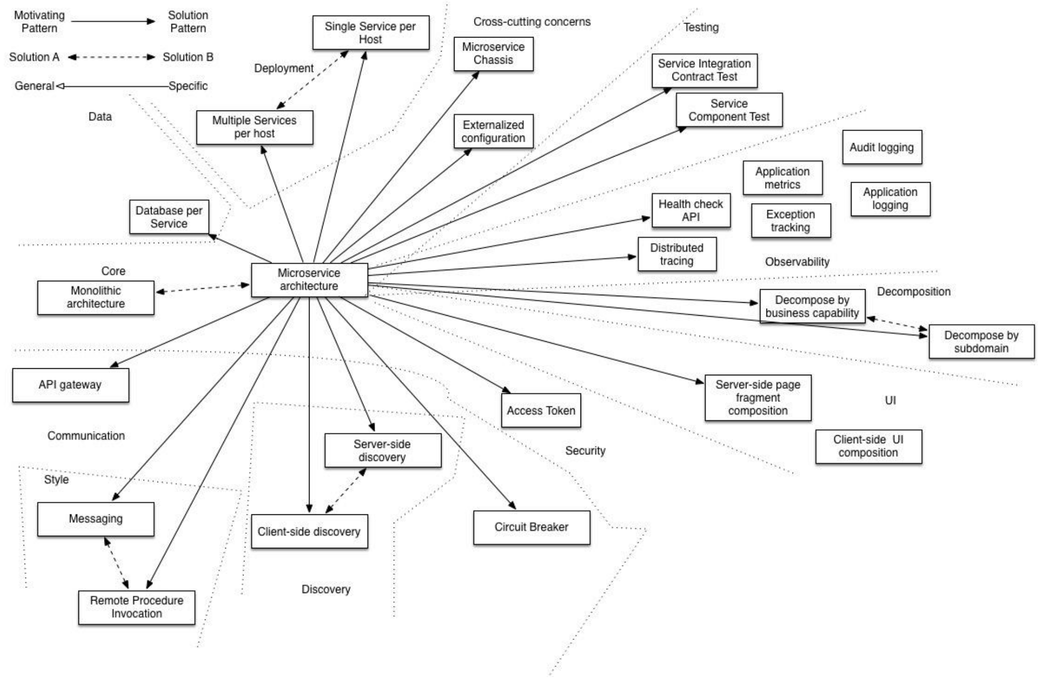
Click Here To Watch Video Tutorial

1) What exactly are Microservices?
2) What is Continuous Integration?
3) What is Continuous Delivery & Deployment?
4) What is Infrastructure as Code & Terraform?
5) AWS Tools for CICD & Microservices
6) Microservices CI/CD Demo with AWS + Terraform

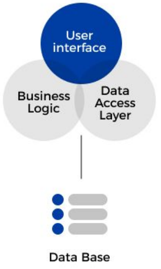
Microservices CI/CD With AWS + Terraform

What is Microservices?

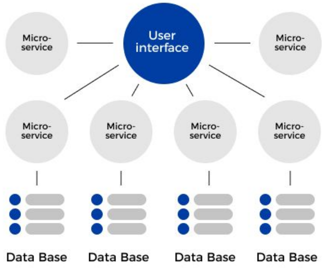
The microservices architecture is a design approach to build a single application as a set of small services. Each service runs in its own process and communicates with other services through a well-defined interface using a lightweight mechanism, typically an HTTP-based application programming interface (API). Microservices are built around business capabilities; each service is scoped to a single purpose. You can use different frameworks or programming languages to write microservices and deploy them independently, as a single service, or as a group of services.



MONOLITHIC ARCHITECTURE

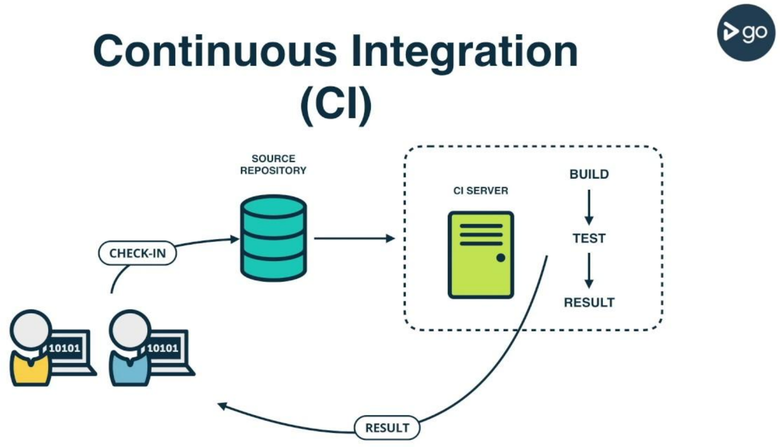


MICROSERVICE ARCHITECTURE



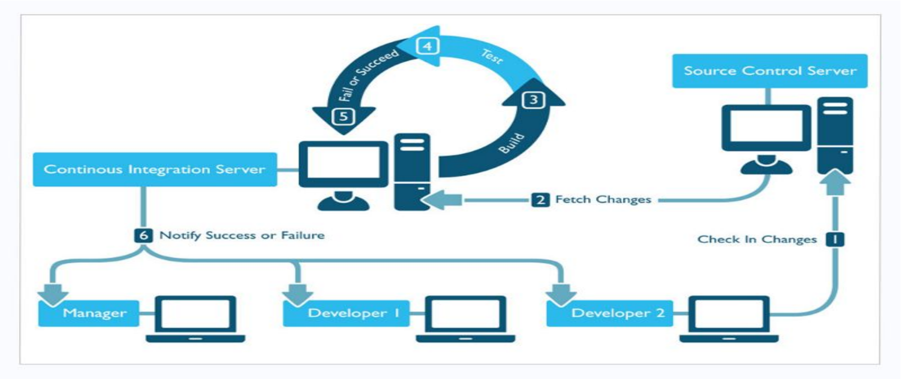
What is Continuous Integration?

Continuous integration (CI) is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.



In more simple words:

Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. It's a primary DevOps best practice, allowing developers to frequently merge code changes into a central repository where builds and tests then run.



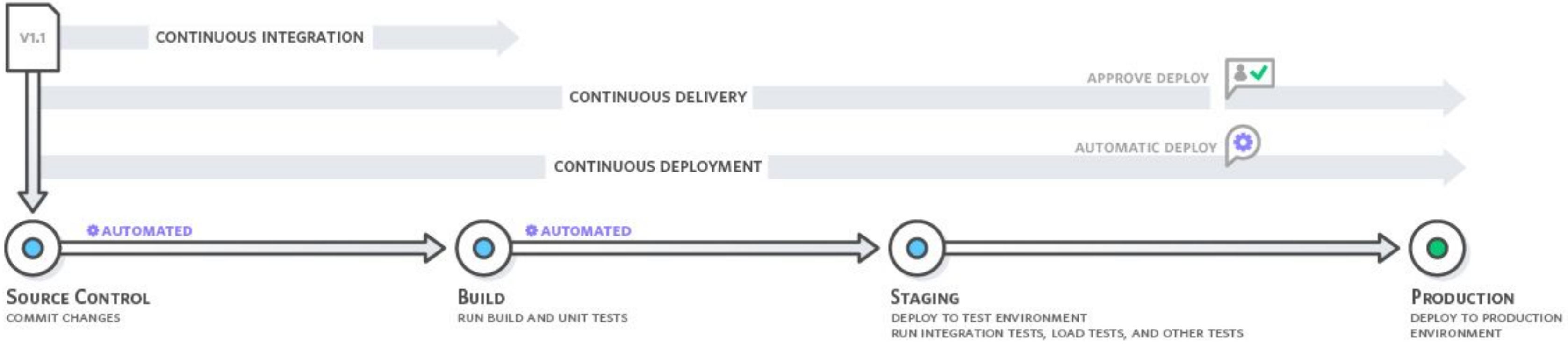
Continuous Delivery & Deployment

Continuous delivery is a software development practice where code changes are automatically built, tested, and prepared for a release to production. It expands upon continuous integration by deploying all code changes to a testing environment and/or a production environment after the build stage. When continuous delivery is implemented properly, developers will always have a deployment-ready build artifact that has passed through a standardized test process.

Continuous Deployment (CD) is a software release process that uses automated testing to validate if changes to a codebase are correct and stable for immediate autonomous deployment to a production environment.

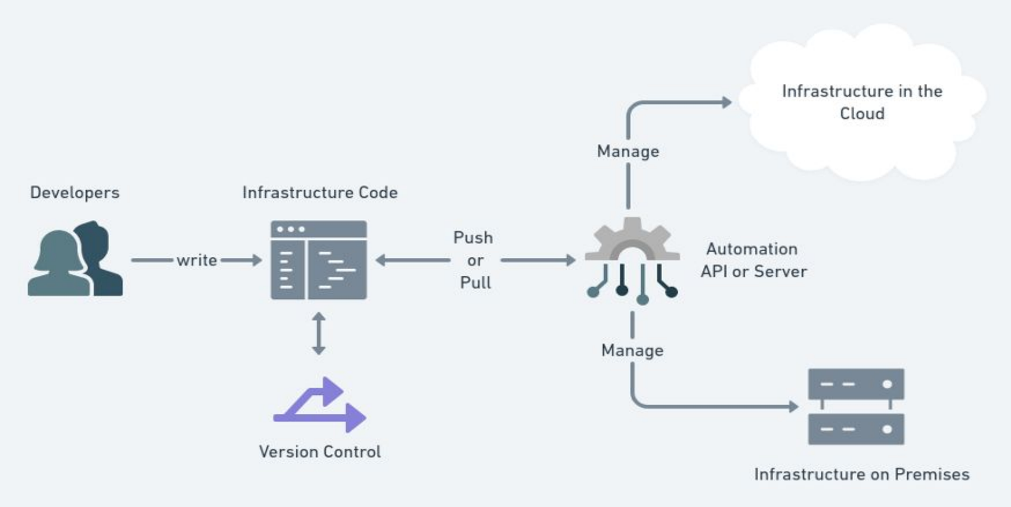
Continuous Delivery vs. Continuous Deployment

With continuous delivery, every code change is built, tested, and then pushed to a non-production testing or staging environment. There can be multiple, parallel test stages before a production deployment. The difference between continuous delivery and continuous deployment is the presence of a manual approval to update to production. With continuous deployment, production happens automatically without explicit approval.



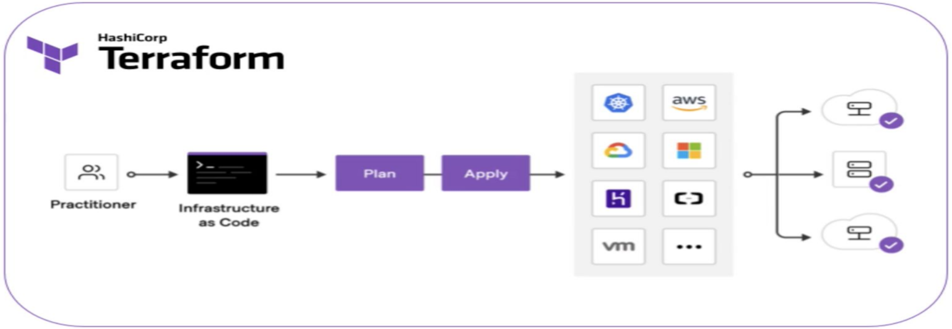
What is Infrastructure As Code (IaC)?

Infrastructure as code (IaC) is a practice in which infrastructure is provisioned and managed using code and software development techniques, such as version control and continuous integration. The cloud's API-driven model enables developers and system administrators to interact with infrastructure programmatically, and at scale, instead of needing to manually set up and configure resources. Thus, engineers can interface with infrastructure using code-based tools and treat infrastructure in a manner similar to how they treat application code. Because they are defined by code, infrastructure and servers can quickly be deployed using standardized patterns, updated with the latest patches and versions, or duplicated in repeatable ways.



What is Terraform?

Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. This includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc. Terraform can manage both existing service providers and custom in-house solutions.



Watch Terraform Basics [FREE FULL COURSE] Here

AWS Tools for CICD & Microservices

Continuous Integration and Continuous Delivery

AWS CodePipeline: AWS CodePipeline is a continuous integration and continuous delivery service for fast and reliable application and infrastructure updates. CodePipeline builds, tests, and deploys your code every time there is a code change, based on the release process models you define. This enables you to rapidly and reliably deliver features and updates.

AWS CodeBuild: AWS CodeBuild is a fully managed build service that compiles source code, runs tests, and produces software packages that are ready to deploy. With CodeBuild, you don't need to provision, manage, and scale your own build servers. CodeBuild scales continuously and processes multiple builds concurrently, so your builds are not left waiting in a queue.

AWS CodeDeploy: AWS CodeDeploy automates code deployments to any instance, including Amazon EC2 instances and on-premises servers. AWS CodeDeploy makes it easier for you to rapidly release new features, helps you avoid downtime during application deployment, and handles the complexity of updating your applications.

AWS CodeCommit: AWS CodeCommit is a fully-managed source control service that makes it easy for companies to host secure and highly scalable private Git repositories. You can use CodeCommit to securely store anything from source code to binaries, and it works seamlessly with your existing Git tools.

AWS CodeStar: AWS CodeStar enables you to quickly develop, build, and deploy applications on AWS. AWS CodeStar provides a unified user interface, enabling you to easily manage your software development activities in one place. With AWS CodeStar, you can set up your entire continuous delivery toolchain in minutes, allowing you to start releasing code faster.

Microservices

Amazon Elastic Container Service: Amazon Elastic Container Service (ECS) is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances.

AWS Lambda: AWS Lambda lets you run code without provisioning or managing servers. With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code and Lambda takes care of everything required to run and scale your code with high availability.

Microservice CI/CD Demo using

AWS CodePipeline

+

AWS CodeCommit

+

AWS CodeBuild

+

AWS ECS

GitHub Project Url: Click Here

Node App + Docker

Run Below command

To Make Docker Build

docker image build -t <image_name>:tag .

(“.” refer to current directory)

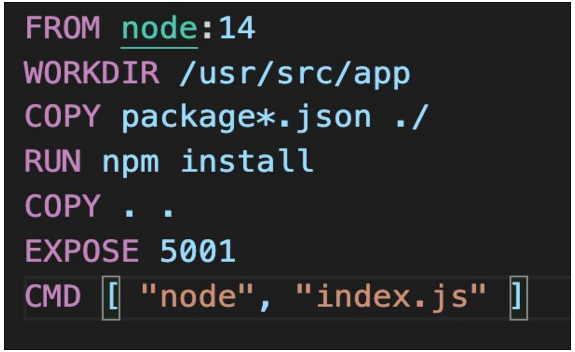
e.g.

```
docker build --tag docker-node-app:latest .
```

Test image running fine or not:

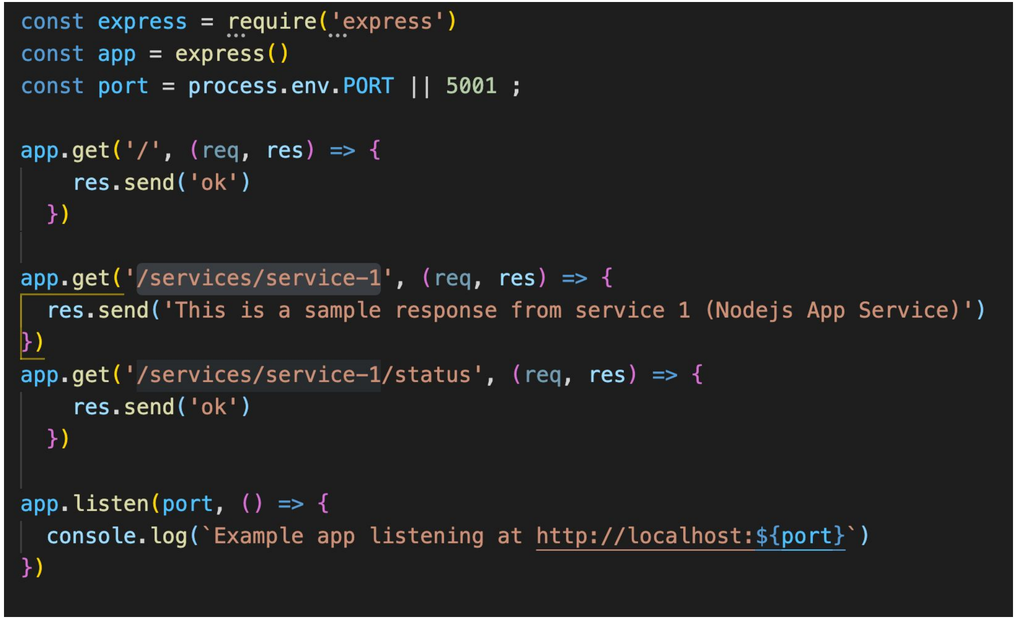
```
docker run -d -p 5001:5001  docker-node-app:latest
```

Dockerfile



```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 5001
CMD [ "node", "index.js" ]
```

index.js



```
const express = require('express')
const app = express()
const port = process.env.PORT || 5001 ;

app.get('/', (req, res) => {
    res.send('ok')
  })

app.get('/services/service-1', (req, res) => {
  res.send('This is a sample response from service 1 (Nodejs App Service)')
})
app.get('/services/service-1/status', (req, res) => {
    res.send('ok')
  })

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Python App + Docker

Run Below command

To Make Docker Build

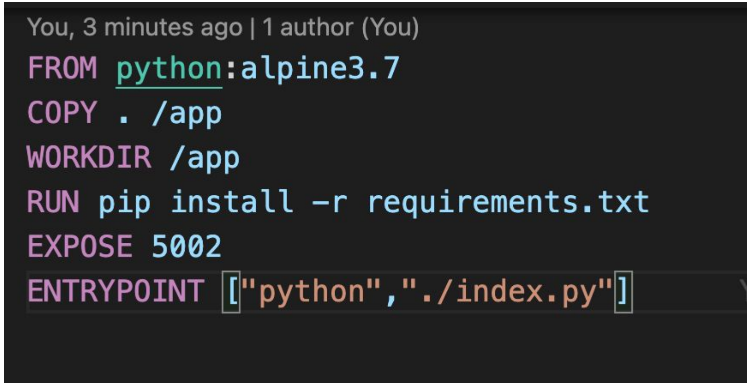
e.g.

```
docker build --tag docker-python-app:latest .
```

Test image running fine or not:

```
docker run -d -p 5002:5002  docker-python-app:latest
```

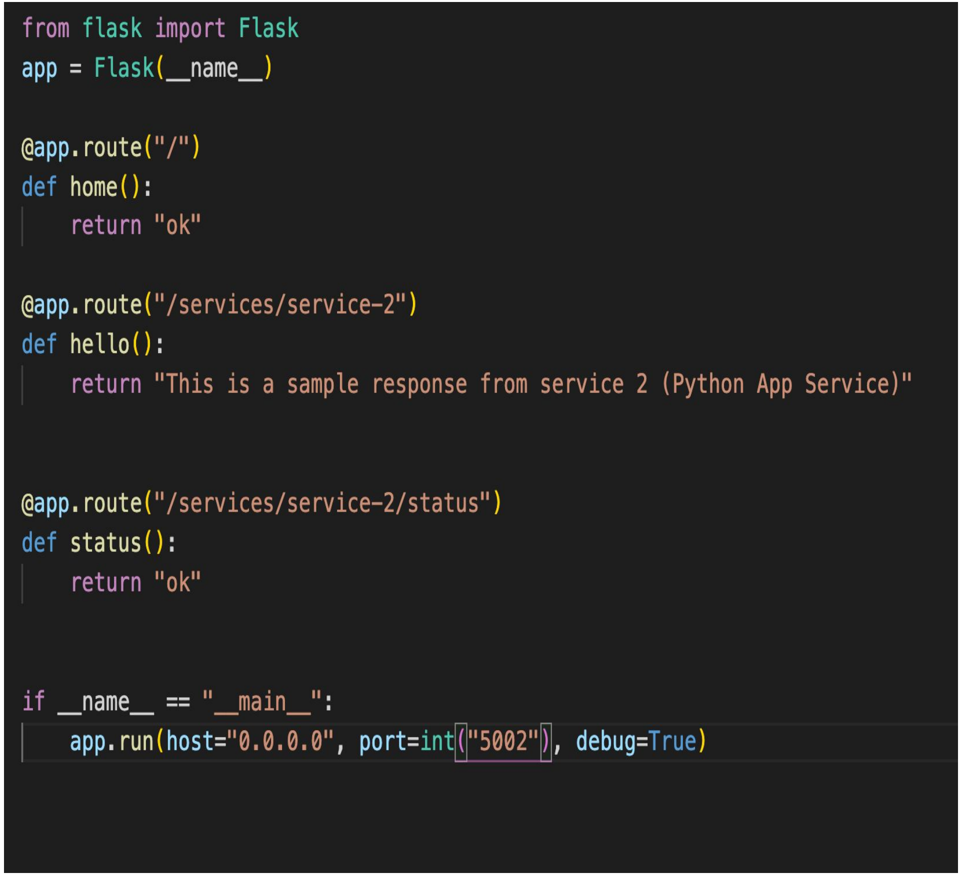
Dockerfile



You, 3 minutes ago | 1 author (You)

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5002
ENTRYPOINT ["python","./index.py"]
```

index.py



```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "ok"

@app.route("/services/service-2")
def hello():
    return "This is a sample response from service 2 (Python App Service)"


@app.route("/services/service-2/status")
def status():
    return "ok"


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=int("5002"), debug=True)
```

Go App + Docker

Run Below command

To Make Docker Build

```
docker image build -t <image_name>:tag .
```

(“.” refer to current directory)

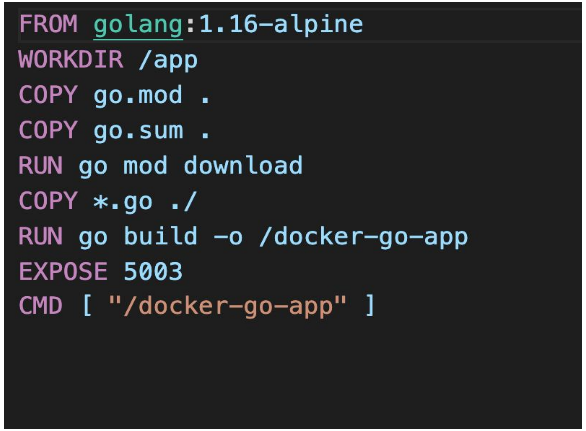
e.g.

```
docker build --tag docker-go-app:latest .
```

Test image running fine or not:

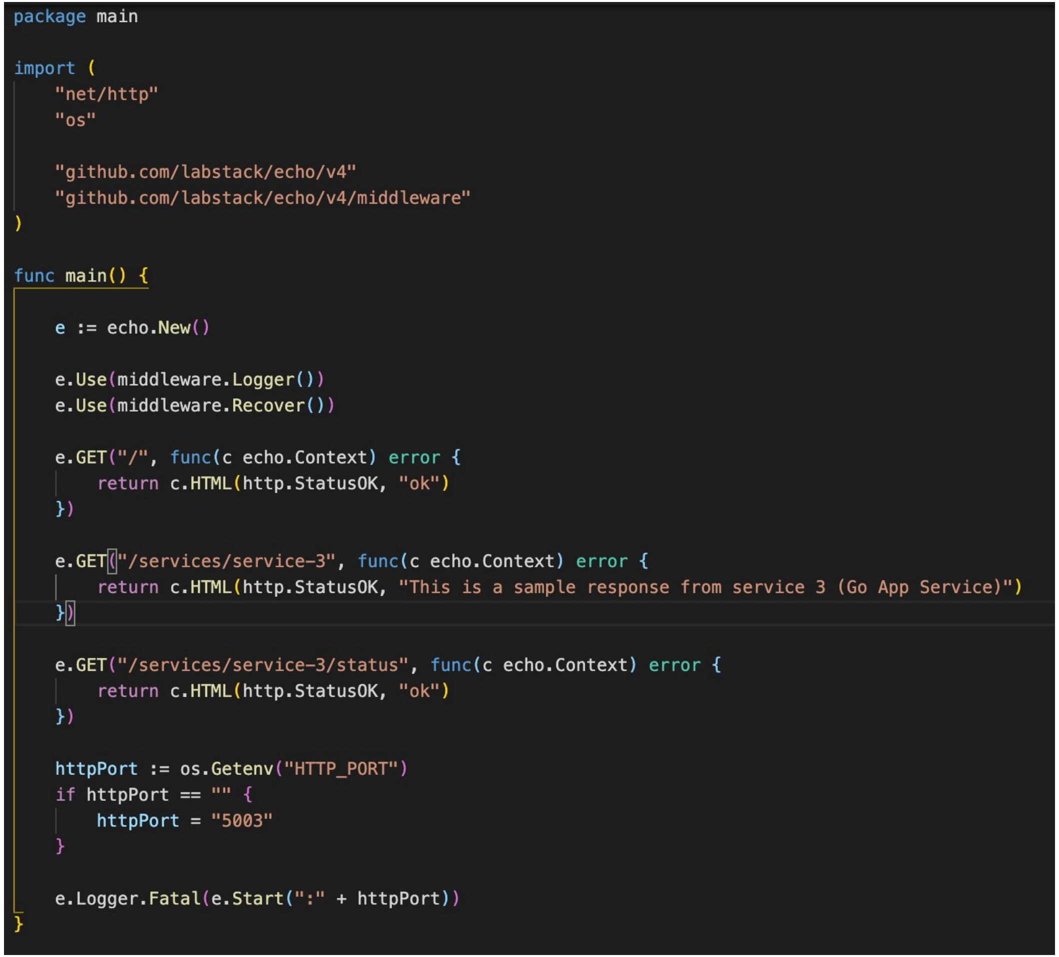
```
docker run -d -p 5003:5003  docker-go-app:latest
```

Dockerfile



```dockerfile
FROM golang:1.16-alpine
WORKDIR /app
COPY go.mod .
COPY go.sum .
RUN go mod download
COPY *.go ./
RUN go build -o /docker-go-app
EXPOSE 5003
CMD [ "/docker-go-app" ]
```

main.go



```go
package main

import (
    "net/http"
    "os"

    "github.com/labstack/echo/v4"
    "github.com/labstack/echo/v4/middleware"
)

func main() {

    e := echo.New()

    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    e.GET("/", func(c echo.Context) error {
        return c.HTML(http.StatusOK, "ok")
    })

    e.GET("/services/service-3", func(c echo.Context) error {
        return c.HTML(http.StatusOK, "This is a sample response from service 3 (Go App Service)")
    })

    e.GET("/services/service-3/status", func(c echo.Context) error {
        return c.HTML(http.StatusOK, "ok")
    })

    httpPort := os.Getenv("HTTP_PORT")
    if httpPort == "" {
        httpPort = "5003"
    }

    e.Logger.Fatal(e.Start(":" + httpPort))
}
```

Step by Step Terraform Integration with AWS

Step 1: Input Variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

For this demo, we have declared all required parameters , since source codes are in AWS CodeCommit, we have mentioned repo names and branch, and existing ECS Cluster name and linked services, that will be used in main terraform code/resources

Variables.tf

```
variable "aws_region" {
 description = "AWS region to launch servers."
 default     = "us-west-2"
}
variable "env" {
 description = "Targeted Deployment environment"
 default     = "dev"
}
variable "nodejs_project_repository_name" {
 description = "Nodejs Project Repository name to
connect to"
 default     = "nodeapp"
}
variable "nodejs_project_repository_branch" {
 description = "Nodejs Project Repository branch to
connect to"
 default     = "master"
}
```

```
variable "python_project_repository_name" {
 description = "Python Project Repository name to
connect to"
 default     = "pythonapp"
}

variable "python_project_repository_branch" {
 description = "Python Project Repository branch to
connect to"
 default     = "master"
}

variable "golang_project_repository_name" {
 description = "Go Lang Project Repository name to
connect to"
 default     = "goapp"
}
variable "golang_project_repository_branch" {
 description = "Python Project Repository branch to
connect to"
 default     = "master"
}
```

```
variable "artifacts_bucket_name" {
 description = "S3 Bucket for storing artifacts"
 default     = "sandip-cicd-artifacts-bucket"
}

variable "aws_ecs_cluster_name" {
 description = "Target Amazon ECS Cluster Name"
 default     = "MicroServicesCluster"
}

variable "aws_ecs_node_app_service_name" {
 description = "Target Amazon ECS Cluster NodeJs
App Service name"
 default     = "nodeAppService"
}

variable "aws_ecs_python_app_service_name" {
 description = "Target Amazon ECS Cluster Python
App Service name"
 default     = "pythonAppService"
}

variable "aws_ecs_go_app_service_name" {
 description = "Target Amazon ECS Cluster Go App
Service name"
 default     = "goAppService"
}
```

Step by Step Terraform Integration with AWS

Step 2: Add terraform required version, setup backend (if any required) , add provider
The Second step would be to add terraform basic codes in the main.tf and add aws provider as follow:

In this code we have mentioned :

1) What is the minimum required terraform version

2) Setup AWS S3 as backend , mentioned bucket name

3) Set AWS as the Provider and setup default region

main.tf

```
terraform {
 required_version = ">= 0.12"
 backend "s3" {
   bucket = "terraform-demo-sandip"
   key = "terraform.tfstate"
   region = "us-west-2"
 }
}


provider "aws" {
 region = var.aws_region
}
```

Run below command to initialize the project:
Terraform init

Step by Step Terraform Integration with AWS

Step 3: Add required IAM Roles and policies (Permissions) Click Here for IAM Policy Reference

In this steps we need to add all required access for AWS CodeBuild Project and AWS CodePipeline Project and declare as a resource in main.tf file

```
resource "aws_iam_role" "containerAppBuildProjectRole" {
 name = "containerAppBuildProjectRole"

 assume_role_policy = <<EOF
{
 "Version": "2012-10-17",
 "Statement": [
  {
    "Effect": "Allow",
    "Principal": {
      "Service": "codebuild.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }
 ]
}
EOF
}

resource "aws_iam_role_policy" "containerAppBuildProjectRolePolicy" {
 role = aws_iam_role.containerAppBuildProjectRole.name
policy = <<POLICY
{
....
..
}
```

Note: Full Policy statement you can find in main GitHub repo with this doc 👍

```
resource "aws_iam_role" "apps_codepipeline_role" {
 name = "apps-code-pipeline-role"

 assume_role_policy = <<EOF
{
 "Version": "2012-10-17",
 "Statement": [
  {
    "Effect": "Allow",
    "Principal": {
      "Service": "codepipeline.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }
 ]
}
EOF
}
resource "aws_iam_role_policy"
"apps_codepipeline_role_policy" {
 name = "apps-codepipeline-role-policy"
 role = aws_iam_role.apps_codepipeline_role.id

 policy = <<EOF
....
...
 }
```

Step by Step Terraform Integration with AWS

Step 4: **Create AWS CodeBuild Project** **(Click here For the AWS CodeBuild Reference)**

In this step we will add all references to create a AWS CodeBuild project in order to create updated Docker images and push to AWS ECR, for build to run we must specify the buildspec.yml file in either in terraform codes or in the source codes. Important here: **privileged_mode must be true to make docker builds**

```
resource "aws_codebuild_project" "containerAppBuild" {
 badge_enabled  = false
 build_timeout  = 60
 name           = "container-app-build"
 queued_timeout = 480
 service_role   = aws_iam_role.containerAppBuildProjectRole.arn
 tags = {
   Environment = var.env
 }

 artifacts {
   encryption_disabled = false
   # name                 = "container-app-code-${var.env}"
   # override_artifact_name = false
   packaging = "NONE"
   type      = "CODEPIPELINE"
 }

 environment {
   compute_type                = "BUILD_GENERAL1_SMALL"
   image                       = "aws/codebuild/standard:5.0"
   image_pull_credentials_type = "CODEBUILD"
   privileged_mode             = true
   type                        = "LINUX_CONTAINER"
 }

 logs_config {
   cloudwatch_logs {
     status = "ENABLED"
   }

   s3_logs {
     encryption_disabled = false
     status              = "DISABLED"
   }
 }

 source {

   # buildspec           = data.template_file.buildspec.rendered

   git_clone_depth     = 0

   insecure_ssl        = false

   report_build_status = false

   type                = "CODEPIPELINE"

 }

}
```

Step by Step Terraform Integration with AWS

Step 5: Create AWS Codepipeline Project (AWS CodePipeline Reference)
In this step we create the final AWS CodePipeline Project, which will handle the main CI/CD Flow

Here we are:

1) Mentioning the role of the codepipeline to make sure this codepipeline project have necessary permissions
2) Specifying Artifact Storage Location

```
resource "aws_codepipeline" "python_app_pipeline" {
 name     = "python-app-pipeline"
 role_arn = aws_iam_role.apps_codepipeline_role.arn
 tags = {
   Environment = var.env
 }

 artifact_store {
   location = var.artifacts_bucket_name
   type     = "S3"
 }
```

Step by Step Terraform Integration with AWS

Source Stage

Inside pipeline resource we have to specify stages, such as Source, Build and Deploy etc

Here we are:

1) Mentioning our first stage of codepipeline i.e. Source stage

2) Specified the AWS CodeCommit repository name and branch

3) Most importantly, the “provider” field is set as CodeCommit, All valid values are here: Full Provider List

4) Configuration option differ provider to provider, so best is to check above list before proceeding

5) For AWS CodeCommit provider reference Check here

```
stage {
  name = "Source"

  action {
    category = "Source"
    configuration = {
      "BranchName"        = var.python_project_repository_branch
      # "PollForSourceChanges" = "false"
      "RepositoryName"     = var.python_project_repository_name
    }
    input_artifacts = []
    name            = "Source"
    output_artifacts = [
      "SourceArtifact",
    ]
    owner    = "AWS"
    provider  = "CodeCommit"
    run_order = 1
    version   = "1"
  }
}
```

Step by Step Terraform Integration with AWS

Build Stage

In this stage we specify all build related configuration

Here we are:

1) Mentioning our second stage of the pipeline

2) Proving all the environment variables required for this build project, for non confidential we can use type = "PLAINTEXT" and for secure parameters we should use AWS Parameter store or secret manager

3) Make sure provider = “CodeBuild” for this stage

4) All Configuration for CodeBuild Provider is here

```
stage {
  name = "Build"

  action {
    category = "Build"
    configuration = {
      "EnvironmentVariables" = jsonencode(
       [
        {
          name  = "environment"
          type  = "PLAINTEXT"
          value = var.env
        },
        {
          name  = "AWS_DEFAULT_REGION"
          type  = "PLAINTEXT"
          value = var.aws_region
        },
        {
          name  = "AWS_ACCOUNT_ID"
          type  = "PARAMETER_STORE"
          value = "ACCOUNT_ID"
        },
        {
          name  = "IMAGE_REPO_NAME"
          type  = "PLAINTEXT"
          value = "nodeapp"
        },
        {
          name  = "IMAGE_TAG"
          type  = "PLAINTEXT"
          value = "latest"
        },
        {
          name  = "CONTAINER_NAME"
          type  = "PLAINTEXT"
          value = "pythonAppContainer"
        },
       ] )
      "ProjectName" =
aws_codebuild_project.containerAppBuild.name
    }
    input_artifacts = [
      "SourceArtifact",
    ]
    name = "Build"
    output_artifacts = [
      "BuildArtifact",
    ]
    owner     = "AWS"
    provider  = "CodeBuild"
    run_order = 1
    version   = "1"
  }
}
```

Step by Step Terraform Integration with AWS

Deploy Stage

In this stage we are deploying code changes to targeted AWS ECS Service

Here we are:

1) Mentioning our third and final stage of the pipeline, make sure provider as as “ECS”

2) Here in this stage mentioning what is the cluster name and service name where the new image should get deployed

3) In this we have to mention what is the file name where we have the deployment related information , the file name here is: **imagedefinitions.json, this file get generated in AWS CodeBuild and passed as artifact**

4) All Configuration for AWS ECS Provider is here

```
stage {
  name = "Deploy"

  action {
    category = "Deploy"
    configuration = {
      "ClusterName" =
var.aws_ecs_cluster_name
      "ServiceName" =
var.aws_ecs_python_app_service_name
      "FileName"    =
"imagedefinitions.json"
      #"DeploymentTimeout" = "15"
    }
    input_artifacts = [
      "BuildArtifact",
    ]
    name             = "Deploy"
    output_artifacts = []
    owner            = "AWS"
    provider         = "ECS"
    run_order        = 1
    version          = "1"
  }
 }
}
```

Good luck!

I hope you'll use this knowledge and build awesome solutions.
GitHub Project Url: [Click Here]

If any issue contact me in Linkedin:
https://www.linkedin.com/in/sandip-das-developer/