Lab 6: Multi-Environments (Teams) and Multi-Services in Kubernetes

> **Author:** Badr TAJINI - DevOps Data for SWE - ESIEE - 2025

This lab will guide you through the process of setting up multiple environments using separate AWS accounts and managing multiple microservices in Kubernetes. You'll learn how to isolate resources, configure applications differently across environments, and deploy interconnected services using Kubernetes.

Objectives:

- Set up multiple AWS accounts for different environments (dev, stage, prod).
- Configure OpenTofu workspaces to manage multiple deployments.
- Deploy the same application with different configurations in multiple environments.
- Deploy multiple microservices (frontend and backend) in Kubernetes.
- Understand service discovery in Kubernetes.

Prerequisites:

- Completed Lab 5
- AWS account
- Local Kubernetes cluster (Docker Desktop with Kubernetes enabled or Minikube)
- Installed and configured tools: Git, Docker, `kubectl`, OpenTofu, `npm`, Node.js, `aws-cli`

Part 1: Setting Up Multiple AWS Accounts

In the real world, you don't usually deploy everything into a single environment. Instead, you use multiple isolated environments for development, staging, and production. We'll use separate AWS accounts to keep our environments truly separated.

1.1. Why Multiple Accounts?

Using multiple AWS accounts provides a strong level of isolation between your environments. This approach helps with:

- **Isolating Tests:** Making sure your tests don't mess with your production setup.

- **Isolating Products/Teams:** Allowing each team or product to have its own dedicated resources.
- **Increasing Resiliency:** Preventing a single issue in one account from affecting everything.

1.2. Creating Child Accounts

Instead of manually creating each AWS account, we'll use AWS Organizations to create them as child accounts under a management account.

Steps:

1. **Prepare your main AWS account:**
    - Make sure the root AWS account you created in Lab 1, doesn't have resources deployed that would conflict with this lab.
    - You can achieve this by running `tofu destroy` on OpenTofu modules you may have deployed and manually undeploy resources from the EC2 console that you have deployed with Ansible, Bash, etc.
2. **Create a new project folder:**
    - Navigate to the root of your `devops_base` folder and make sure you are on the `main` branch.

```
$ cd ~/devops_base
$ git checkout main
$ git pull origin main
```

    - Create a new folder for this lab.

```
$ mkdir -p td6/tofu/live/child-accounts
$ cd td6/tofu/live/child-accounts
```

3. **Create the main configuration file:**
    - Create `main.tf` with the code below. Be sure to fill in **your** email addresses, these must be unique for each account and they must be valid email addresses because AWS will require you to verify them.

```
# td6/tofu/live/child-accounts/main.tf (Example 6-1)
provider "aws" {
  region = "us-east-2"  # Or your preferred region
}

module "child_accounts" {
  source = "github.com/your_github_name/devops-base//td6/tofu/modules/aws-
organization"

  create_organization = true  # Set to false if you already enabled AWS
Organizations

  dev_account_email   = "username+dev@email.com" # Replace with your email
alias
  stage_account_email = "username+stage@email.com" # Replace with your email
alias
  prod_account_email  = "username+prod@email.com" # Replace with your email
alias
}
```

- **Explanation:**
  - `source`: Specifies the location of the OpenTofu module to use.
  - `create_organization`: Set to `true` to enable AWS Organizations if you haven't already.
  - `dev_account_email`, `stage_account_email`, `prod_account_email`: Your root user email addresses for each account. You'll need to use different, unique email addresses for each. You can use aliases if your email provider supports them.

4. **Create output file:**
   - Create an `outputs.tf` file in `td6/tofu/live/child-accounts` to expose the created role ARNs:

```
# td6/tofu/live/child-accounts/outputs.tf (Example 6-2)
output "dev_role_arn" {
description = "The ARN of the IAM role you can use to manage dev from mgmt"
value       = module.child_accounts.dev_role_arn
}

output "stage_role_arn" {
description = "The ARN of the IAM role you can use to manage stage from mgmt"
value       = module.child_accounts.stage_role_arn
}

output "prod_role_arn" {
description = "The ARN of the IAM role you can use to manage prod from mgmt"
value       = module.child_accounts.prod_role_arn
}
```

- **Explanation:**
  - These outputs will give you the ARNs of the IAM roles you need to use to manage the dev, stage, and prod accounts, respectively, from your management account.

5. **Deploy the accounts:**
   - Run the usual commands to deploy your infrastructure:

```
tofu init
tofu apply
```

   - After the deployment is complete, note down the `dev_role_arn`, `stage_role_arn`, and `prod_role_arn` output values. You'll need these later.

1.3 Accessing Child Accounts

Now that you have created the child accounts, you will need to learn to access them programatically using AWS profiles and roles. There are many different ways to assume an IAM role. For example, follow these instructions to assume an IAM role in the AWS Web Console.

Steps:

1. **Set up AWS profiles:**
   - Edit your AWS configuration file, usually located at `~/.aws/config`, and add a new profile for each child account.

```
# ~/.aws/config (Example 6-3)
[profile dev-admin]
role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole" #
Replace with your dev_role_arn output
credential_source = Environment

[profile stage-admin]
role_arn = "arn:aws:iam::333333333333:role/OrganizationAccountAccessRole" #
Replace with your stage_role_arn output
credential_source = Environment

[profile prod-admin]
role_arn = "arn:aws:iam::444444444444:role/OrganizationAccountAccessRole" #
Replace with your prod_role_arn output
credential_source = Environment
```

- **Explanation:**
  - `[profile dev-admin]`: This creates a profile named `dev-admin` that you can use to authenticate to your development account. You can choose any name for the profile.
  - `role_arn`: The ARN of the IAM role you noted down earlier.
  - `credential_source`: Tells the AWS CLI where to get the credentials (from your current environment variables).

2. **Verify your setup:**
   - Test that the profiles work by executing the following command and observing the account ID from the output.

Example 6-4. Use the AWS CLI with a profile

```
```bash
AWS_PROFILE=dev-admin aws sts get-caller-identity
```

```json
{
    "UserId": "<USER>",
    "Account": "<ACCOUNT_ID>",
    "Arn": "<ARN>"
}
```
```

- Repeat this for the `stage-admin` and `prod-admin` profiles, and ensure the correct account ID is output for each. The `get-caller-identity` command returns information about the authenticated user, so if you configured the profile correctly, `ACCOUNT_ID` should be the ID of the dev account, and ARN should be the ARN of the dev IAM role.

Part 2: Managing Deployments with OpenTofu Workspaces

We'll use OpenTofu workspaces to deploy the same application into multiple environments, each with its own configuration.

2.1. Understanding Workspaces

OpenTofu workspaces let you manage multiple deployments of the same infrastructure configuration. Each workspace has its own state file, representing a separate set of resources.

2.2. Copying the Lambda Sample App

Let's get the lambda app from the previous lab, and put it in the correct folder for this lab.

Steps:

1. **Create directory:**
   - From the root of the `devops_base` directory, create the directory for the lambda sample app:

     ```
     $ cd ~/devops_base
     $ mkdir -p td6/tofu/live
     ```

2. **Copy the lambda sample app:**
   - Copy the lambda sample app from the previous lab to `td6/tofu/live`.

     ```
     $ cp -r td5/scripts/tofu/live/lambda-sample td6/tofu/live/
     ```

3. **Copy the test endpoint module:**
   - Create the modules directory and copy the test endpoint module into it.

     ```
     $ mkdir -p td6/tofu/modules
     $ cp -r td5/scripts/tofu/modules/test-endpoint td6/tofu/modules
     ```

2.3. Preparing the Lambda Sample App

Let's make some changes to the lambda-sample module to make it work in multiple environments

Steps:

1. **Disable the backend configuration:**
   - Inside the copied `td6/tofu/live/lambda-sample` module, delete the `backend.tf` file.
   - For this lab we want the state file to be local and not stored remotely.
2. **Modify the lambda function:**
   - Update the source code of the lambda function to show the environment name in the response. Replace the existing code with the example below:

```
// td6/tofu/live/lambda-sample/src/index.js (Example 6-5)
exports.handler = (event, context, callback) => {
  callback(null, {statusCode: 200, body: `Hello from
${process.env.ENV_NAME}!`});
};
```

3. **Configure the environment variable:**
   - Modify `td6/tofu/live/lambda-sample/main.tf` to use the workspace name in the environment variable. ***Example 6-6**. Set `NODE_ENV` dynamically to the value of `terraform.workspace`*

```
# td6/tofu/live/lambda-sample/main.tf # (Example 6-6)
module "function" {
  source = "github.com/your_github_name/devops-base//td3/tofu/modules/lambda"

  # ... (other params omitted) ...

  environment_variables = {
    NODE_ENV = "production"
    ENV_NAME = terraform.workspace
  }
}
```

     - **Explanation:**
       - `terraform.workspace` : This is a built-in OpenTofu variable that automatically captures the name of the current workspace.

- In OpenTofu, you can use workspaces to manage multiple deployments of the same configuration. Each workspace has its own state file, so it represents a separate copy of all the infrastructure, and each workspace has a unique name, which is returned by `terraform.workspace`.

2.4. Deploying to Multiple Environments

Now let's use workspaces to deploy your app to different environments.

Steps:

1. **Initialize OpenTofu:**
   - Navigate to the `lambda-sample` folder and initialize OpenTofu:

     ```
     $ cd td6/tofu/live/lambda-sample
     $ tofu init
     ```

2. **Create workspaces:**
   - Create new workspaces for development, staging, and production:

     ```
     tofu workspace new development
     tofu workspace new staging
     tofu workspace new production
     ```

3. **Deploy to each workspace:**
   - Switch to the `development` workspace, and use the profile to deploy the infrastructure:

     ```
     tofu workspace select development
     AWS_PROFILE=dev-admin tofu apply
     ```

   - You should see a plan for creating a lambda function, API Gateway route, and so on. Type `yes` if everything looks good.
   - When the command is complete, take note of the `api_endpoint` output. You will need this value in the next step.
   - Test the URL:

```
curl <DEV_URL>
```

- You should see `Hello from development!` in the output.
- Repeat the previous steps for both `staging` and `production`.

```
tofu workspace select staging
AWS_PROFILE=stage-admin tofu apply
```

```
tofu workspace select production
AWS_PROFILE=prod-admin tofu apply
```

  - Be sure to also test the URL to see `Hello from staging!` and `Hello from production!` respectively.

2.5. Using Different Configurations for Different Environments

Often, you need to configure your app differently in different environments. Let's use JSON configuration files for this.

Steps:

1. **Create configuration files:**
   - Create the config directory:

```
$ mkdir -p src/config
```

   - Create a `development.json` file in `td6/tofu/live/lambda-sample/src/config` folder:

```
# td6/tofu/live/lambda-sample/src/config/development.json (Example 6-7)
{
  "text": "dev config"
}
```

   - Create similar files `staging.json` and `production.json`, with different text values, in the same directory.

```
# td6/tofu/live/lambda-sample/src/config/staging.json (Example 6-7)
{
  "text": "stage config"
}
```

```
# td6/tofu/live/lambda-sample/src/config/production.json (Example 6-7)
{
  "text": "prod config"
}
```

2. **Update Lambda function code:**

   - Modify `index.js` to load the correct config file based on the environment variable and use the text field.

```
// td6/tofu/live/lambda-sample/src/index.js (Example 6-8)
const config = require(`./config/${process.env.ENV_NAME}.json`)

exports.handler = (event, context, callback) => {
  callback(null, {statusCode: 200, body: `Hello from ${config.text}!`});
};
```

     - **Explanation:**
       - The app will load the JSON file based on the `ENV_NAME` environment variable, that has been set to `terraform.workspace`, as defined in `main.tf`.

3. **Redeploy:**

   - Switch to the `development` workspace and redeploy.

```
tofu workspace select development
AWS_PROFILE=dev-admin tofu apply
```

   - Test the URL:

```
curl <DEV_URL>
```

     You should see `Hello from dev config!` in the output.

   - Repeat the previous step for `staging` and `production` and verify the expected text is returned.

2.6. Closing your child accounts

When you're done testing the multiple environments, it's good practice to delete them.

Steps:

1. Commit your code to git.
2. Undeploy your infrastructure from each of the workspaces.

```
tofu workspace select development
AWS_PROFILE=dev-admin tofu destroy
tofu workspace select staging
AWS_PROFILE=stage-admin tofu destroy
tofu workspace select production
AWS_PROFILE=prod-admin tofu destroy
```

3. Destroy your accounts.

```
cd ../child-accounts
tofu destroy
```

Part 3: Deploying Microservices in Kubernetes

Let's move on to deploying multiple services in Kubernetes. You'll set up a frontend and a backend service that communicate with each other.

3.1. Creating the Backend Service

The backend service will handle data and expose it via an API.

Steps:

1. **Copy the sample app:**
   - Go back to the root of `devops_base` and copy the existing `sample-app` folder to `td6/scripts/sample-app-backend`:

```
$ cd ~/devops_base
$ cp -r td5/scripts/sample-app td6/scripts/sample-app-backend
```

- Copy the Kubernetes Deployment and Service configurations from `td4/scripts/kubernetes` to `td6/scripts/sample-app-backend/`:

```
$ cp td3/scripts/kubernetes/sample-app-deployment.yml td6/scripts/sample-app-backend/
$ cp td3/scripts/kubernetes/sample-app-service.yml td6/scripts/sample-app-backend/
```

2. **Update the backend app:**

   - Modify `td6/scripts/sample-app-backend/app.js`:

```
// td6/scripts/sample-app-backend/app.js (Example 6-9)
app.get('/', (req, res) => {
  res.json({ text: "backend microservice" });
});
```

   - Update `td6/scripts/sample-app-backend/package.json`:

```
// td6/scripts/sample-app-backend/package.json (Example 6-10)
{
  "name": "sample-app-backend",
  "version": "0.0.1",
  "description": "Backend app for 'DevOps Labs!'"
}
```

   - Modify `td6/scripts/sample-app-backend/sample-app-deployment.yml`:

```
# td6/scripts/sample-app-backend/sample-app-deployment.yml (Example 6-11)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app-backend-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sample-app-backend-pods
  template:
    metadata:
      labels:
        app: sample-app-backend-pods
    spec:
      containers:
        - name: sample-app-backend
          image: sample-app-backend:0.0.1
          ports:
            - containerPort: 8080
          env:
            - name: NODE_ENV
              value: production
```

- Modify `td6/scripts/sample-app-backend/sample-app-service.yml`:

```
# td6/scripts/sample-app-backend/sample-app-service.yml (Example 6-12)
apiVersion: v1
kind: Service
metadata:
  name: sample-app-backend-service
spec:
  type: ClusterIP
  selector:
    app: sample-app-backend-pods
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

  - Note the `type: ClusterIP` field, which allows the backend to be accessed from within the cluster.

3. **Build and deploy:**

- Build the Docker image:

```
$ cd td6/scripts/sample-app-backend
$ npm run dockerize
```

- You can authenticate to the Kubernetes cluster in Docker Desktop as follows:

```
kubectl config use-context docker-desktop
```

- Deploy to Kubernetes:

```
kubectl apply -f sample-app-deployment.yml
kubectl apply -f sample-app-service.yml
```

4. **Verify service**

- Verify that the service is running by using `kubectl`:

```
kubectl get services
```

  - Note down the service name, this will be used to access the service internally within the Kubernetes cluster.

3.2. Creating the Frontend Service

The frontend service will make requests to the backend and display the data in a web browser.

Steps:

1. **Copy the sample app:**

- Go to the root of `devops_base` and copy the existing `sample-app` folder to `td6/scripts/sample-app-frontend`:

```
$ cd ~/devops_base
$ cp -r td5/scripts/sample-app td6/scripts/sample-app-frontend
```

- Copy the Kubernetes Deployment and Service configurations from `td3/scripts/kubernetes` to `td6/scripts/sample-app-frontend/`:

```
$ cp td3/scripts/kubernetes/sample-app-deployment.yml td6/scripts/sample-app-frontend/
$ cp td3/scripts/kubernetes/sample-app-service.yml td6/scripts/sample-app-frontend/
```

2. **Update the frontend app to make an HTTP request to the backend and to return HTML :**

   - Modify `td6/scripts/sample-app-frontend/app.js` :

```
// td6/scripts/sample-app-frontend/app.js (Example 6-13)
const backendHost = 'sample-app-backend-service';

app.get('/', async (req, res) => {
  const response = await fetch(`http://${backendHost}`);
  const responseBody = await response.json();
  res.render('hello', { name: responseBody.text });
});
```

     - **Note:** The backend URL is `sample-app-backend-service` , which is the name of the Kubernetes service you created earlier.

   - Modify `td6/scripts/sample-app-frontend/views/hello.ejs` :

```
<!-- td6/scripts/sample-app-frontend/views/hello.ejs (Example 6-14) -->
<p>Hello from <b><%= name %></b>!</p>
```

   - Update `td6/scripts/sample-app-frontend/package.json` :

```
// td6/scripts/sample-app-frontend/package.json (Example 6-15)
{
  "name": "sample-app-frontend",
  "version": "0.0.1",
  "description": "Frontend app for 'DevOps Labs!'"
}
```

   - Modify `td6/scripts/sample-app-frontend/sample-app-deployment.yml` :

```
# td6/scripts/sample-app-frontend/sample-app-deployment.yml (Example 6-16)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app-frontend-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sample-app-frontend-pods
  template:
    metadata:
      labels:
        app: sample-app-frontend-pods
    spec:
      containers:
        - name: sample-app-frontend
          image: sample-app-frontend:0.0.1
          ports:
            - containerPort: 8080
          env:
            - name: NODE_ENV
              value: production
```

- Modify `td6/scripts/sample-app-frontend/sample-app-service.yml`:

```
# td6/scripts/sample-app-frontend/sample-app-service.yml (Example 6-17)
apiVersion: v1
kind: Service
metadata:
  name: sample-app-frontend-loadbalancer
spec:
  type: LoadBalancer
  selector:
    app: sample-app-frontend-pods
```

  - This configuration uses `type: LoadBalancer` to make the frontend accessible from outside the Kubernetes cluster.

3. **Build and deploy:**

- Build the Docker image:

```
$ cd td6/scripts/sample-app-frontend
$ npm run dockerize
```

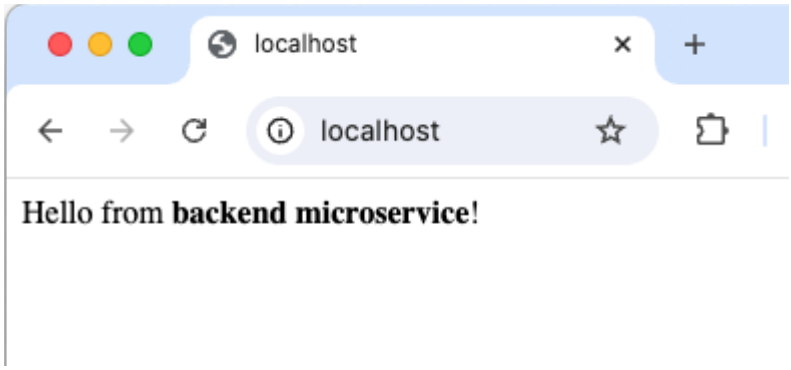
- Deploy to Kubernetes:

```
kubectl apply -f sample-app-deployment.yml
kubectl apply -f sample-app-service.yml
```

4. **Test the frontend service:**
   - Use the command below to list the service. Note the `EXTERNAL-IP` and `PORT` of the frontend service.

```
kubectl get services
```

     - Open a web browser and go to the `EXTERNAL-IP:PORT` of the frontend where it's set to `localhost` and that it's listening on port `80`, so you can test it by going to `http://localhost`, you should see `Hello from backend microservice!`.



Exercise [Practice]

Here are a few exercises you can try to go deeper:

1. **Port Configuration for Application Instances**:
   The frontend and backend components are configured to listen on port 8080 by default. While this setup is functional when the applications are executed within isolated Docker containers, it results in port conflicts if the applications are run locally without Docker (e.g., using the `npm start` command). To address this issue, modify the configuration of one of the applications to use a different port for listening.

2. **Resolution of Test Failures and Implementation of Test Doubles**:
   Following recent updates, the automated tests defined in `app.test.js` for both the frontend and backend are failing. It is necessary to debug and resolve these test failures to ensure the functionality of the system. Additionally, explore the use of **test doubles** (commonly referred to as *mocks*) to enable testing of the frontend independently of the

backend. This approach will decouple the test dependencies and streamline the testing process for the frontend.

3. **Error Handling for Backend Request Failures**:
   When the frontend attempts to communicate with the backend and the request fails, the frontend currently crashes, leading to an unsatisfactory user experience. Implement proper error handling mechanisms to catch these failures and display a meaningful and user-friendly error message instead of allowing the application to crash. This improvement will enhance the robustness and usability of the frontend application.

> [!IMPORTANT] When you're done testing, you may want to run `kubectl delete` on each of the Deployments and Services to undeploy them from your Kubernetes cluster. You should also commit your changes to Git, as you will continue to iterate on this code later.

Conclusion

In this lab, you've learned how to set up multiple isolated AWS environments using different AWS accounts, configure OpenTofu workspaces for deployments to multiple environments, and deploy interconnected microservices in Kubernetes. These skills are essential for building scalable and resilient applications in real-world scenarios. Remember to continue experimenting and practicing the techniques you've learned here to deepen your understanding and expertise.