Implement DevOps Workflows in Google Cloud: Challenge Lab

experiment Lab schedule 1 hour 30 minutes universal_currency_alt No cost

show_chart Intermediate

Rate Lab

GSP330

Overview

In a challenge lab you're given a scenario and a set of tasks. Instead of following step-by-step instructions, you will use the skills learned from the labs in the course to figure out how to complete the tasks on your own! An automated scoring system (shown on this page) will provide feedback on whether you have completed your tasks correctly.

When you take a challenge lab, you will not be taught new Google Cloud concepts. You are expected to extend your learned skills, like changing default values and reading and researching error messages to fix your own mistakes.

To score 100% you must successfully complete all tasks within the time period!

This lab is recommended for students enrolled in the Implement DevOps Workflows in Google Cloud course. Are you ready for the challenge?

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).

> **Note:** Use an Incognito or private browser window to run this lab. This prevents any conflicts between your personal account and the Student account, which may cause extra charges incurred to your personal account.

- Time to complete the lab---remember, once you start, you cannot pause a lab.

> **Note:** If you already have your own personal Google Cloud account or project, do not use it for this lab to avoid extra charges to your account.

Challenge scenario

As a recent hire as a DevOps Engineer for Cymbal Superstore a few months ago, you have learned the ins and outs of how the company operates its e-commerce website. Specifically, the DevOps team is working on a large scale CI/CD pipeline that they would like your assistance with building. This allows the company to help developers automate tasks, collaborate more effectively with other teams, and release software more frequently and reliably. Your experience with Cloud Source Repositories, Artifact Registry, Docker, and Cloud Build is going to be a large help since Cymbal Superstore would like to use all native Google Cloud Services for their pipeline.

Before you start this project, the DevOps team would like you to demonstrate your new skills. As part of this demonstration, they have a list of tasks they would like to see you do in an allotted period of time in a sandbox environment.

Your challenge

Your tasks include the following:

- Creating a GKE cluster based on a set of configurations provided.
- Creating a Google Source Repository to host your Go application code.
- Creating Cloud Build Triggers that deploy a production and development application.
- Pushing updates to the app and creating new builds.
- Rolling back the production application to a previous version.

Overall, you are creating a simple CI/CD pipeline using Cloud Source Repositories, Artifact Registry, and Cloud Build.

Task 1. Create the lab resources

In this section, you initialize your Google Cloud project for the demo environment. You enable the required APIs, configure Git in Cloud Shell, create an Artifact Registry Docker repository, and create a GKE cluster to run your production and development applications on.

1. Run the following command to enable the APIs for GKE, Cloud Build, and Cloud Source Repositories:

```
gcloud services enable container.googleapis.com \
    cloudbuild.googleapis.com \
    sourcerepo.googleapis.com
```

content_c

2. Add the Kubernetes Developer role for the Cloud Build service account:

```
export PROJECT_ID=$(gcloud config get-value project)
gcloud projects add-iam-policy-binding $PROJECT_ID \
--member=serviceAccount:$(gcloud projects describe $PROJECT_ID \
--format="value(projectNumber)")@cloudbuild.gserviceaccount.com --
role="roles/container.developer"
```

content_c

3. Run the following to configure Git in Cloud Shell, replacing `<email>` with your generated lab email address and `<name>` with your name.

```
git config --global user.email <email>
git config --global user.name <name>
```

content_c

4. Create an Artifact Registry Docker repository named **my-repository** in the `REGION` region to store your container images.

5. Create a GKE Standard cluster named `hello-cluster` with the following configuration:

Setting	Value
Zone	`ZONE`
Release channel	**Regular**
Cluster version	`1.27.8-gke.1067004` *or newer*

Cluster autoscaler	**Enabled**
Number of nodes	**3**
Minimum nodes	**2**
Maximum nodes	**6**

6. Create the `prod` and `dev` namespaces on your cluster.

Click *Check my progress* to verify the objective.

Create the lab resources

Check my progress

Task 2. Create a repository in Cloud Source Repositories

In this task, you create a repository **sample-app** in Cloud Source Repositories and initialize it with some sample code. This repository holds your Go application code, and be the primary source for triggering builds.

1. Create an empty repository named **sample-app** in Cloud Source Repositories.

2. Clone the **sample-app** Cloud Source Repository in Cloud Shell.

3. Use the following command to copy the sample code into your `sample-app` directory:

```
cd ~
```

content_c

```
gsutil cp -r gs://spls/gsp330/sample-app/* sample-app
```

4. Run the following command, which will automatically replace the `<your-region>` and `<your-zone>` placeholders in the `cloudbuild-dev.yaml` and `cloudbuild.yaml` files with the assigned region and zone of your project:

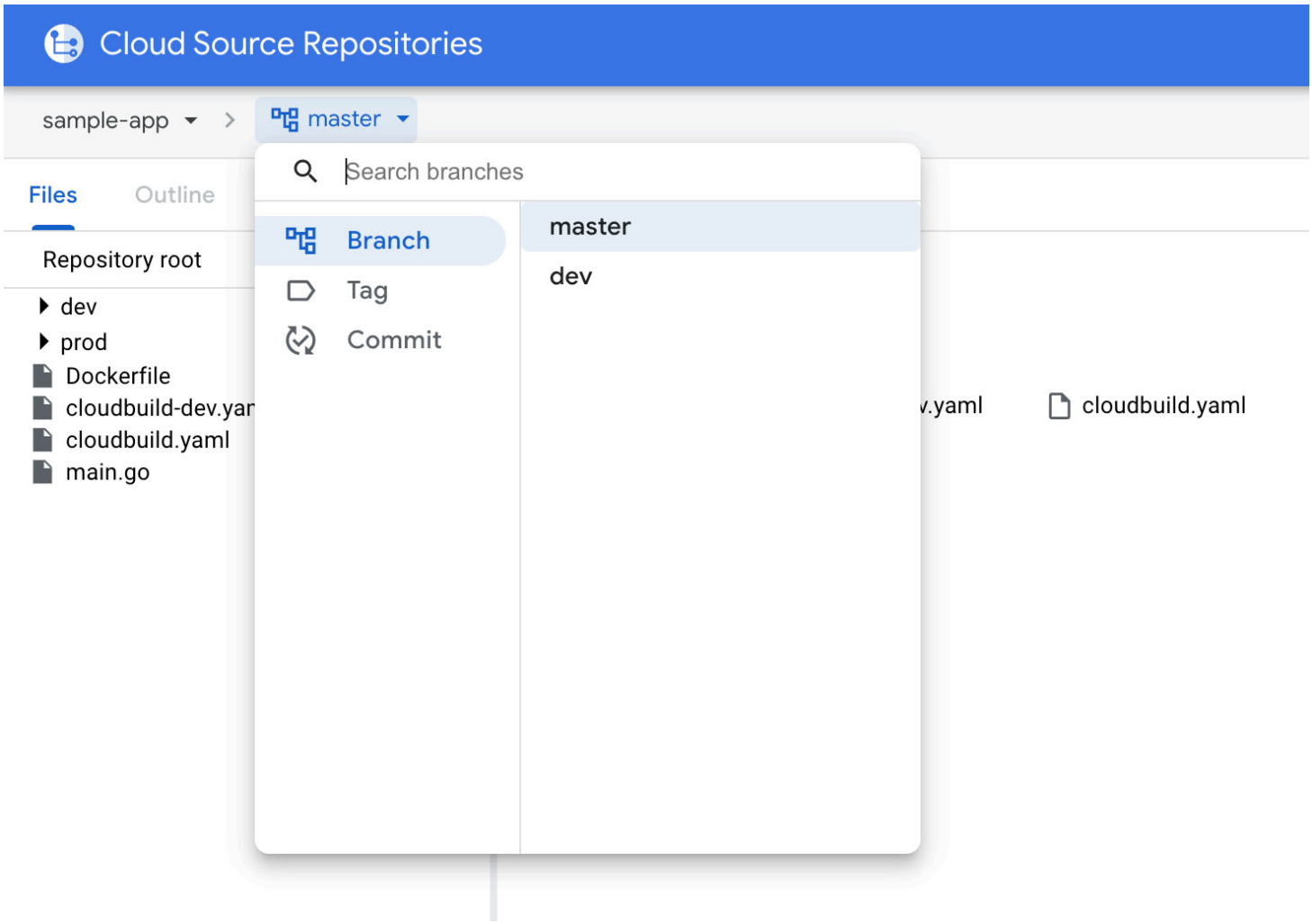
```
export REGION="REGION"
export ZONE="ZONE"
for file in sample-app/cloudbuild-dev.yaml sample-app/cloudbuild.yaml;
do
    sed -i "s/<your-region>/${REGION}/g" "$file"
    sed -i "s/<your-zone>/${ZONE}/g" "$file"
done
```

content_c

5. Make your first commit with the sample code added to your `sample-app` directory, and push the changes to the **master** branch.

6. Create a branch named **dev**. Make a commit with the sample code added to your `sample-app` directory and push the changes to the **dev** branch.

7. Verify you have the sample code and branches stored in the Source Repository.



The code you just cloned contains a simple Go application that has two entry points: Red and Blue. Each displays a simple colored square on the web page depending on the entry point you go to.

Click *Check my progress* to verify the objective.

Create the repository in Cloud Source Repositories

Check my progress

Task 3. Create the Cloud Build Triggers

In this section, you create two Cloud Build Triggers.

- The first trigger listens for changes on the `master` branch and builds a **Docker image** of your application, pushes it to Google Artifact Registry, and deploys the latest version of the image to the **prod** namespace in your GKE cluster.

- The second trigger listens for changes on the `dev` branch and build a Docker image of your application and push it to Google Artifact Registry, and deploy the latest version of the image to the **dev** namespace in your GKE cluster.

    1. Create a Cloud Build Trigger named **sample-app-prod-deploy** that with the following configurations:

        - Event: **Push to a branch**
        - Source Repository: `sample-app`
        - Branch: `^master$`
        - Cloud Build Configuration File: `cloudbuild.yaml`

    2. Create a Cloud Build Trigger named **sample-app-dev-deploy** that with the following configurations:

        - Event: **Push to a branch**
        - Source Repository: `sample-app`
        - Branch: `^dev$`
        - Cloud Build Configuration File: `cloudbuild-dev.yaml`

After setting up the triggers, any changes to the branches trigger the corresponding Cloud Build pipeline, which builds and deploy the application as specified in the `cloudbuild.yaml` files.

Click *Check my progress* to verify the objective.

Create the Cloud Build Triggers

Check my progress

Task 4. Deploy the first versions of the application

In this section, you build the first version of the production application and the development application.

Build the first development deployment

1. In Cloud Shell, inspect the `cloudbuild-dev.yaml` file located in the **sample-app** directory to see the steps in the build process. In `cloudbuild-dev.yaml` file, replace the `<version>` on lines 9 and 13 with `v1.0`.

2. Navigate to the `dev/deployment.yaml` file and Update the `<todo>` on line 17 with the correct container image name. Also, replace the **`PROJECT_ID`** variable with actual project ID in the container image name.

Note: Make sure you have same container image name in **dev/deployment.yaml** and **cloudbuild-dev.yaml** file.

3. Make a commit with your changes on the **`dev`** branch and push changes to trigger the **sample-app-dev-deploy** build job.

4. Verify your build executed successfully in **Cloud build History** page, and verify the **development-deployment** application was deployed onto the **`dev`** namespace of the cluster.

5. Expose the **development-deployment** deployment to a **LoadBalancer** service named `dev-deployment-service` on port 8080, and set the target port of the container to the one specified in the Dockerfile.

6. Navigate to the Load Balancer IP of the service and add the `/blue` entry point at the end of the URL to verify the application is up and running. It should resemble something like the following: `http://34.135.97.199:8080/blue`.

Build the first production deployment

1. Switch to the `master` branch. Inspect the `cloudbuild.yaml` file located in the **sample-app** directory to see the steps in the build process. In `cloudbuild.yaml` file, replace the `<version>` on lines **11** and **16** with `v1.0`.

2. Navigate to the `prod/deployment.yaml` file and update the `<todo>` on line 17 with the correct container image name. Also, replace the `PROJECT_ID` variable with actual project ID in the container image name.

> **Note:** Make sure you have same container image name in **prod/deployment.yaml** and **cloudbuild.yaml** file.

3. Make a commit with your changes on the `master` branch and push changes to trigger the **sample-app-prod-deploy** build job.

4. Verify your build executed successfully in **Cloud build History** page, and verify the **production-deployment** application was deployed onto the `prod` namespace of the cluster.

5. Expose the **production-deployment** deployment on the `prod` namespace to a **LoadBalancer** service named `prod-deployment-service` on port 8080, and set the target port of the container to the one specified in the Dockerfile.

6. Navigate to the Load Balancer IP of the service and add the `/blue` entry point at the end of the URL to verify the application is up and running. It should resemble something like the following: `http://34.135.245.19:8080/blue`.

Click *Check my progress* to verify the objective.

Deploy the first versions of the application

Check my progress

Task 5. Deploy the second versions of the application

In this section, you build the second version of the production application and the development application.

Build the second development deployment

1. Switch back to the **dev** branch.

> **Note:** Before proceeding, make sure you are on **dev** branch to create deployment for **dev** environment.

2. In the `main.go` file, update the `main()` function to the following:

```
func main() {
        http.HandleFunc("/blue", blueHandler)
        http.HandleFunc("/red", redHandler)
        http.ListenAndServe(":8080", nil)
}
```

content_c

3. Add the following function inside of the `main.go` file:

```
func redHandler(w http.ResponseWriter, r *http.Request) {
        img := image.NewRGBA(image.Rect(0, 0, 100, 100))
        draw.Draw(img, img.Bounds(), &image.Uniform{color.RGBA{255, 0, 0,
        w.Header().Set("Content-Type", "image/png")
        png.Encode(w, img)
}
```

content_c

4. Inspect the `cloudbuild-dev.yaml` file to see the steps in the build process. Update the version of the Docker image to `v2.0`.

5. Navigate to the `dev/deployment.yaml` file and update the container image name to the new version (`v2.0`).

6. Make a commit with your changes on the **`dev`** branch and push changes to trigger the **sample-app-dev-deploy** build job.

7. Verify your build executed successfully in **Cloud build History** page, and verify the **development-deployment** application was deployed onto the `dev` namespace of the cluster and is using the `v2.0` image.

8. Navigate to the Load Balancer IP of the service and add the `/red` entry point at the end of the URL to verify the application is up and running. It should resemble something like the following: `http://34.135.97.199:8080/red`.

Note: it may take a couple of minutes for the updates to propagate to your load balancer.

Build the second production deployment

1. Switch to the **`master`** branch.

Note: Before proceeding, make sure you are on **master** branch to create deployment for **master** environment.

2. In the `main.go` file, update the `main()` function to the following:

```
func main() {
        http.HandleFunc("/blue", blueHandler)
        http.HandleFunc("/red", redHandler)
        http.ListenAndServe(":8080", nil)
}
```

content_c

3. Add the following function inside of the `main.go` file:

```
func redHandler(w http.ResponseWriter, r *http.Request) {
        img := image.NewRGBA(image.Rect(0, 0, 100, 100))
        draw.Draw(img, img.Bounds(), &image.Uniform{color.RGBA{255, 0, 0,
        w.Header().Set("Content-Type", "image/png")
        png.Encode(w, img)
}
```

content_c

4. Inspect the `cloudbuild.yaml` file to see the steps in the build process. Update the version of the Docker image to `v2.0`.

5. Navigate to the `prod/deployment.yaml` file and update the container image name to the new version (`v2.0`).

6. Make a commit with your changes on the `master` branch and push changes to trigger the **sample-app-prod-deploy** build job.

7. Verify your build executed successfully in **Cloud build History** page, and verify the **production-deployment** application was deployed onto the **`prod`** namespace of the cluster and is using the `v2.0` image.

8. Navigate to the Load Balancer IP of the service and add the `/red` entry point at the end of the URL to verify the application is up and running. It should resemble something like the following: `http://34.135.245.19:8080/red`.

Note: it may take a couple of minutes for the updates to propagate to your load balancer.

Great! You have successfully created fully functioning production and development CI/CD pipelines.

Click *Check my progress* to verify the objective.

Deploy the second versions of the application

Check my progress

Task 6. Roll back the production deployment

In this section, you roll back the production deployment to a previous version.

1. Roll back the **production-deployment** to use the `v1.0` version of the application.

> **Hint:** Using Cloud build history, you can easily rollback/rebuild the deployments with the previous versions.

2. Navigate to the Load Balancer IP of the service and add the `/red` entry point at the end of the URL of the production deployment and response on the page should be **`404`**.

Click *Check my progress* to verify the objective.

Roll back the production deployment

Check my progress

Congratulations!