🚀 WORKSHOP DEVOPS – 8 HORAS

AWS para Iniciantes, LocalStack, Deploy e Pipeline com GitHub Actions

---

📌 Público-alvo

Iniciantes em DevOps, estudantes e desenvolvedores que querem aprender o fluxo completo: **desenvolver → testar → empacotar → subir na AWS → automatizar com CI/CD**.

📌 Pré-requisitos

- Conhecimento básico de programação (qualquer linguagem)
- Git instalado
- Docker instalado
- Conta na AWS (opcional, pois usaremos LocalStack também)

🕑 CRONOGRAMA DETALHADO – 8 HORAS

🧩 MÓDULO 1 — Fundamentos DevOps + AWS (1h)

Objetivos:

- Apresentar o mindset DevOps

- Entender como AWS se conecta a práticas modernas de entrega

Conteúdos:

- O que é DevOps: cultura, CI/CD, automação e infraestrutura como código
- Visão geral de AWS: EC2, S3, VPC, IAM, CloudWatch, ECS, Lambda
- Onde o iniciante mais erra: permissões IAM, redes, pipelines e containers
- Diferença entre deploy manual x automatizado

Mini exercício:

- Criar uma conta IAM "DevOpsWorkshop"
- Explicar o princípio de **least privilege**

🧩 MÓDULO 2 — Fundamentos de Linux para DevOps (1h)

Objetivos:

Ensinar o básico necessário para administrar servidores em ambientes Linux.

Conteúdos:

- Estrutura de diretórios Linux
- Gerenciamento de serviços (`systemctl`, `service`)
- Comandos essenciais: `ls`, `cd`, `cat`, `grep`, `tail -f`, `chmod`, `chown`
- Configurando firewall (UFW / Security Groups)
- Logging e troubleshooting (nginx, system logs)

Hands-on:

-
- Criar diretórios, editar arquivos e iniciar um serviço
- Instalar NGINX e abrir porta 80

🧩 MÓDULO 3 — Explorando o LocalStack (1h15)

Objetivos:

Ensinar como simular serviços AWS localmente para testes e desenvolvimento.

Conteúdos:

- O que é LocalStack e por que empresas usam
- Serviços suportados: S3, Lambda, API Gateway, EC2 fake, CloudWatch
- Instalação com Docker
- AWS CLI configurado com LocalStack

Hands-on:

- Criar um bucket S3 no LocalStack
- Fazer upload de arquivos
- Subir um serviço simples via LocalStack
- Criar uma API Gateway + Lambda fake

Resultado:

Participante entende como testar recursos AWS **sem gastar dinheiro e sem risco**.

🧩 MÓDULO 4 — Deploy do meu site na AWS (1h45)

Objetivo:

Fazer deploy de um site real em uma EC2 utilizando Linux + NGINX + domínio (opcional).

Conteúdos:

- Criando uma instância EC2
- Escolhendo AMI, tipo de máquina, permissões e chaves
- Configuração do Security Group
- Instalando dependências no servidor
- Configurando NGINX para servir a aplicação
- Deploy manual: copiar arquivos via SCP
- Deploy automatizado (preview)

- Comparação: LocalStack x AWS real

Hands-on:

- Conectar em uma instância via SSH

- Criar uma instância EC2
- Subir um site estático (HTML) ou uma API simples
- Configurar NGINX
- Validar via browser

Resultado:

O aluno vê **seu site rodando na AWS**.

🧩 MÓDULO 5 — Docker para DevOps (1h)

Objetivo:

Criar a imagem Docker do projeto que será usada no pipeline CI/CD.

Conteúdos:

- O que é Docker e por que DevOps depende dele
- Dockerfile otimizado
- Docker build, run, logs
- Multi-stage build (se quiser avançado)
- Enviar imagem para Docker Hub ou ECR

Hands-on:

- Criar Dockerfile da aplicação
- Gerar imagem e testar localmente
- Publicar imagem no Docker Hub (opcional)

🧩 MÓDULO 6 — Criando meu primeiro pipeline CI/CD – GitHub Actions (1h)

Objetivo:

Configurar CI/CD completo com teste → build Docker → deploy automático.

Conteúdos:

- O que é GitHub Actions
- Como funciona um workflow: jobs, steps, triggers
- Secrets (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY)
- Templates de pipelines
- Pipeline real:

```
name: Deploy to AWS

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Run tests
      run: npm test

    - name: Build Docker image
      run: docker build -t meuapp:latest .

    - name: Deploy via SSH
      uses: appleboy/ssh-action@v1.0.0
      with:
        host: ${{ secrets.SERVER_IP }}
        username: ubuntu
        key: ${{ secrets.SERVER_SSH_KEY }}
        script: |
          docker pull meuapp:latest
          docker stop app || true
          docker rm app || true
          docker run -d --name app -p 80:80 meuapp:latest
```

Hands-on:

- Configurar pipeline real

- Pipeline dispara ao fazer push
- Servidor EC2 recebe nova versão automaticamente

Resultado final:

Alunos têm um **pipeline real, funcional e automatizado**.

🧩 MÓDULO 7 — Observabilidade e Troubleshooting rápido (30 min)

Conteúdos:

- Logs da aplicação
- Monitoramento básico com CloudWatch
- Como diagnosticar erros comuns de pipeline, Docker e EC2

Hands-on:

- Identificar erro proposital em um deploy e corrigir

🧩 ENCERRAMENTO – Recap + Certificado + Dicas de Carreira (15 min)

🎁 ENTREGÁVEIS DO TREINAMENTO

- Repositório GitHub com o pipeline pronto
- Dockerfile de exemplo

- Script de deploy
- Guia de comandos Linux
- Template EC2 com instruções
- Cheatsheet de AWS + LocalStack
- PDF com arquitetura final da solução

Links de suporte:
- Acronimos, dicionário de palavras de infra
- Treinamentos de infra - Linuxtips
- Comandos Básicos Linux
- Livro Docker para prograadores - Gomex
- Como criar um pipeline no Github Actions
- Documentação do LocalStack
- Descomplicando docker
- Girus CLI (Laboratorios)

Tutoriais práticos:
1- Como instalar o docker
2- Como Criar uma maquina linux na AWS
   - Video Tutorial: Live 26 - Como Criar e acessar um servidor Linux na AWS?
3- Tutoriais localstack
4- Descomplicando Github Actions

Tutorial de comandos para rodar o nginx:

Para subir um servidor **Nginx** usando Docker e expô-lo na porta **8080** da sua máquina local, utilizamos o comando:

docker run --rm -d --name nginx -p 8080:80 nginx:1.28.0

Depois disso, ao acessar:

http://localhost:8080

você verá a página padrão do Nginx.

Como alterar o arquivo **index.html** exibido pelo Nginx

Existem 3 maneiras principais:

1. Usando volume para substituir o **index.html**

   Crie um arquivo **index.html** no diretório atual e execute:

   *docker run -v $(pwd)/index.html:/usr/share/nginx/html/index.html *
   --rm -d --name nginx -p 8080:80 nginx:1.28.0

   Assim, o arquivo local será usado como página inicial dentro do container.

2. Editando diretamente dentro do container

   Acesse o terminal do container:

   docker exec -it nginx bash

   Dentro dele, edite o arquivo localizado em: */usr/share/nginx/html/index.html*

3. Copiando um arquivo do seu computador para dentro do container

   Se você preferir alterar localmente e depois enviar para o container, use:

   docker container cp $(pwd)/index.html nginx:/usr/share/nginx/html

Tutorial fazer o upload de arquivos para o bucket S3:

Este guia mostra como simular a AWS localmente utilizando **LocalStack** e fazer upload de arquivos para um bucket S3, usando o **AWS CLI** apontando para o endpoint local.

Subindo o LocalStack com Docker

Para iniciar o LocalStack e simular serviços da AWS localmente, execute:

```
docker run -it -d --name localstack \
  -e LAMBDA_EXECUTOR=local \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -p 4566:4566 \
```

```
  -p 4510-4559:4510-4559 \
  localstack/localstack:latest
```

Instalando o AWS CLI (caso ainda não tenha)

Você pode instalar via pip:

pip install awscli

Configurando o AWS CLI para usar o LocalStack

Antes de tudo, configure credenciais **dummy** para evitar interagir acidentalmente com uma conta real da AWS:

aws configure

Pode usar valores fictícios, por exemplo:

- AWS Access Key ID: `test`
- AWS Secret Access Key: `test`
- Region: `us-east-1`
- Output: `json`

Agora exporte o endpoint que aponta para o LocalStack:

export AWS_ENDPOINT_URL=http://localhost:4566

Criando um bucket S3 no LocalStack

Com tudo configurado, crie o bucket:

```
aws s3api create-bucket \
  --bucket devopsdays \
  --region us-east-1 \
  --endpoint-url $AWS_ENDPOINT_URL
```

Fazendo upload de um arquivo para o bucket

Agora basta enviar o arquivo desejado:

```
aws s3 cp ./myfile.txt s3://devopsdays \
  --endpoint-url $AWS_ENDPOINT_URL
```

Se o comando executar sem erros, o arquivo já estará dentro do bucket simulado no LocalStack.

Tutorial de como cirar uma lambda function e acessar via api gateway

Criar uma IAM Role para a Lambda

Mesmo que o LocalStack não valide permissões IAM reais, precisamos criar uma role fictícia para que a Lambda seja aceita.

Crie um arquivo trust-policy.json:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Agora crie a role no LocalStack:

```
awslocal iam create-role \
  --role-name lambda-exec-role \
  --assume-role-policy-document file://trust-policy.json
```

Pegue o ARN retornado, algo como:

arn:aws:iam::000000000000:role/lambda-exec-role

🐍 2. Criar a função Lambda com Python 3.12

Crie o arquivo lambda.py:

```
def lambda_handler(event, context):
    print("hello world")


    return {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "body": "{\"message\": \"It's working!\"}"
    }
```

Compacte o arquivo:

```
zip function.zip lambda.py
```

Crie a Lambda:

```
awslocal lambda create-function \
  --function-name apigw-lambda \
  --runtime python3.12 \
```

--handler lambda.lambda_handler \

--memory-size 128 \

--zip-file fileb://function.zip \

--role arn:aws:iam::000000000000:role/lambda-exec-role

🌐 3. Criar a REST API

awslocal apigateway create-rest-api --name "API Gateway Lambda Integration"

Guarde o REST_API_ID.

📁 4. Obter o recurso raiz /

awslocal apigateway get-resources --rest-api-id <REST_API_ID>

Saída típica:


```
{
  "items": [
    {
      "id": "u53af9hm83",
      "path": "/"
    }
  ]
}
```

Guarde o RESOURCE_ID (id da raiz).

🔧 5. Criar método GET na raiz /

```
awslocal apigateway put-method \
  --rest-api-id <REST_API_ID> \
  --resource-id <RESOURCE_ID> \
  --http-method GET \
  --authorization-type "NONE"
```

🔗 6. Integrar com a Lambda

```
awslocal apigateway put-integration \
  --rest-api-id <REST_API_ID> \
  --resource-id <RESOURCE_ID> \
  --http-method GET \
  --type AWS_PROXY \
  --integration-http-method POST \
  --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:000000000000:function:apigw-lambda/invocations \
  --passthrough-behavior WHEN_NO_MATCH
```

🚀 7. Deploy da API

**awslocal apigateway create-deployment **

**--rest-api-id <REST_API_ID> **

--stage-name dev

📞 8. Invocar a API (na raiz /)

Formato oficial:

curl http://<REST_API_ID>.execute-api.localhost.localstack.cloud:4566/dev/

Saída esperada:

{"message": "It's working!"}

🔄 URL alternativa (caso seu DNS não resolva)

curl http://localhost:4566/_aws/execute-api/<REST_API_ID>/dev/

Links das imagens docker usadas nesse tutorial:
https://hub.docker.com/_/nginx

https://hub.docker.com/r/localstack/localstack

- **https://github.com/Jonta-Sancar/dod**

- https://github.com/Jonta-Sancar/dod-fsa

Gravacao dos comandos no meu terminal enquanto eu replicava essa documentação:
https://www.awesomescreenshot.com/video/47283466?key=4290a2972cbb1bad25c1a311debde690