- name: Add SSH key to user
  authorized_key:
    user: '{{ docker_user }}'
    key: '{{ lookup('file', docker_ssh_key }}'
  when: ssh_key.stat.exists
```

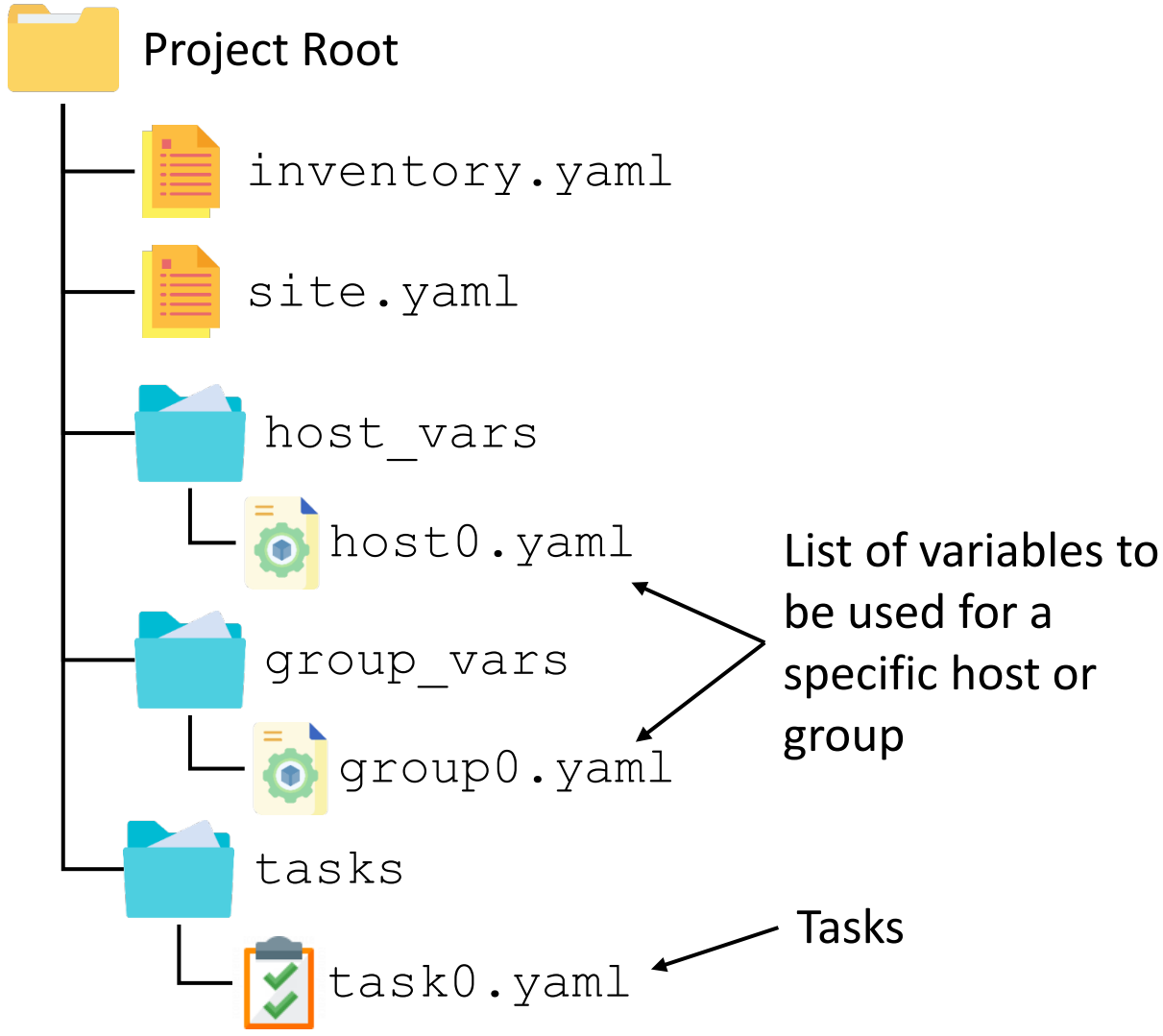
From https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04

Example - Using Docker Host Role

```
site.yaml
- name: Install Docker
  hosts: a_docker_host
  roles:
  - role: roles/docker_host
    vars:
      docker_user: fred
      docker_ssh_key: /path/to/public/key
```

Structuring an Ansible Project

- Single playbook cannot scale
- Use separate specially named directories for hosts and groups
- Put commonly used task in a separate file so that playbooks can reuse them



`host_vars` and `group_vars` Directories

- Special directories to define host and group variables
- Keep the number of variable declarations in inventory file to a minimum
- File name is the name of the host or group
  - Eg. `hosts_vars/localhost.yaml`, `group_vars/all.yaml` for variables to be applied to localhost and the all group
- When a playbook is executed, the variables from these files will be merged with those from the inventory file (if any)

Example - host_vars and group_vars

```
all:
  hosts:
    server-0:
      ansible_host: 192.168.0.100
      ansible_connection: ssh
      ansible_user: fred
      ansible_password: fred
    server-1:
      ansible_host: 192.168.0.101
      ansible_connection: ssh
      ansible_user: fred
      ansible_password: fred
    server-2:
      ansible_host: 192.168.0.102
      ansible_connection: ssh
      ansible_user: fred
      ansible_password: fred
      db_user: barney
      db_password: barney
      db_name: inventory
```

Example - host_vars and group_vars

inventory.yaml

```
all:
  hosts:
    server-0: {}
    server-1: {}
    server-2: {}
```

group_vars/all.html

```
ansible_connection: ssh
ansible_user: fred
ansible_password: fred
```

host_vars/server-0.yaml

```
ansible_host: 192.168.0.100
```

hosts_vars/server-1.yaml

```
ansible_host: 129.168.0.101
```

hosts_vars/server-2.yaml

```
ansible_host: 129.168.0.102
db_user: barney
db_password: barney
db_name: inventory
```

Roles

- Playbook is used to configure a host for a particular purpose
  - Eg. database server, bastion host, CMS/WordPress, etc.
  - Eg. Need to configure MySQL server multiple times in different projects
- A role is a set of task applied to a host to configure the host for a specific purpose
  - Provide an intuitive role name to identify the purpose of the tasks
  - Hide complex configuration by providing a way to reuse playbooks
  - Share know-how by reusing tasks
- It is independent and can be incorporated into any playbooks

Using a Role

- Search from Ansible Galaxy, filter by roles
  - https://galaxy.ansible.com/
- Community, can publish the roles you have created
  - Require a GitHub account
- Documentation on how to use the role in your playbook
  - Version and OS platform
- Roles are executed before tasks
- Instructions to download and install a role
  - `ansible-galaxy`

```
ansible-galaxy install geerlingguy.docker
```

Install Docker role
from Ansible Galaxy

https://galaxy.ansible.com/geerlingguy/docker

Example - Using the Docker

```
site.yaml
- name: Install Docker
  host: docker_host
  pre_tasks:
  - name: Create user
    user:
      name: '{{ docker_user }}'
      password: '{{ docker_user_password }}'
  roles:
  - role: roles/docker
    vars:
      docker_users:
      - '{{ docker_user }}'
  tasks:
  - name: Enable remote access
    lineinfile:
      path: /lib/systemd/system/docker.service
      regexp: '^ExecStart='
      line: ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock-
H=tcp://0.0.0.0:2375
  - name: Restart Docker daemon
    ...
```

Like task but executed before roles or tasks

Role is downloaded to roles directory

Role variable(s) to configure Docker

```
- name: Restart Docker daemon
   systemd:
     name: docker
     daemon_reload: true
     state: restarted
```

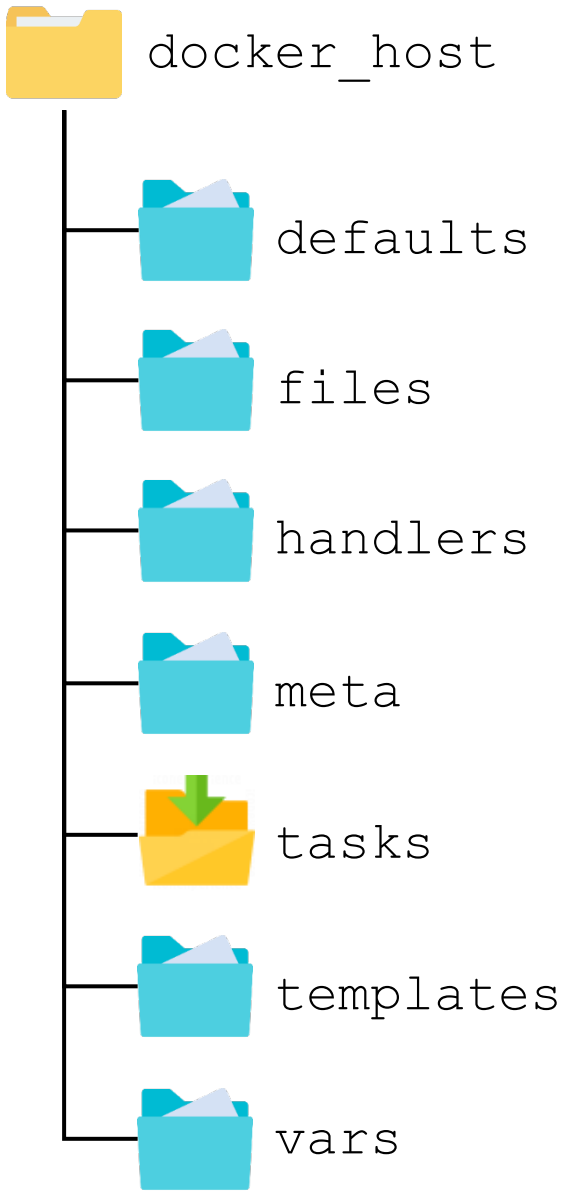
Note: an alternative way to provision Docker host is to use docker-machine https://docs.docker.com/machine

Creating Role

- A set of pre defined directory that can be generated with ansible-galaxy

```
ansible-galaxy init docker_host
```

- Directory uses
  - `defaults` - default variables
  - `files` - files, eg configuration, to be copied over to the target
  - `handlers` - for handlers
  - `tasks` - contains all the task to be performed by the role. When a role is applied, Ansible will perform the list of task listed here
  - `templates` - dynamically generated files
  - `vars` - other variables, override defaults
  - `meta` - metadata for the role



Example - Docker Host Role

```
default/main.yaml
---
package_list:
- apt-transport
- ca-certificates
- software-properties-common
- curl
- openssh-server
docker_gpg_key: https://download.docker.com/linux/ubuntu/gpg
docker_repo: “deb [arch=amd64] https://download.docker.com/linux/ubuntu
focal stable”
```

Example - Docker Host Role

tasks/main.yaml

```
---
- name: Install packages
  apt:
    update_cache: yes
    name: '{{ item }}'
    state: latest
  loop: '{{ package_list}}'
- name: Add Docker GPG key
  apt-key:
    url: '{{ docker_gpg_key }}'
    state: present
- name: Add Docker repository
  apt_repository:
    repo: '{{ docker_repo }}'
    state: present
```

User defined variables

```
- name: Install Docker
  apt:
    update_cache: yes
    name: docker-ce
    state: latest
- name: Create non-root Docker user
  user:
    name: '{{ docker_user }}'
    groups: [ 'docker' ]
  when: docker_user is defined
- name: Check if SSH key is present
  stat:
    path: '{{ docker_ssh_key }}'
  register: docker_ssh_key is defined
- name: Add SSH key to user
  authorized_key:
    user: '{{ docker_user }}'
    key: '{{ lookup('file', docker_ssh_key }}'
  when: ssh_key.stat.exists
```

From https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04

Example - Using Docker Host Role

```
site.yaml
- name: Install Docker
  hosts: a_docker_host
  roles:
  - role: roles/docker_host
    vars:
      docker_user: fred
      docker_ssh_key: /path/to/public/key
```

Example - HTTP/WebDAV

```
terraform {
  required_version = ">= 0.15.0"
  required_providers = {
    digitalocean = { ... }
  }

  backend http {
    address = "http://myserver.com/myproj"
    lock_address = "http://myserver.com/myproj"
    unlock_address = "http://myserver.com/myproj"
    username = "fred"
    password = "fred"
  }
}
```

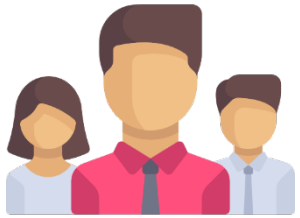
The endpoint for creating, retrieving and updating the state file

Associate an identifier to endpoint if the server is used by many projects

Specify username and password if endpoint is uses HTTP Basic authentication

Locking is optional. Configure locking if server supports WebDAV

Example - HTTP/WebDAV



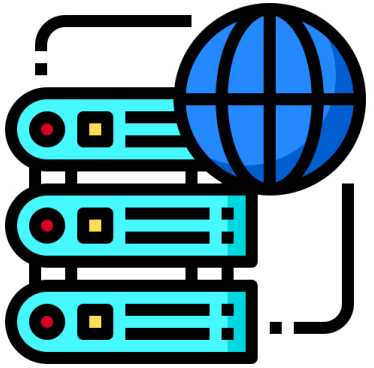
Other members

```
LOCK /myproject
Authorization: Basic btoa("username:password")
<lock details in the body>
```



```
POST /myproject?lockid=...
Authorization: Basic btoa("username:password")
<update state file>

UNLOCK /myproject?lockid=...
Authorization: Basic btoa("username:password")
```



HTTP/WebDAV

Developer