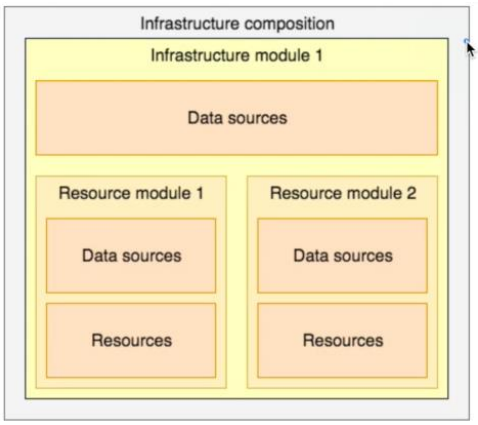
This is Prod Grade terraform EKS Course:

- Generally, all the other courses terraform code is not scalable. The standard software pattern still applies to terraform and in this course, we are making use of the Façade Pattern so we can build clean, scalable, reusable, manageable and extensive terraform code.



- Contrary to we what we see on the internet the below code architecture can barely scale for Production Usage.

```
── us-east-1
|   ├── prod
|   |   ├── backend.config
|   |   ├── main.tf  <------ # main entrypoint
├── modules # <------ modules contain reusable code
|   ├── compute
|   |   ├── ec2
|   |   ├── ec2_key_pair
|   |   ├── security_group
|   |   └── ssm
|   ├── network
|   |   ├── route53
|   |   |   ├── hosted_zone
|   |   |   └── record
|   |   └── vpc
|   └── storage
|       ├── dynamodb
|       ├── efs
|       └── s3
```

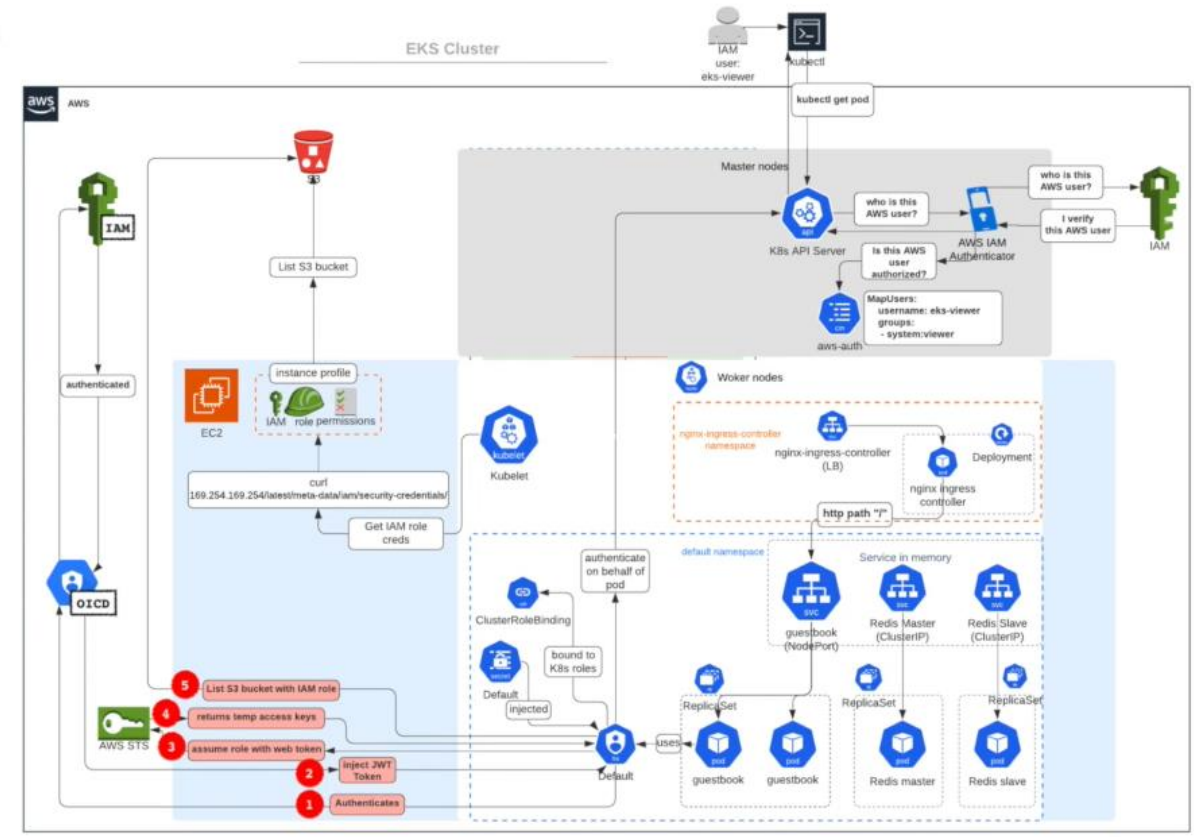
- Here we will cover more advanced and scalable design in chapter Three Layered Modules. Given below.

```
├── composition
|   ├── us-east-1
|   |   ├── prod
|   |   |   └── main.tf # <---main entrypoint calling infra-module
|   |   └── staging                                    |
|   └── us-west-2                                      |
├── infra_module                                       |
|   ├── app # this will wrap multiple resource modules  <--
|   ├── bastion                                |
|   ├── elb                                    |
|   └── rds                                    |
└── resource_module                            |
    ├── compute                                |
    |   ├── ec2 # <------------------------------|
    |   ├── ec2_key_pair # <---------------------|
    |   ├── security_group  # <------------------|
    |   └── ssm # <------------------------------|
    ├── network
    |   ├── route53
    |   |   ├── hosted_zone
    |   |   └── record
    |   └── vpc
    └── storage
        ├── dynamodb
        ├── efs
        └── s3
```

- In the above two terraform structure the latter is more scalable and prod usage like. The former is most used across but it's not efficient. In the above the terraform structure the resources under modules in former is same as the resource_module in latter. Similarly, composition layer in latter is like the region(us-east-1) in former under which the *main entry file of terraform* is located. The only change we have in both the approaches is the infra_module, this contains app, elb, eks, rds etc., this is where we can use new purpose-oriented resource name for a particular purpose. This is where the *Façade Pattern* comes in where the main use is to wrap and abstract away multiple sub-systems in the resource_module so that it can provide a simple interface to the client.
- In the former approach the main.tf is *main entry point* which knows too much about the sub systems(underlying resources in modules) which is not desirable. Also, another downside in former is if you want to add a bastion host latter on we need to update the main.tf and it becomes a large code as we keep adding on. Instead in the latter approach we create an Abstraction in the infra_module where we add the bastion resources and change the related resource_module like ec2, ec2_key_pair and security group. Instead of referencing the resources from the main.tf we can rub them in bastion infra_module. So only thing the main.tf needs to know is the bastion interface. This is what the section is all about and we going to create the EKS cluster using the AWS EKS best practices.

We will cover:

- encrypting K8s secrets and EBS volumes
- AWS identity authentication & authorization into K8s cluster
- adding taints and labels to K8s worker nodes from Terraform
- enabling master node's logging
- pod-level AWS IAM role (IRSA)
- Cluster Autoscaler
- customizing EKS worker node's userdata script to auto-mount EFS all using terraform code.

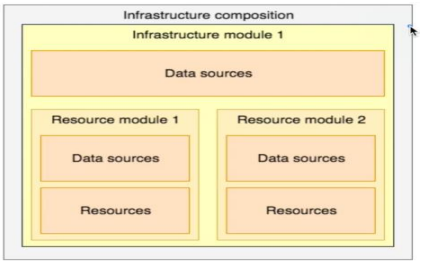


- The *above picture is mainly about the IRSA architecture* which we are going to create using terraform.
- Here is the code excerpt for adding AWS IAM roles to aws-auth configmap using Terraform:
    - map_roles = var.authenticate_using_role == true ? concat(var.map_roles, [
    - {
    - rolearn = "arn:aws:iam::${data.aws_caller_identity.this.account_id}:role/${var.role_name}"
    - username = "k8s_terraform_builder"
    - groups = ["system:masters"] # k8s group name
    - },
    - {
    - rolearn = "arn:aws:iam::${data.aws_caller_identity.this.account_id}:role/Developer" # create Developer IAM role in Console
    - username = "k8s-developer"
    - groups = ["k8s-developer"]
    - },
    -]) : var.map_roles
- The above terraform snippet helps in *authentication* of the user role using the map_roles and allows entry into the K8S cluster. After this the *authorization* is done using the **cluster role binding** in which we gonna assign the above created group in the K8S cluster using YAML file. So the guy having the developer role can do certain things inside the K8S cluster. IMP NOTE: Check these concepts in the AWS EKS best practices course and get clear idea.
- Note: Simply put, authentication is the process of verifying who someone is, whereas authorization is the process of verifying what specific applications, files, and data a user has access to.

Using the 3-layered approach to terraform:

Ref: https://www.terraform-best-practices.com/key-concepts

The below picture represents the best scalable approach of terraform.



CONS:

Why this approach works best in regards with the normal approach?

- Below is the 2 layered normal terraform structure which we can create.
    - ├── modules # <------ modules are organized under this dir
    - │ ├── compute
    - │ │ ├── ec2
    - │ │ ├── ec2_key_pair
    - │ │ ├── security_group
    - │ │ └── ssm
    - │ ├── network
    - │ │ ├── route53
    - │ │ │ ├── hosted_zone
    - │ │ │ └── record
    - │ │ └── vpc
    - │ └── storage
    - │ ├── dynamodb
    - │ ├── efs
    - │ └── s3
    -
- In the above approach in case if we want to scale we need to a lot of details in the main.tf which makes it complex.
- Based on single class responsibility principle, you shouldn't mix IAM role and security group resources in `modules/compute/ec2/main.tf` file.
- **>Don't Break Single Class Responsibility Principle**

  ├── modules

  │ ├── compute

  │ │ ├── ec2 # <----- you shouldn't add IAM role and SG in EC2's main.tf

- **>Don't Break Abstraction and Encapsulation (main() shouldn't be nosy)**
-
- How about adding them in `us-east-1/prod/main.tf`? That means everytime you want to create new AWS resources, you will be appending to `us-east-1/prod/main.tf`, hence the `main.tf` file gets bigger and less modularized, less manageable.

  ── us-east-1

  │ ├── prod
  │ │ ├── backend.config
  │ │ ├── main.tf # <----- add IAM role and SG for bastion and app servers here respectively?

- Usually main() should be as simple and short as possible. If we keep adding multiple EC2, multiple security groups, multiople IAM roles, multiple R53 A records in `us-east-1.prod.main.tf`, it'll look like this, which won't be modularized and not cohesive based around AWS resources

  ```
  # in /us-east-1/prod/main.tf
    module "bastion_ec2" {
    source = "./modules/ec2/"
  }

  module "app_ec2" {
    source = "./modules/ec2/"
  }

  module "bastion_security_group" {}

  module "app_security_group" {}

  module "bastion_private_ip" {}

  module "app_private_ip" {}

  module "elb_a_record" {}

  module "rds_kms_key" {}

  # keep adding more and more
  .
  ```

  This goes against keeping one `.tf` file short and simple
- **>separate unrelated resources from each other by placing them in separate compositions reduces the risk if something goes wrong**

PROS: (3 layered model):

- In the 3 layered approach we have C-Composition, I-Infrastructure Modules, R-Resource Modules. With this approach the main.tf is maintained small and each additional component is added to the infra_module and called in the main.tf, within the resource_module we can enter the required instances and their relative key_pair and SG accordingly.
- **>Use Class-like infra-module to Abstract Away details (main() shouldn't be nosy)**
- So intead, you should be wrapping resource modules with another modules (infra module) to group resources together.
- This abstraction layer is similar to __[Facade pattern](https://sourcemaking.com/design_patterns/facade)__, meaning a class wrapping bunch of classes to create a simple interface to a client:
- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface
- For example:
  - `bastion` infra module (i.e. EC2, SG, SSM, IAM role)
  - `app` infra module (i.e. EC2, SG, SSM parameter store, IAM role)
  - `elb` infra module (i.e. ELB, Route53 record, etc)
  - `rds` infra module (i.e. RDS, KMS, SSM parameter store)

- **It will look like this as in `2.2_three_layered_modules/`:**

```
├── composition
│   ├── us-east-1
│   │   ├── prod
│   │   │   └── main.tf # <---main entrypoint calling infra-module
│   │   └── staging                                    |
│   └── us-west-2                                      |
├── infra_module                                       |
│   ├── app # this will wrap multiple resource modules  <--
│   ├── bastion                                 |
│   ├── elb                                   |
│   └── rds                                   |
└── resource_module                               |
    ├── compute                                 |
    │   ├── ec2 # <-------------------------------|
    │   ├── ec2_key_pair # <----------------------|
    │   ├── security_group  # <-------------------|
    │   └── ssm # <-------------------------------|
    ├── network
    │   ├── route53
    │   │   ├── hosted_zone
    │   │   └── record
    │   └── vpc
    └── storage
        ├── dynamodb
        ├── efs
        └── s3
```

- And in the top-level `composition/us-east-1/prod/main.tf`, __all it needs to know is each module's path and dependent input variables__

```
    ├── composition
    │   ├── us-east-1
    │   │   ├── prod
    │   │   │   └── main.tf  # <------ here
    
    # in composition/us-east-1/prod/main.tf
    module "bastion" {
      source = "../infra_module/bastion/"  # <-- kinda calling bastion.main(), instead of calling bastion.create_ec2(), bastion.create_security_group(), and so on
    }
    
    module "app" {
      source = "../infra_module/app/"
    }
    
    module "elb" {
      source = "../infra_module/elb/"
    }
    
    module "rds" {
      source = "../infra_module/rds/"
    }
```

- And in `infra_module/bastion/main.tf`, it contains __low-level details__ of all the auxiliary AWS resources needed to make bastion EC2 functional (i.e. IAM role, security group, ec2 key paier, etc)

```
# in infra_module/bastion/main.tf  # <-- kinda like mutliple helper functions called inside bastion.main()
module "ec2" {
  source = "../resource_module/compute/ec2"
}

module "security_group" {
  source = "../resource_module/compute/security_group"
}

module "ssm_parameter_store" {
  source = "../resource_module/compute/ssm"
}

module "route53_record" {
  source = "../resource_module/network/route53/record"
}
```

- with this, `composition/us-east-1/prod/main.tf` will be acting like a client (i.e. main()) and doesn't need to know low-level details about `module "bastion"` implementation of EC2, SG, SSM, IAM role, etc, therefore achieving __abstraction and modularization__.