Building Microservices and a CI/CD Pipeline with AWS

Table of contents

Project overview and objectives

Return to table of contents

In this project, you're challenged to use at least 11 AWS offerings, including some that might be new to you, to build a microservices and continuous integration and continuous development (CI/CD) solution. Throughout various AWS Academy courses, you have completed hands-on labs. You have used different AWS services and features to build a variety of solutions.

In many parts of the project, step-by-step guidance is *not* provided. This is intentional. These specific sections of the project are meant to challenge you to practice skills that you have acquired throughout your learning experiences prior to this project. In some cases, you might be challenged to use resources to independently learn new skills.

By the end of this project, you should be able to do the following:

- Recognize how a Node.js web application is coded and deployed to run and connect to a relational database where the application data is stored.
- Create an AWS Cloud9 integrated development environment (IDE) and a code repository (repo) in which to store the application code.
- Split the functionality of a monolithic application into separate containerized microservices.
- Use a container registry to store and version control containerized microservice Docker images.
- Create code repositories to store microservice source code and CI/CD deployment assets.
- Create a serverless cluster to fulfill cost optimization and scalability solution requirements.
- Configure an Application Load Balancer and multiple target groups to route traffic between microservices.
- Create a code pipeline to deploy microservices containers to a blue/green cluster deployment.
- Use the code pipeline and code repository for CI/CD by iterating on the application design.

The lab environment and monitoring your budget

Return to table of contents

This environment is long lived. When the session timer runs to 0:00, the session will end, but any data and resources that you created in the AWS account will be retained. If you later launch a new session (for example, the next day), you will find that your work is still in the lab environment. Also, at any point before the session timer reaches 0:00, you can choose **Start Lab** again to extend the current session time.

❗ **Important:** Monitor your lab budget in the lab interface. When you have an active lab session, the latest known remaining budget information displays at the top of this screen. The remaining budget that you see might not reflect your most recent account activity. If you exceed your lab budget, your lab account will be disabled, and all progress and resources will be lost. If you exceed the budget, please contact your educator for assistance.

If you ever want to reset the lab environment to the original state, before you started the lab, use the **Reset** option above these instructions. Note that this won't reset your budget. ⚠ **Warning:** If you reset the environment, you will *permanently delete* everything that you have created or stored in this AWS account.

AWS service restrictions

In this lab environment, access to AWS services and service actions are restricted to the ones that are needed to complete the project. You might encounter errors if you attempt to access other services or perform actions beyond the ones that are described in this lab.

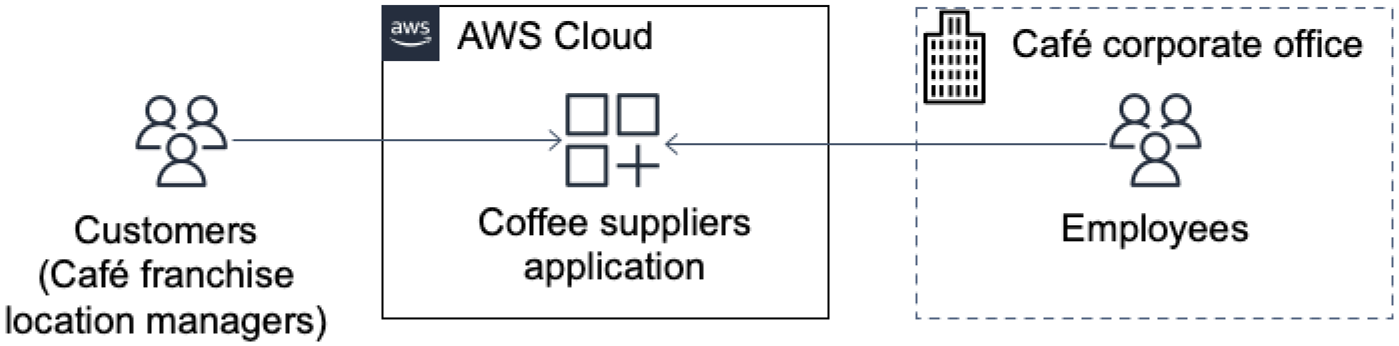
Scenario

Return to table of contents

The owners of a café corporation with many franchise locations have noticed how popular their gourmet coffee offerings have become.

Customers (the café franchise location managers) cannot seem to get enough of the high-quality coffee beans that are needed to create amazing cappuccinos and lattes in their cafés.

Meanwhile, the *employees* in the café corporate office have been challenged to consistently source the highest-quality coffee beans. Recently, the leaders at the corporate office learned that one of their favorite coffee suppliers wants to sell her company. The café corporate managers jumped at the opportunity to buy the company. The acquired coffee supplier runs a coffee supplier listings application on an AWS account, as shown in the following image.


AWS Cloud
Café corporate office
Customers (Café franchise location managers)
Coffee suppliers application
Employees

The coffee suppliers application currently runs as a *monolithic* application. It has reliability and performance issues. That is one of the reasons that you have recently been hired to work in the café corporate office. In this project, you perform tasks that are associated with software development engineer (SDE), app developer, and cloud support engineer roles.

You have been tasked to split the monolithic application into *microservices*, so that you can scale the services independently and allocate more compute resources to the services that experience the highest demand, with the goal of avoiding bottlenecks. A microservices design will also help avoid single points of failure, which could bring down the entire application in a monolithic design. With services isolated from one another, if one microservice becomes temporarily unavailable, the other microservices might remain available.

You have also been challenged to develop a CI/CD pipeline to automatically deploy updates to the production cluster that runs containers, using a blue/green deployment strategy.

Solution requirements

Return to table of contents

The solution must meet the following requirements:

- R1 - Design: The solution must have an architecture diagram.
- R2 - Cost optimized: The solution must include a cost estimate.
- R3 - Microservices-based architecture: Ensure that the solution is functional and deploys a microservice-based architecture.
- R4 - Portability: The solution must be portable so that the application code isn't tied to running on a specific host machine.
- R5 - Scalability/resilience: The solution must provide the ability to increase the amount of compute resources that are dedicated to serving requests as usage patterns change, and the solution must use routing logic that is reliable and scalable.
- R6 - Automated CI/CD: The solution must provide a CI/CD pipeline that can be automatically invoked when code is updated and pushed to a version-controlled code repository.

Lab project tips

Return to table of contents

Knowledge of Linux Bash commands would be helpful, but it isn't mandatory.

Tip: One online Linux Bash reference is the Linux Man Pages on linux.die.net. You can search or browse for commands to learn more about them.

Finally, you are encouraged to be resourceful as you complete this project. For example, reference the AWS Documentation or a search engine if you need an answer to a technical question.

As you work through the project, you will find other tips to help you complete specific phases or tasks.

Approach

Return to table of contents

The following table describes the phases of the project:

Phase	Detail	Solution requirement
1	Create an architecture diagram and cost estimate for the solution.	R1, R2
2	Analyze the design of the monolithic application and test the application.	R3
3	Create a development environment on AWS Cloud9, and check the monolithic source code into CodeCommit.	R3
4	Break the monolithic design into microservices, and launch test Docker containers.	R3, R4
5	Create ECR repositories to store Docker images. Create an ECS cluster, ECS task definitions, and CodeDeploy application specification files.	R3
6	Create target groups and an Application Load Balancer that routes web traffic to them.	R5
7	Create ECS services.	R5
8	Configure applications and deployments groups in CodeDeploy, and create two CI/CD pipelines by using CodePipeline.	R5, R6

Phase	Detail	Solution requirement
9	Modify your microservices and scale capacity, and use the pipelines that you created to deploy iterative improvements to production by using a blue/green deployment strategy.	R5, R6

Phase 1: Planning the design and estimating cost

In this phase, you will plan the design of your architecture. First, you will create an architecture diagram.

You will also estimate the cost of the proposed solution and present the estimate to your educator. An important first step for any solution is to plan the design and estimate the cost. Review the various components in the architecture to adjust the estimated cost. Cost, along with the features and limitations of specific AWS services, is an important factor when you build a solution.

Task 1.1: Create an architecture diagram

Return to table of contents

1. Create an architectural diagram to illustrate what you plan to build. Consider how you will accomplish each requirement in the solution. Read through the phases in this document to be aware of which AWS services and features you have been asked to use. Be sure to include the following services or resources in your diagram:
   - Amazon Virtual Private Cloud (Amazon VPC)
   - Amazon EC2: Instances, Application Load Balancer, target groups
   - AWS CodeCommit: Repository
   - AWS CodeDeploy
   - AWS CodePipeline: Pipeline
   - Amazon Elastic Container Service (Amazon ECS): Services, containers, tasks
   - Amazon Elastic Container Registry (Amazon ECR): Repository
   - AWS Cloud9 environment
   - AWS Identity and Access Management (IAM): Roles
   - Amazon Relational Database Service (Amazon RDS)
   - Amazon CloudWatch: Logs

References

- AWS Architecture Icons: This site provides tools to draw AWS architecture diagrams.
- AWS Reference Architecture Diagrams: This site provides a list of AWS architecture diagrams for various use cases. You might want to use these diagrams as references.

Task 1.2: Develop a cost estimate

Return to table of contents

Develop a cost estimate that shows the cost to run the solution that you created an architecture diagram for. Assume that the solution will run in the us-east-1 Region for 12 months. Use the AWS Pricing Calculator for this estimate.

If instructed by your educator, complete these additional requirements:

- Add your architectural diagram and cost estimate to presentation slides. Your educator might want to evaluate this information as part of assessing your work on this project. A presentation template is provided.
- Capture screenshots of your work at the end of each task or phase to include in the presentation or document. Your instructor might use the presentation or document to help assess how well you completed the project requirements.

Reference

- A PowerPoint presentation template is available to you in the course materials.

Phase 2: Analyzing the infrastructure of the monolithic application

In this task, you will analyze the current application infrastructure and then test the web application.

Task 2.1: Verify that the monolithic application is available

Return to table of contents

1. Verify that the monolithic web application is accessible from the internet.
   - Navigate to the Amazon EC2 console.
   - Copy the Public IPv4 address of the *MonolithicAppServer* instance, and load it in a new browser tab.

     The coffee suppliers website displays.

   Note: The page is available at `http://` instead of `https://`. Your browser might indicate that the site isn't secure because it doesn't have a valid SSL/TLS certificate. You can ignore the warning in this development environment.

Task 2.2: Test the monolithic web application

Return to table of contents

In this task, you add data to the web application, test the functionality, and observe the different URL paths that are used to display the different pages. These URL paths are important to understand for when you divide this application into microservices later.

1. Choose **List of suppliers**.

   Notice that the URL path includes `/suppliers`.
2. Add a new supplier.
   - On the page where you add a new supplier, notice that the URL path includes `/supplier-add`.
   - Fill in all of the fields to create a supplier entry.
3. Edit an entry.
   - On the page where you edit a supplier entry, notice that the URL path now includes `supplier-update/1`.
   - Modify the record in some way and save the change.

     Notice that the change was saved in the record.

Task 2.3: Analyze how the monolithic application runs

Return to table of contents

1. Use EC2 Instance Connect to connect to the *MonolithicAppServer* instance.
2. Analyze how the application is running.
   - In the terminal session, run the following command:

     ```
     sudo lsof -i :80
     ```

     What did you notice in the command output? What port and protocol is the *node* daemon using?
   - Next, run the following command:

     ```
     ps -ef | head -1; ps -ef | grep node
     ```

     What did you notice in the command output? Which user on this EC2 instance is running a *node* process? Does the node process ID (PID) match any of the PIDs from the output of the command that you ran before the last one?
3. To analyze the application structure, run the following commands:

   ```
   cd ~/resources/codebase_partner
   ls
   ```

   This is where the index.js file exists. It contains the base application logic, which you will look at in detail in a moment.

   Questions for thought: Based on what you have observed, what can you determine about how and where this node application is running? How do you think it was installed? What prerequisite libraries, if any, were required to make it run? Where does the application store data?
4. Connect a MySQL client to the RDS database that the node application stores data in.
   - Find and copy the endpoint of the RDS database that is running in the lab environment.
   - To verify that the database can be reached from the *MonolithicAppServer* instance on the standard MySQL port number, use the `nmap -Pn` command with the RDS database endpoint that you copied.
   - To connect to the database, use the MySQL client that is already installed on the *MonolithicAppServer* instance. Use the following values:
     - **Username:** `admin`
     - **Password:** `lab-password`
5. Observe the data in the database.
   - From the `mysql>` prompt, run SQL commands as appropriate to see that a database named *COFFEE* contains a table named *suppliers*.

     This table contains the supplier entry or entries that you added earlier when you tested the web application.
   - Exit the MySQL client and then close the EC2 Instance Connect tab. Also close the coffee suppliers web application tab.

References

- Connect to Your Linux Instance with EC2 Instance Connect
- Connecting from the MySQL Command-Line Client (Unencrypted)
- For information about the lsof, ps, grep, and nmap commands, see the Linux Man Pages on linux.die.net.

Phase 3: Creating a development environment and checking code into a Git repository

In this phase, you will create a development environment by using AWS Cloud9. You will also check your application code into AWS CodeCommit, which is a Git-compatible repository.

Task 3.1: Create an AWS Cloud9 IDE as your work environment

Return to table of contents

In this task, you will create your development environment. If you aren't familiar with AWS Cloud9, the following references might be helpful:

- Creating an Environment in AWS Cloud9
- Tour of the AWS Cloud9 IDE

1. Create an AWS Cloud9 instance that is named `MicroservicesIDE` and then open the IDE.

   It should run as a new EC2 instance of size *t3.small* and run Amazon Linux 2. The instance should support SSH connections and run in the *LabVPC* in *Public Subnet1*.

Task 3.2: Copy the application code to your IDE

Return to table of contents

In this task, you will copy the source code for the monolithic application to your development environment.

1. From the **AWS Details** panel on this lab instructions page, download the **labsuser.pem** file to your local computer.
2. Upload the .pem file to your AWS Cloud9 IDE, and use the Linux `chmod` command to set the proper permissions on the file so that you can use it to connect to an EC2 instance.
3. Create a temp directory on the AWS Cloud9 instance at `/home/ec2-user/environment/temp`.
4. From the Amazon EC2 console, retrieve the private IPv4 address of the *MonolithicAppServer* instance.
5. Use the Linux `scp` command in the Bash terminal on the AWS Cloud9 instance to copy the source code for the node application from the *MonolithicAppServer* instance to the temp directory that you created on the AWS Cloud9 instance.

   The following snippet provides an example scp command:

   ```
   scp -r -i ~/environment/labsuser.pem ubuntu@$appServerPrivIp:/home/ubuntu/resources/codebase_partner/*
   ~/environment/temp/
   ```

6. In the file browser of the IDE, verify that the source files for the application have been copied to the temp directory on the AWS Cloud9 instance.

Task 3.3: Create working directories with starter code for the two microservices

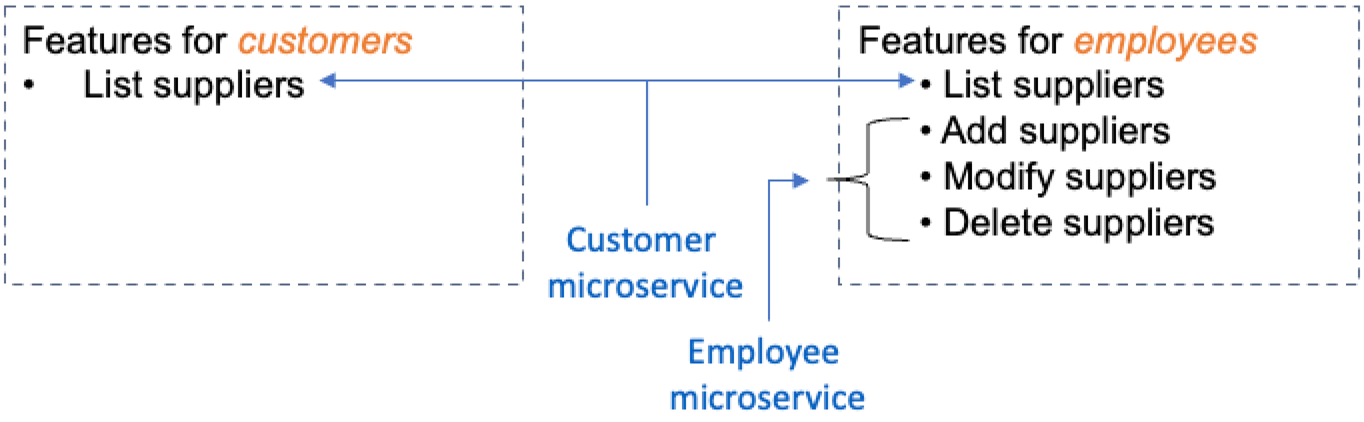
Return to table of contents

In this task, you will create areas in your development environment to support separating the application logic into two different microservices.

Based on the solution requirements of this project, it makes sense to split the monolithic application into two microservices. You will name the microservices *customer* and *employee*.

The following table explains the functionality that is needed for each microservice.

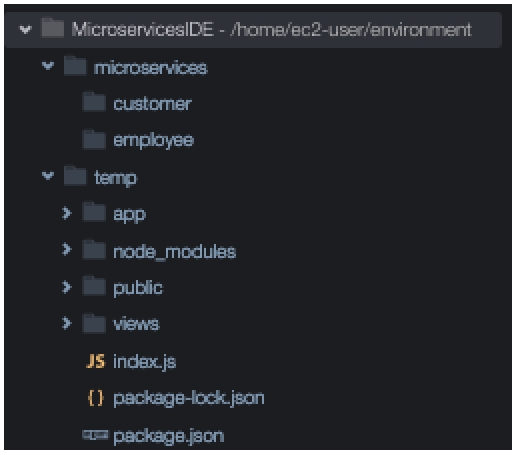
Primary User	**Microservice Functionality**	**Access Level**
Customer	The *customer* microservice will provide the functionality that customers (the café franchise location managers who want to buy coffee beans) need. The customers need a read-only view of the contact information for the suppliers to be able to buy coffee beans from them. You can think of the café franchise location managers as *customers* of the application.	Read-only
Employee	The *employee* microservice will provide the functionality that employees (the café corporate office employees) need. Employees need to add, modify, and delete suppliers who are listed in the application. Employees are responsible for keeping the listings accurate and up to date.	Read-write



The employee microservice will eventually be made available only to employees. You will accomplish this by first encapsulating them as a separate microservice (in phases 3 and 4 of the project), and then later in phase 9 of the project, you will limit who can access the employee microservice.

1. In the **microservices** directory, create two new directories that are named `customer` and `employee`.

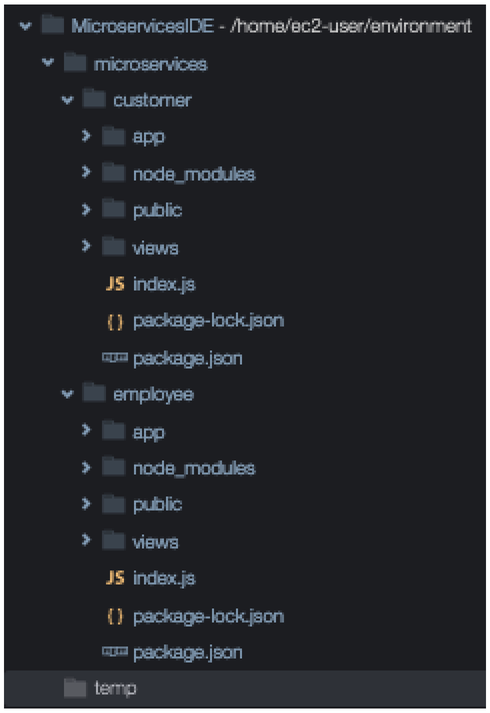
   Verify that your directory structure matches the following image:

   ```
   MicroservicesIDE - /home/ec2-user/environment
     microservices
       customer
       employee
     temp
       app
       node_modules
       public
       views
       index.js
       package-lock.json
       package.json
   ```

2. Place a copy of the source code for the monolithic application in each new directory, and remove the files from the **temp** directory.

Verify that your directory structure matches the following image:



```
MicroservicesIDE - /home/ec2-user/environment
  microservices
    customer
      app
      node_modules
      public
      views
      index.js
      package-lock.json
      package.json
    employee
      app
      node_modules
      public
      views
      index.js
      package-lock.json
      package.json
  temp
```

3. Delete the empty **temp** directory.

Task 3.4: Create a Git repository for the microservices code and push the code to CodeCommit

Return to table of contents

You now have directories named *customer* and *employee*, and each will contain a microservice. The two microservices will replicate the logic in your monolithic application. However, you will also be able to evolve the application functionality and to time the deployment of feature enhancements for these microservices separately.

You will benefit from checking this source code into a Git repository. In this project, you will use CodeCommit as your Git repository (repo).

1. Create a CodeCommit repository that is named `microservices`.
2. To check the unmodified application code into the microservices CodeCommit repository, run the following commands:

```
cd ~/environment/microservices
git init
git branch -m dev
git add .
git commit -m 'two unmodified copies of the application code'
git remote add origin https://git-codecommit.us-east-1.amazonaws.com/v1/repos/microservices
git push -u origin dev
```

Tip: For information about Git commands, see the Git documentation.

Analysis: By running these commands, you first *initialized* the microservices directory to be a Git repository. Then, you created a *branch* in the repository named *dev*. You *added* all files from the microservices directory to the Git repository and *committed* them. Then, you defined the microservices repository that you created in CodeCommit as the *remote origin* of this Git repository area on your IDE. Finally, you *pushed* the changes that were committed in the *dev* branch to the remote origin.

3. Configure your Git client to know your username and email address.

   Tip: For information about the commands that you need to use, see Getting Started - First-Time Git Setup.

   Note: You don't need to use your real name or email address; however, completing this step is an important part of configuring a Git client.

4. In a new browser tab, browse to the CodeCommit console and observe that the code is now checked into your microservices repository.

Phase 4: Configuring the application as two microservices and testing them in Docker containers

In this phase, you will modify the two copies of the monolithic application starter code so that the application functionality is implemented as two separate microservices. Then, for initial testing purposes, you will run the containers on the same EC2 instance that hosts the AWS Cloud9 IDE that you are using. You will use this IDE to build the Docker images and launch the Docker containers.

Task 4.1: Adjust the AWS Cloud9 instance security group settings

Return to table of contents

In this phase of the project, you will use the AWS Cloud9 instance as your test environment. You will run Docker containers on the instance to test the microservices that you will create. To be able to access the containers from a browser over the internet, you must open the ports that you will run the containers on.

1. Adjust the security group of the AWS Cloud9 EC2 instance to allow inbound network traffic on TCP ports 8080 and 8081.

Task 4.2: Modify the source code of the *customer* microservice

Return to table of contents

In this task, you will edit the source code for the *customer* microservice. Recall that the source code you are starting with is an exact copy of the monolithic application. Therefore, it still has features in it that will be handled by the employee microservice and that you don't want as part of the customer microservice. Specifically, customers shouldn't have the ability to add, edit, or delete suppliers; therefore, the changes that you make will remove that functionality from the source code. Ideally, the source code should contain only code that is needed.

1. In the AWS Cloud9 file panel, *collapse* the **employee** directory, if it is expanded, and then *expand* the **customer** directory.
2. Edit the customer/app/controller/**supplier.controller.js** file so that the remaining functions provide only the read-only actions that you want customers to be able to perform.

   Tip: After you edit the file, it should contain only the following lines:

```
const Supplier = require("../models/supplier.model.js");
const {body, validationResult} = require("express-validator");

exports.findAll = (req, res) => {
    Supplier.getAll((err, data) => {
        if (err)
            res.render("500", {message: "The was a problem retrieving the list of suppliers"});
        else res.render("supplier-list-all", {suppliers: data});
    });
};

exports.findOne = (req, res) => {
    Supplier.findById(req.params.id, (err, data) => {
        if (err) {
            if (err.kind === "not_found") {
                res.status(404).send({
                    message: `Not found Supplier with id ${req.params.id}.`
                });
            } else {
                res.render("500", {message: `Error retrieving Supplier with id ${req.params.id}`});
            }
        } else res.render("supplier-update", {supplier: data});
    });
};
```

3. Edit the customer/app/models/**supplier.model.js** file. Delete the unnecessary functions in it so that what remains are only read-only functions.

   Important: *KEEP* the last line of the file: `module.exports = Supplier;`

   Note: The model should still contain two functions: Supplier.getAll and Supplier.findById.

4. Later in the project, when you deploy the microservices behind an Application Load Balancer, you will want *employees* to be able to navigate from the main *customer* page to the area of the web application where they can add, edit, or delete supplier entries. To support this, edit the customer/views/**nav.html** file:
   - On line 3, change `Monolithic Coffee suppliers` to `Coffee suppliers`
   - On line 7, change `Home` to `Customer home`
   - Add a new line *after* line 8 that contains the following HTML:

     Important: *DON'T* delete or overwrite any of the existing lines in the file.

     ```
     <a class="nav-link" href="/admin/suppliers">Administrator link</a>
     ```

     Analysis: Adding this link will provide a navigation path to those pages that will be hosted under the `/admin/` URL path.

5. You don't want customers to see the **Add a new supplier** button or any **edit** buttons next to supplier rows. To implement these changes, edit the customer/views/**supplier-list-all.html** file:
   - Remove line 32, which contains `Add a new supplier`.
   - Remove lines 26 and 27, which contain `badge badge-info` and `supplier-update`.

6. Because the customer microservice doesn't need to support read-write actions, *DELETE* the following .html files from the customer/**views** directory:
   - supplier-add.html
   - supplier-form-fields.html
   - supplier-update.html

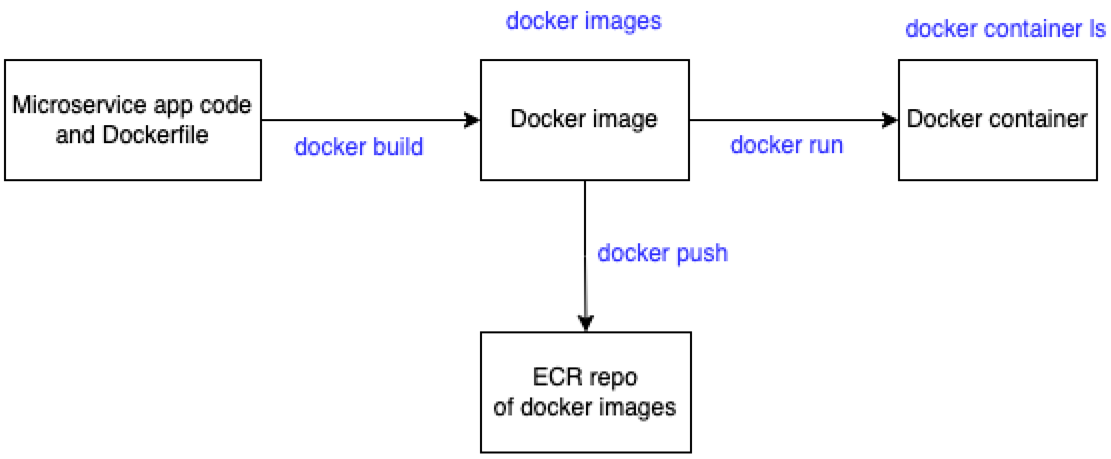
7. Edit the customer/**index.js** file as needed to account for the fact that the node application will now run on Docker containers:
   - Comment out lines 27 to 37 (ensure that each line starts with `//`).
   - On line 45, change the port number to `8080`

Tip: Recall that when this application ran on the *MonolithicAppServer* instance, it ran on port 80. However, when it runs as a Docker container, you will want the container to run on port 8080.

Task 4.3: Create the *customer* microservice Dockerfile and launch a test container

Return to table of contents

The following diagram provides an overview of how you will use Docker. Assets that you will use or create are represented by rectangles. Commands that you will use are shown in blue between and above the rectangles.



With the application code base and a Dockerfile, which you will create, you will build a *Docker image*. A Docker image is a template with instructions to create and run a Docker *container*. You can think of a Docker *image* as roughly equivalent to an Amazon Machine Image (AMI) from which you can launch an EC2 instance. A Docker *container* is roughly equivalent to an EC2 instance. However, Docker images and containers are much smaller.

1. In the **customer** directory, create a new file named `Dockerfile` that contains the following code:

   ```
   FROM node:11-alpine
   RUN mkdir -p /usr/src/app
   WORKDIR /usr/src/app
   COPY . .
   RUN npm install
   EXPOSE 8080
   CMD ["npm", "run", "start"]
   ```

   Analysis: This Dockerfile code specifies that an Alpine Linux distribution with Node.js runtime requirements should be used to create a Docker image. The code also specifies that the container should allow network traffic on TCP port 8080 and that the application should be run and started when a container that was created from the image is launched.

2. Build an image from the customer Dockerfile.
   - In the AWS Cloud9 terminal, change to the **customer** directory.
   - Run the following command:

     ```
     docker build --tag customer .
     ```

     Note: In the output, ignore the npm warning about no repository field.

Notice that the build downloaded the node Alpine starter image and then completed the other instructions as specified in the Dockerfile.

3. Verify that the customer-labeled Docker image was created.
   - Run a Docker command to list the Docker images that your Docker client is aware of.

     Tip: To find the command that you need to run, see Use the Docker Command Line in the Docker documentation.

     Tip: The output of the command should be similar to the following:

     ```
     REPOSITORY     TAG          IMAGE ID       CREATED          SIZE
     customer       latest       cdc593c9bf51   59 seconds ago   82.7MB
     node           11-alpine    f18da2f58c3d   3 years ago      75.5MB
     ```

     Note: The *node* image is the Alpine Linux image that you identified in the Dockerfile contents to download and use as the starter image. Your Docker client downloaded it from docker.io. The *customer* image is the one that you created.

4. Launch a Docker *container* that runs the *customer* microservice on port 8080. As part of the command, pass an environment variable to tell the node application the correct location of the database.
   - To set a dbEndpoint variable in your terminal session, run the following commands:

     ```
     dbEndpoint=$(cat ~/environment/microservices/customer/app/config/config.js | grep 'APP_DB_HOST' | cut -
     d '"' -f2)
     echo $dbEndpoint
     ```

     Note: You could manually find the database endpoint in the Amazon RDS console and set it as an environment variable by running `dbEndpoint="<actual-db-endpoint>"` instead of using the cat command.

     Important: If you close your AWS Cloud9 terminal or stop and restart the project lab environment, and then need to run a command that uses the $dbEndpoint variable, you might need to create the variable again. To test whether the variable is set, run `echo $dbEndpoint`
   - The following code provides an example of the command you should run that launches a container from the image:

     ```
     docker run -d --name customer_1 -p 8080:8080 -e APP_DB_HOST="$dbEndpoint" customer
     ```

     Analysis: This command launched a container named *customer_1* by using the *customer* image that you created as the template. The `-d` parameter in the command specified that it should run in the background. After the `-p` parameter, the *first number* specified the port on the AWS Cloud9 instance to publish the container to. The *second number* indicated the port that the container is running on in the *docker namespace* (also port 8080). The `-e` parameter passes the database host location as an environment variable to Docker, which gives the node application the information that it needs to establish network connectivity to the database.

5. Check which Docker containers are currently running on the AWS Cloud9 instance.

   Tip: To find the command that you need to run, see Use the Docker Command Line in the Docker documentation.

6. Verify that the *customer* microservice is running in the container and working as intended.
   - Load the following page in a new browser tab. Replace the IP address placeholder with the public IPv4 address of the AWS Cloud9 instance that you are using: `http://<cloud-9-public-IPv4-address>:8080`

     The web application microservice should load in the browser.

     Important: If you stop the lab environment and start it again, the public IPv4 address of the AWS Cloud9 instance will change.
   - Choose **List of suppliers**.

- Confirm that the supplier entries that you added earlier are displayed.

  Note: Although you changed the location where the application runs, it still connects to the same RDS database where the supplier records are stored.

  Tip: The **Administrator** link doesn't work because you haven't created the employee microservice yet.

- Confirm that the suppliers page doesn't have **Add a new supplier** and **edit** buttons.

  Troubleshooting tip: If any functionality is missing, follow these steps: (1) Stop and delete the running container, (2) modify the microservice source code as appropriate, (3) create an updated Docker image from the source code, (4) launch a new test container, and (5) verify whether the functionality is now available. For a list of commands to run to accomplish these steps, see Updating a test container running on Cloud9 in the appendix of this file.

7. Commit and push your source code changes into CodeCommit.

   Tip: You can use the Git source control panel in the AWS Cloud9 IDE, or you can use the `git commit` and `git push` commands in the terminal.

   Note: If your educator has asked you to collect information about your solution, be sure to record the commands that you run in this step and the output that was returned.

8. Optional: Observe the commit details in the CodeCommit console.

   In the **Commits** area of the repository, choose the ID for the most recent commit. Scroll down to see information about what changed in the files since the previous commit.

   Notice that deleted lines are shown in red, and added lines are shown in green so that you are able to see every detail of every change to every file that was modified.

Task 4.4: Modify the source code of the *employee* microservice

Return to table of contents

In this task, you will modify the source code for the *employee* microservice similarly to how you modified the code for the customer microservice. Customers (café franchise location managers) should have read-only access to the application data, but employees of the café corporate office should be able to add new entries or modify existing entries in the list of coffee suppliers.

As you will see later in this project, you will deploy the microservices behind an Application Load Balancer and route traffic to the microservices based on the path that is contained in the URL of the request. In this way, if the URL path includes `/admin/`, the load balancer will route the traffic to the *employee* microservice. Otherwise, if the URL path doesn't include `/admin/`, then the load balancer will route the traffic to the *customer* microservice.

Because of the need to route traffic, much of the work in this task is to configure the employee microservice to add `/admin/` to the path of the pages that it serves.

1. In the AWS Cloud9 IDE, return to the file view (toggletree view).
2. *Collapse* the **customer** directory, and then *expand* the **employee** directory.
3. In the employee/app/controller/**supplier.controller.js** file, for all the redirect calls, prepend `/admin` to the path.

   Tip: To find the three lines that need to be updated, run the following commands in the terminal:

   ```
   cd ~/environment/microservices/employee
   grep -n 'redirect' app/controller/supplier.controller.js
   ```

4. In the employee/**index.js** file, update the *app.get* calls, *app.post* calls, and a port number.
   - For all `app.get` and `app.post` calls, prepend `/admin` to the first parameter.

Tip: To find the seven lines that need to be updated, run the following command in the terminal:

```
grep -n 'app.get\|app.post' index.js
```

Important: After you edit line 22, the path should be `/admin`, *not* `/admin/`

- On line 45, change the port number to `8081`

  Note: When you run both the customer and employee microservice containers on the AWS Cloud9 instance as a test, they will need to use different port numbers so that they won't conflict with each other.

5. In the employee/views/**supplier-add.html** and employee/views/**supplier-update.html** files, for the form action paths, prepend `/admin` to the path.

   Tip: To find the three lines that need to be updated in the two files, run the following command in the terminal:

```
grep -n 'action' views/*
```

6. In the employee/views/**supplier-list-all.html** and employee/views/**home.html** files, for the HTML paths, prepend `/admin` to the path.

   Tip: To find the three lines that need to be updated in the two files, run the following command in the terminal:

```
grep -n 'href' views/supplier-list-all.html views/home.html
```

7. In the employee/views/**header.html** file, modify the title to be `Manage coffee suppliers`
8. Edit the employee/views/**nav.html** file.
   - On line 3, change `Monolithic Coffee suppliers` to `Manage coffee suppliers`
   - On line 7, replace the existing line of code with the following:

```
<a class="nav-link" href="/admin/suppliers">Administrator home</a>
```

   Note: Both the href value and the name of the link are modified in the new line.

   - Add a new line *after* line 8 that contains the following HTML:

   Important: *DON'T* delete or overwrite any of the existing lines in the file.

```
<a class="nav-link" href="/">Customer home</a>
```

   Analysis: Later in the project, when you deploy the microservices behind an Application Load Balancer, you will want employees to be able to navigate from the admin pages back to the customer area of the web application. Adding this link provides a navigation path for employees to the pages that will be hosted by the *customer* microservice under the `/` URL path.

   - Save the changes.

Task 4.5: Create the *employee* microservice Dockerfile and launch a test container

Return to table of contents

In this task, you will create a Docker image for the *employee* microservice. Then, you will launch a test container from the image and verify that the microservice works as expected.

1. Create a Dockerfile for the *employee* microservice.
   - Duplicate the Dockerfile from the customer microservice into the employee microservice area.
   - Edit the employee/**Dockerfile** to change the port number on the `EXPOSE` line to `8081`
2. Build the Docker image for the *employee* microservice. Specify `employee` as the tag.
3. Run a container named `employee_1` based on the employee image. Run it on port 8081 and be sure to pass in the database endpoint.
4. Verify that the employee microservice is running in the container and that the microservice functions as intended.
   - Load the microservice web page in a new browser tab at `http://<cloud9-public-ip-address>:8081/admin/suppliers`

     The application should load.
   - Verify that this view shows buttons to edit existing suppliers and to add a new supplier.

     Note: Links that should take you to the *customer* microservice will not work. For example, if you choose **Customer home** or **Suppliers list**, the pages won't be found because the link assumes that the *customer* microservice also runs on port 8081 (but it doesn't). You can ignore this issue—these links should work as intended when you deploy the microservices to Amazon ECS later.
   - Test adding a new supplier.

     Verify that the new supplier appears on the suppliers page.
   - Test editing an existing supplier.

     Verify that the edited supplier information appears on the suppliers page.
   - Test deleting an existing supplier:
     - On the row for a particular supplier entry, choose **edit**.
     - At the bottom of the page, choose **Delete this supplier**, and then choose **Delete this supplier** again.

       Verify that the supplier no longer appears on the suppliers page.

       Troubleshooting tip: If any of the functionality mentioned above isn't working as intended, follow these steps: (1) Stop and delete the running container, (2) modify the microservice source code as appropriate, (3) create an updated Docker image from the source code, (4) launch a new test container, and (5) verify whether the functionality is now available. For a list of commands to run to accomplish these steps, see Updating a test container running on Cloud9 in the appendix of this file.
5. To observe details about both running test containers, run the following command:

```
docker ps
```

Task 4.6: Adjust the *employee* microservice port and rebuild the image

Return to table of contents

When you tested the employee microservice on the AWS Cloud9 instance, you ran it on port 8081. However, when you deploy it to Amazon ECS, you will want it to run on port 8080. To adjust this, you need to modify two files.

1. Edit the employee/**index.js** and employee/**Dockerfile** files to change the port from `8081` to `8080`
2. Rebuild the Docker image for the employee microservice.

- To stop and delete the existing container (assumes that the container name is *employee_*), run the following command:

  ```
  docker rm -f employee_1
  ```

- Ensure that your terminal is in the **employee** directory.
- Use the `docker build` command to build a new image from the latest source files that you edited. Use the employee tag.

  Tip: If you build an image with the name of an existing image, the existing image will be overwritten.

  Note: You don't need to run a new test container, so you don't need to run `docker run`.

Task 4.7: Check code into CodeCommit

Return to table of contents

In this task, you will commit and push the changes that you made to the employee microservice to CodeCommit.

1. Review the updates that you made to the source code. To accomplish this:
   - Choose the source control icon in the AWS Cloud9 IDE.

     Notice the changes list, which indicates which files were changed since you last checked files in to the remote Git repository (CodeCommit).

   - Choose one of the files that was modified, such as index.js, to compare the version from the last Git commit to the latest version. Changes are highlighted.

     This demonstrates a benefit of using a source control system and a Git-compatible IDE such as AWS Cloud9. You can review your code changes prior to committing.

2. Check your changes into CodeCommit.

   Tip: You performed this same type of action in task 4.3. You can accomplish this step by using the Git source control panel, or you can use the `git commit` and `git push` commands in the terminal.

Phase 5: Creating ECR repositories, an ECS cluster, task definitions, and AppSpec files

At this point, you have successfully implemented numerous solution requirements. You split the monolithic application into two microservices that can run as Docker containers. You have also verified that the containers support the needed application actions, such as adding, editing, and deleting entries from the database. The microservices architecture still uses Amazon RDS to store the coffee supplier entries.

However, your work isn't finished. There are more solution requirements to implement. The containers are able to run on the AWS Cloud9 instance, but that isn't a scalable deployment architecture. You need the ability to scale the number of containers that run on each microservice up and down depending on need. Also, you need to have a load balancer to route traffic to the appropriate microservice. Finally, you need to be able to easily update each application microservice's codebase independently and roll those changes into production. In the remaining phases of the project, you will work to accomplish these solution requirements.

Task 5.1: Create ECR repositories and upload the Docker images

Return to table of contents

In this phase, you will upload the latest Docker images of the two microservices to separate Amazon ECR repositories.

1. To authorize your Docker client to connect to the Amazon ECR service, run the following commands:

```
account_id=$(aws sts get-caller-identity |grep Account|cut -d '"' -f4)
echo $account_id
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin
$account_id.dkr.ecr.us-east-1.amazonaws.com
```

A message in the command output indicates that the login succeeded.

2. Create a separate private ECR repository for each microservice.
   - Name the first repository `customer`
   - Name the second repository `employee`
3. Set permissions on the *customer* ECR repository.
   - For information about editing the existing JSON policy, see Setting a Private Repository Statement in the Amazon ECR User Guide.
   - Replace the existing lines in the policy with the following:

```
{
  "Version": "2008-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "ecr:*"
    }
   ]
}
```

4. Use the same approach to set the same permissions on the *employee* ECR repository.
5. Tag the Docker images with your unique *registryId* (account ID) value to make it easier to manage and keep track of these images.
   - In the AWS Cloud9 IDE, run the following commands:

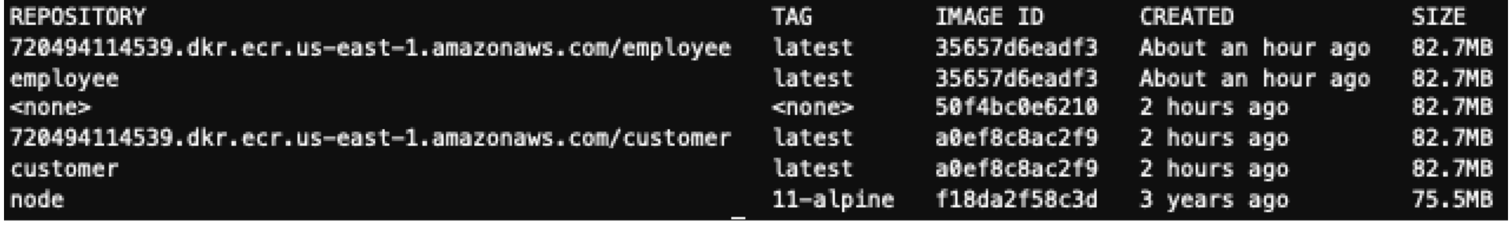
```
account_id=$(aws sts get-caller-identity |grep Account|cut -d '"' -f4)

# Verify that the account_id value is assigned to the $account_id variable
echo $account_id

# Tag the customer image
docker tag customer:latest $account_id.dkr.ecr.us-east-1.amazonaws.com/customer:latest

# Tag the employee image
docker tag employee:latest $account_id.dkr.ecr.us-east-1.amazonaws.com/employee:latest
```

Note: The commands don't return output.

   - Run the appropriate `docker` command to verify that the images exist and the tags were applied.

Tip: To find the command that you need to run, see Use the Docker Command Line in the Docker documentation.

Tip: The output of the command should be similar to the following image. Notice that the *latest* tag was applied and that the image names now include the remote repository name where you intend to store it:



```
REPOSITORY                                              TAG         IMAGE ID       CREATED              SIZE
720494114539.dkr.ecr.us-east-1.amazonaws.com/employee   latest      35657d6eadf3   About an hour ago    82.7MB
employee                                                latest      35657d6eadf3   About an hour ago    82.7MB
<none>                                                  <none>      50f4bc0e6210   2 hours ago          82.7MB
720494114539.dkr.ecr.us-east-1.amazonaws.com/customer   latest      a0ef8c8ac2f9   2 hours ago          82.7MB
customer                                                latest      a0ef8c8ac2f9   2 hours ago          82.7MB
node                                                    11-alpine   f18da2f58c3d   3 years ago          75.5MB
```

6. Run the appropriate `docker` command to push each of the Docker images to Amazon ECR.

   Tip: To find the command that you need to run, see [Use the Docker Command Line](#) in the Docker documentation.

   Tip: Before running the Docker commands, run the following command to set `account_id` as a variable in the terminal. Then, when you run the Docker commands, you can reference the account ID as `$account_id`.

   ```
   account_id=$(aws sts get-caller-identity |grep Account|cut -d '"' -f4)
   ```

   Additional tip: The commands that you run should look like the following commands but with **REPLACE_ME** replaced with the correct command:

   ```
   docker REPLACE_ME $account_id.dkr.ecr.us-east-1.amazonaws.com/customer:latest
   docker REPLACE_ME $account_id.dkr.ecr.us-east-1.amazonaws.com/employee:latest
   ```

   The output for *each* Docker command that you run to push each image to Amazon ECR should look similar to the following:

   ```
   The push refers to repository [642015801240.dkr.ecr.us-east-1.amazonaws.com/node-app]
   006e0ec54dba: Pushed
   59762f95cb06: Pushed
   22736f780b31: Pushed
   d81d715330b7: Pushed
   1dc7f3bb09a4: Pushed
   dcaceb729824: Pushed
   f1b5933fe4b5: Pushed
   latest: digest: sha256:f75b60adddb8d6343b9dff690533a1cd1fbb34ccce6f861e84c857ba7a27b77d size: 1783
   ```

7. Confirm that the two images are now stored in Amazon ECR and that each has the *latest* label applied.

Task 5.2: Create an ECS cluster

[Return to table of contents](#)

In this task, you will create an Amazon ECS cluster.

1. Create a serverless AWS Fargate cluster that is named `microservices-serverlesscluster`

   Ensure that it's configured to use *LabVPC, PublicSubnet1*, and *PublicSubnet2* (remove any other subnets). *DON'T* select Amazon EC2 instances or ECS Anywhere.

   Important: After choosing the button to create the cluster, in the banner that appears across the top of the page, choose **View in CloudFormation**. Wait until the stack that creates the cluster attains the status *CREATE_COMPLETE* before you proceed to the next task. *If the stack fails to create for any reason and therefore rolls back*, repeat these steps to try again. It should succeed the second time.

Task 5.3: Create a CodeCommit repository to store deployment files

Return to table of contents

In this task, you will create another CodeCommit repository. This repository will store the task configuration specification files that Amazon ECS will use for each microservice. The repository will also store AppSpec specification files that CodeDeploy will use for each microservice.

1. Create a new CodeCommit repository that is named `deployment` to store deployment configuration files.
2. In AWS Cloud9, in the **environment** directory, create a new directory that is named `deployment`. Initialize the directory as a Git repository with a branch named `dev`.

Task 5.4: Create task definition files for each microservice and register them with Amazon ECS

Return to table of contents

In this task, you will create a task definition file for each microservice and then register the task definitions with Amazon ECS.

1. In the new **deployment** directory, create an empty file named `taskdef-customer.json`
2. Edit the **taskdef-customer.json** file.
   - Paste the following JSON code into the file:

```
{
    "containerDefinitions": [
        {
            "name": "customer",
            "image": "customer",
            "environment": [
                {
                    "name": "APP_DB_HOST",
                    "value": "<RDS-ENDPOINT>"
                }
            ],
            "essential": true,
            "portMappings": [
                {
                    "hostPort": 8080,
                    "protocol": "tcp",
                    "containerPort": 8080
                }
            ],
            "logConfiguration": {
                "logDriver": "awslogs",
                "options": {
                    "awslogs-create-group": "true",
                    "awslogs-group": "awslogs-capstone",
                    "awslogs-region": "us-east-1",
                    "awslogs-stream-prefix": "awslogs-capstone"
                }
            }
        }
    ],
    "requiresCompatibilities": [
        "FARGATE"
    ],
    "networkMode": "awsvpc",
    "cpu": "512",
    "memory": "1024",
    "executionRoleArn": "arn:aws:iam::<ACCOUNT-ID>:role/PipelineRole",
    "family": "customer-microservice"
}
```

- Replace a couple values in the file:
  - On line 37, replace `<ACCOUNT-ID>` with the actual account ID.
  - On line 9, replace `<RDS-ENDPOINT>` with the actual RDS endpoint.
- Save the changes.

3. To register the *customer* microservice task definition in Amazon ECS, run the following command:

```
aws ecs register-task-definition --cli-input-json "file:///home/ec2-user/environment/deployment/taskdef-customer.json"
```

4. In the Amazon ECS console, verify that the *customer-microservice* task definition now appears in the **Task definitions** pane. Also, notice that the revision number displays after the task definition name.

   Tip: Consult the ECS documentation if it is helpful.

5. In the *deployment* directory, create a `taskdef-employee.json` specification file.
   - Add the same JSON code to it that currently exists in the *taskdef-customer.json* file (where you have already set the account ID and RDS endpoints).
   - After you paste in the code, change the three occurrences of `customer` to `employee`
6. To register the *employee* task definition with Amazon ECS, run an AWS CLI command.
7. In the Amazon ECS console, verify that the *employee-microservice* task definition now appears in the **Task definitions** pane. Also, notice that the revision number displays after the task definition name.

Task 5.5: Create AppSpec files for CodeDeploy for each microservice

Return to table of contents

In this task, you will continue to complete tasks to support deploying the microservices-based web application to run on an ECS cluster where the deployment is supported by a CI/CD pipeline. In this specific task, you will create two application specification (AppSpec) files, one for each microservice. These files will provide instructions for CodeDeploy to deploy the microservices to the Amazon ECS on Fargate infrastructure.

1. Create an AppSpec file for the *customer* microservice.
   - In the *deployment* directory, create a new file named `appspec-customer.yaml`
   - Paste the following YAML code into the file:

     Important: *DON'T* modify `<TASK_DEFINITION>`. This setting will be updated automatically when the pipeline runs.

```
version: 0.0
Resources:
  - TargetService:
      Type: AWS::ECS::Service
      Properties:
        TaskDefinition: <TASK_DEFINITION>
        LoadBalancerInfo:
          ContainerName: "customer"
          ContainerPort: 8080
```

Note: This file is in YAML format. In YAML, indentation is important. Verify that the code in your file maintains the indentation levels as shown in the previous code block.

- Save the changes.

2. In the same directory, create an AppSpec file for the *employee* microservice.
   - Name the file `appspec-employee.yaml`.
   - The contents of the file should be the same as the *appspec-customer.yaml* file. However, change `customer`` on the`containerName`line to be`employee`

Task 5.6: Update files and check them into CodeCommit

Return to table of contents

In this task, you will update the two task definition files. Then, you will push the four files that you created in the last two tasks into the *deployment* repository.

1. Edit the **taskdef-customer.json** file.
   - Modify line 5 to match the following line:

     ```
     "image": "<IMAGE1_NAME>",
     ```

   - Save the change.

     Analysis: `<IMAGE1_NAME>` is not a valid image name, which is why you originally set the image name to *customer* before running the AWS CLI command to register the first revision of the file with Amazon ECS. However, at this point in the project, it's important to set the image value to a placeholder text value. Later in this project, when you configure a pipeline, you will identify `IMAGE1_NAME` as placeholder text that can be dynamically updated. In summary, CodePipeline will set the correct image name dynamically at runtime.
2. Edit the **taskdef-employee.json** file.
   - Modify line 5 to match the following line:

     ```
     "image": "<IMAGE1_NAME>",
     ```

   - Save the change.
3. Push all four files to CodeCommit.

Note: Pushing the latest files to CodeCommit is essential. Later, when you create the CI/CD pipeline, the pipeline will pull these files from CodeCommit and use the details in them as instructions to deploy updates for your microservices to the Amazon ECS cluster.

Phase 6: Creating target groups and an Application Load Balancer

In this phase, you will create an Application Load Balancer, which provides an endpoint URL. This URL will act as the HTTPS entry point for customers and employees to access your application through a web browser. The load balancer will have listeners, which will have routing and access rules that determine which target group of running containers the user request should be directed to.

Task 6.1: Create four target groups

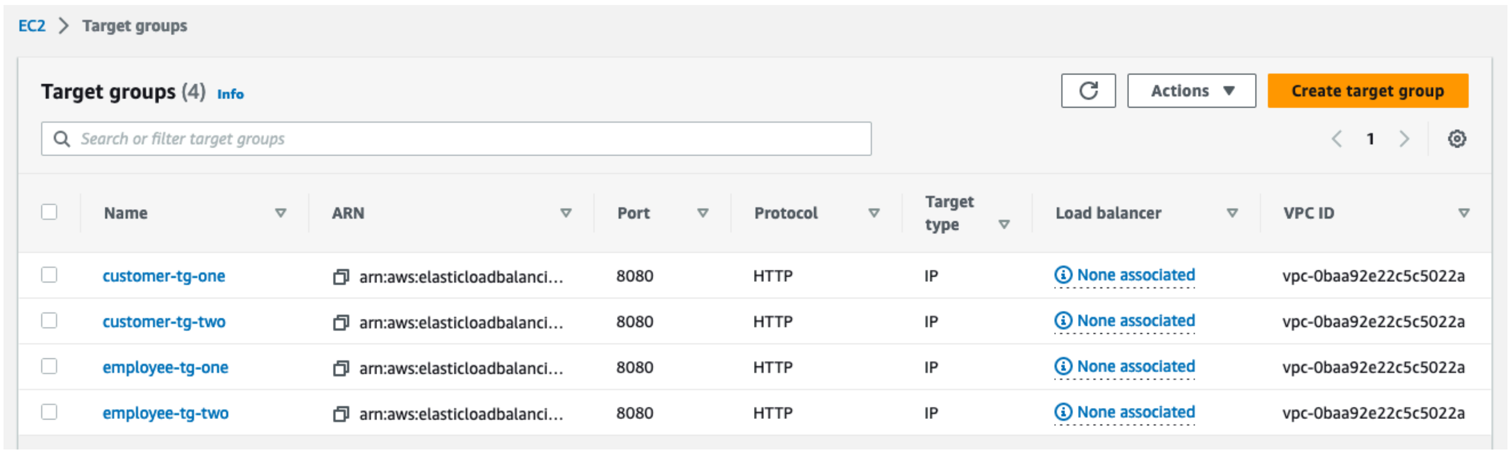
Return to table of contents

In this task, you will create four target groups—two for each microservice. Because you will configure a blue/green deployment, CodeDeploy requires two target groups for each deployment group.

Note: Blue/green is a deployment strategy where you create two separate but identical environments. One environment (blue) runs the current application version, and one environment (green) runs the new application version. For more information, see Blue/Green Deployments in the *Overview of Deployment Options on AWS* whitepaper.

1. Create the first target group for the *customer* microservice.
   - Navigate to the Amazon EC2 console.
   - In the navigation pane, choose **Target Groups**.
   - Choose **Create target group** and configure the following:
     - **Choose a target type:** Choose **IP addresses**.
     - **Target group name:** Enter `customer-tg-one`
     - **Protocol:** Choose **HTTP**.
     - **Port:** Enter `8080`
     - **VPC:** Choose **LabVPC**.
     - **Health check path:** Enter `/`
   - Choose **Next**.
   - On the **Register targets** page, accept all defaults (don't register any targets), and choose **Create target group**.
2. Create a second target group for the *customer* microservice. Use the same settings as the first target group except use `customer-tg-two` as the target group name.
3. Create a target group for the *employee* microservice. Use the same settings as the other target groups with the following exceptions:
   - **Target group name:** Enter `employee-tg-one`
   - **Health check path:** Enter `/admin/suppliers`
4. Create a second target group for the *employee* microservice. Use the same settings as the other target groups with the following exceptions:
   - **Target group name:** Enter `employee-tg-two`
   - **Health check path:** Enter `/admin/suppliers`

Important: Carefully confirm the name and port number of each target group. The following image provides an example:



	Name	ARN	Port	Protocol	Target type	Load balancer	VPC ID
	customer-tg-one	arn:aws:elasticloadbalanci...	8080	HTTP	IP	None associated	vpc-0baa92e22c5c5022a
	customer-tg-two	arn:aws:elasticloadbalanci...	8080	HTTP	IP	None associated	vpc-0baa92e22c5c5022a
	employee-tg-one	arn:aws:elasticloadbalanci...	8080	HTTP	IP	None associated	vpc-0baa92e22c5c5022a
	employee-tg-two	arn:aws:elasticloadbalanci...	8080	HTTP	IP	None associated	vpc-0baa92e22c5c5022a

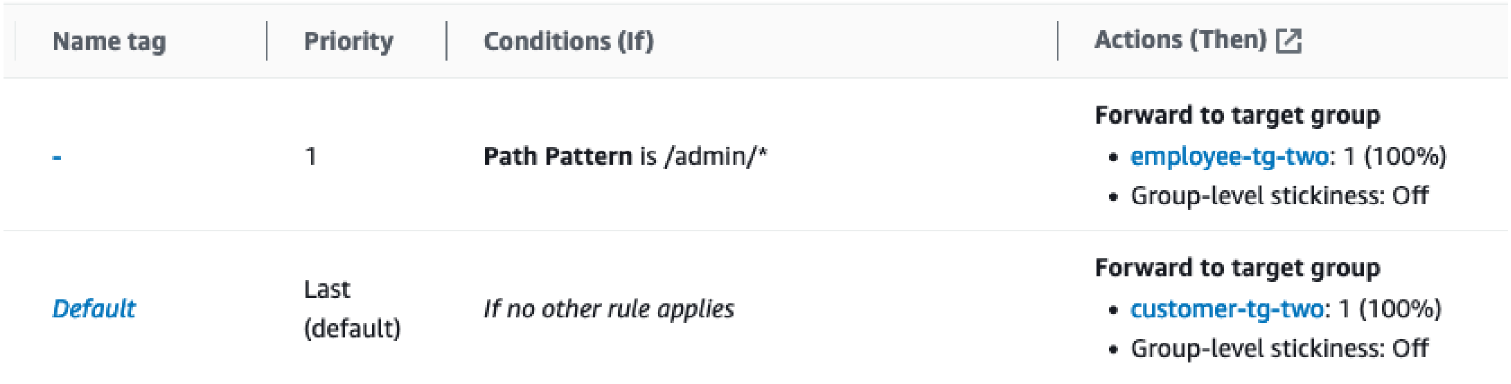
Task 6.2: Create a security group and an Application Load Balancer, and configure rules to route traffic

Return to table of contents

In this task, you will create an Application Load Balancer. You will also define two listeners for the load balancer: one on port 80 and another on port 8080. For each listener, you will then define path-based routing rules so that traffic is routed to the correct target group depending on the URL that a user attempts to load.

1. Create a new EC2 security group named `microservices-sg` to use in *LabVPC*. Add inbound rules that allow TCP traffic from any IPv4 address on ports 80 and 8080.
2. In the Amazon EC2 console, create an Application Load Balancer named `microservicesLB`.
   - Make it internet facing for IPv4 addresses.
   - Use *LabVPC*, *Public Subnet1*, *Public Subnet2*, and the *microservices-sg* security group.
   - Configure two listeners on it. The first should listen on *HTTP:80* and forward traffic to *customer-tg-two* by default. The second should listen on *HTTP:8080* and forward traffic to *customer-tg-one* by default.
3. Add a second rule for the *HTTP:80* listener. Define the following logic for this new rule:
   - IF **Path** is `/admin/*`
   - THEN **Forward to...** the **employee-tg-two** target group.

     The settings should be the same as shown in the following image:

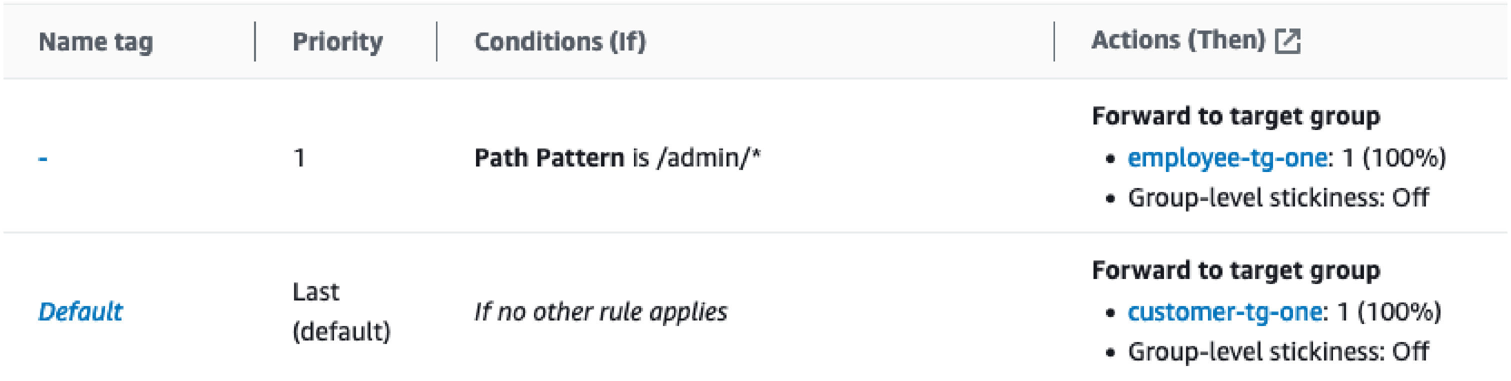


Name tag	Priority	Conditions (If)	Actions (Then)
-	1	**Path Pattern** is /admin/*	**Forward to target group** • employee-tg-two: 1 (100%) • Group-level stickiness: Off
Default	Last (default)	*If no other rule applies*	**Forward to target group** • customer-tg-two: 1 (100%) • Group-level stickiness: Off

4. Add a second rule for the *HTTP:8080* listener. Define the following logic for this new rule:

- IF **Path** is `/admin/*`
- THEN **Forward to** the **employee-tg-one** target group.

  The settings should be the same as shown in the following image:



Name tag	Priority	Conditions (If)	Actions (Then)
-	1	**Path Pattern** is /admin/*	**Forward to target group** • employee-tg-one: 1 (100%) • Group-level stickiness: Off
Default	Last (default)	*If no other rule applies*	**Forward to target group** • customer-tg-one: 1 (100%) • Group-level stickiness: Off

Phase 7: Creating two Amazon ECS services

In this phase, you will create a service in Amazon ECS for each microservice. Although you could deploy both microservices to a single ECS service, for this project, it will be easier to manage the microservices independently if each is deployed to its own ECS service.

Task 7.1: Create the ECS service for the *customer* microservice

Return to table of contents

1. In AWS Cloud9, create a new file named `create-customer-microservice-tg-two.json` in the **deployment** directory.
2. Paste the following JSON code into the file:

```
{
    "taskDefinition": "customer-microservice:REVISION-NUMBER",
    "cluster": "microservices-serverlesscluster",
    "loadBalancers": [
        {
            "targetGroupArn": "MICROSERVICE-TG-TWO-ARN",
            "containerName": "customer",
            "containerPort": 8080
        }
    ],
    "desiredCount": 1,
    "launchType": "FARGATE",
    "schedulingStrategy": "REPLICA",
    "deploymentController": {
        "type": "CODE_DEPLOY"
    },
    "networkConfiguration": {
        "awsvpcConfiguration": {
            "subnets": [
                "PUBLIC-SUBNET-1-ID",
                "PUBLIC-SUBNET-2-ID"
            ],
            "securityGroups": [
                "SECURITY-GROUP-ID"
            ],
            "assignPublicIp": "ENABLED"
            }
        }
    }
```

3. Edit the **create-customer-microservice-tg-two.json** file:
   - Replace **REVISION-NUMBER** with the number of the latest revision of the *customer-microservice* task definition that is registered with Amazon ECS.
     - If this is the first time that you are completing this step, the revision number should be `1`.
     - If you are repeating this step, find the latest revision number in the Amazon ECS console by choosing **Task definitions**, and then choosing **customer-microservice**.
   - Replace **MICROSERVICE-TG-TWO-ARN** with the actual ARN of the *customer-tg-two* target group.
   - Replace **PUBLIC-SUBNET-1-ID** with the actual subnet ID of *Public Subnet1*.
   - Replace **PUBLIC-SUBNET-2-ID** with the actual subnet ID of *Public Subnet2*.
   - Replace **SECURITY-GROUP-ID** with the actual security group ID of *microservices-sg*.
   - Save the changes.
4. To create the Amazon ECS service for the *customer* microservice, run the following commands:

   ```
   cd ~/environment/deployment
   aws ecs create-service --service-name customer-microservice --cli-input-json file://create-customer-
   microservice-tg-two.json
   ```

   Troubleshooting tip: If you are repeating this step and previously created the ECS service, you might receive an error about the creation of the service not being idempotent. To resolve this error, force delete the service from the Amazon ECS console, wait for it to drain, and then run the commands again.

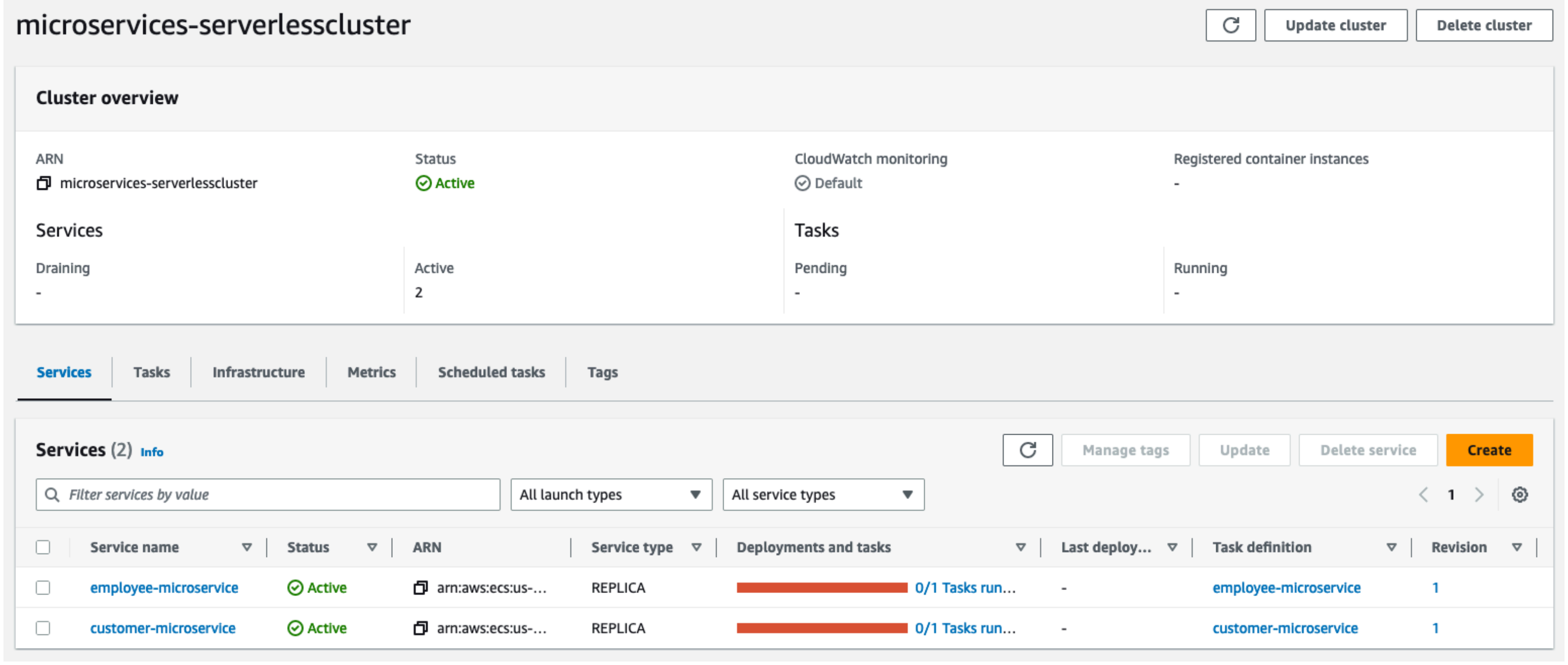
Task 7.2: Create the Amazon ECS service for the *employee* microservice

Return to table of contents

1. Create an Amazon ECS service for the *employee* microservice.
   - Copy the JSON file that you created for the *customer* microservice, and name it `create-employee-microservice-tg-two.json`. Save it in the same directory.
   - Modify the **create-employee-microservice-tg-two.json** file:
     - On line 2, change `customer-microservice` to `employee-microservice` and also update the **revision number**.
     - On line 6, enter the ARN of the **employee-tg-two** target group.

       Tip: Don't just change `customer` to `employee` on this line. The ARN is unique in other ways.
     - On line 7, change `customer` to `employee`
   - Save the changes.
2. Run the appropriate AWS CLI command to create the service in Amazon ECS.

   Note: If you go to the Amazon ECS console and look at the services in the cluster, you might see *0/1 Task running*, as shown in the following image. This is expected for now because you haven't launched task sets for these services yet.


microservices-serverlesscluster

Cluster overview

ARN	Status	CloudWatch monitoring	Registered container instances
microservices-serverlesscluster	Active	Default	-

Services

Draining	Active
-	2

Tasks

Pending	Running
-	-

Services (2) Info

Service name	Status	ARN	Service type	Deployments and tasks	Last deploy...	Task definition	Revision
employee-microservice	Active	arn:aws:ecs:us-...	REPLICA	0/1 Tasks run...	-	employee-microservice	1
customer-microservice	Active	arn:aws:ecs:us-...	REPLICA	0/1 Tasks run...	-	customer-microservice	1

Phase 8: Configuring CodeDeploy and CodePipeline

Now that you have defined the Application Load Balancer, target groups, and the Amazon ECS services that comprise the infrastructure that you will deploy your microservices to, the next step is to define the CI/CD pipeline to deploy the application.

The following diagram illustrates the role of the pipeline in the solution that you are building.

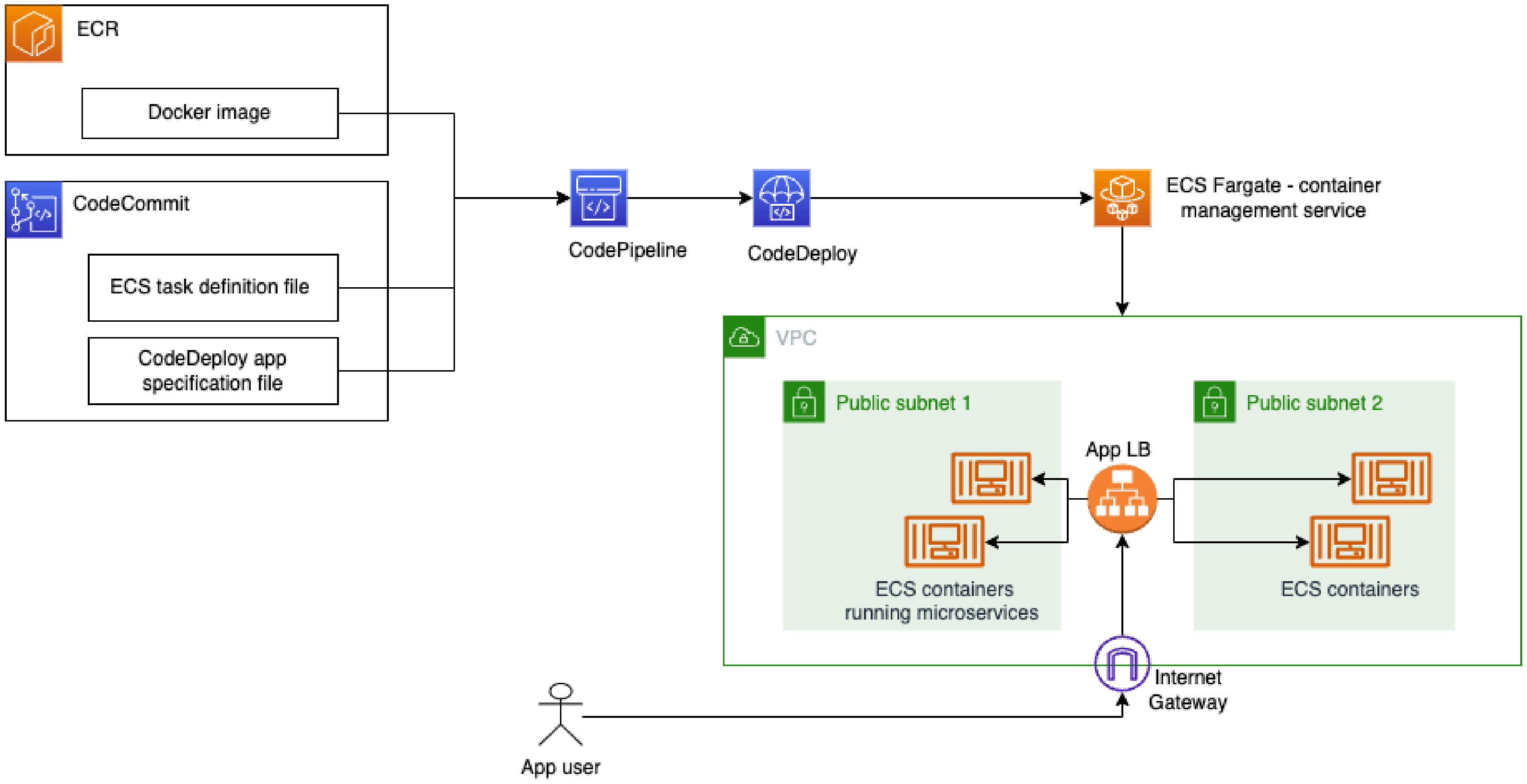


Diagram description: The pipeline will be invoked by updates to CodeCommit, where you have stored the ECS task definition files and the CodeDeploy AppSpec files. The pipeline can also be invoked by updates to one of the Docker image files that you have stored in Amazon ECR. When invoked, the pipeline will call the CodeDeploy service to deploy the updates. CodeDeploy will take the necessary actions to deploy the updates to the green environment. Assuming that no errors occur, the new task set will replace the existing task set.

Task 8.1: Create a CodeDeploy application and deployment groups

Return to table of contents

A CodeDeploy application is a collection of deployment groups and revisions. A deployment group specifies an Amazon ECS service, load balancer, optional test listener, and two target groups. A group specifies when to reroute traffic to the replacement task set, and when to terminate the original task set and Amazon ECS application after a successful deployment.

1. Use the CodeDeploy console to create a CodeDeploy application with the name `microservices` that uses Amazon ECS as the compute platform.

   Tip: See Create an Application for an Amazon ECS Service Deployment (Console) in the *AWS CodeDeploy User Guide*.

   Important: *DON'T* create a deployment group yet. You will do that in the next step.

2. Create a CodeDeploy deployment group for the *customer* microservice.
   - On the *microservices* application detail page, choose the **Deployment groups** tab.
   - Choose **Create deployment group** and configure the following:
     - **Deployment group name:** Enter `microservices-customer`
     - **Service role:** Place your cursor in the search box, and choose the ARN for **DeployRole**.
     - In the **Environment configuration** section:
       - **ECS cluster name:** Choose **microservices-serverlesscluster**.
       - **ECS service name:** Choose **customer-microservice**.
     - In the **Load balancers** section:
       - **Load balancer:** Choose **microservicesLB**.
       - **Production listener port:** Choose **HTTP:80**.
       - **Test listener port:** Choose **HTTP:8080**.
       - **Target group 1 name:** Choose **customer-tg-two**.
       - **Target group 2 name:** Choose **customer-tg-one**.
     - In the **Deployment settings** section:
       - **Traffic rerouting:** Choose **Reroute traffic immediately**.
       - **Deployment configuration:** Choose **CodeDeployDefault.ECSAllAtOnce**.
       - **Original revision termination:** Days: **0**, Hours: **0**, Minutes: **5**
     - Choose **Create deployment group**.

3. Create a CodeDeploy deployment group for the *employee* microservice. Specify the same settings that you did in the prior step, except for the following:
   - **Deployment group name:** Enter `microservices-employee`
   - **ECS service name:** Choose **employee-microservice**.
   - **Target group 1 name:** Choose **employee-tg-two**.
   - **Target group 2 name:** Choose **employee-tg-one**.

Task 8.2: Create a pipeline for the *customer* microservice

Return to table of contents

In this task, you will create a pipeline to update the *customer* microservice. When you first define the pipeline, you will configure CodeCommit as the source and CodeDeploy as the service that is responsible for deployment. You will then edit the pipeline to add the Amazon ECR service as a second source.

With an Amazon ECS blue/green deployment, which you will specify in this task, you provision a new set of containers, which CodeDeploy installs the latest version of your application on. CodeDeploy then reroutes load balancer traffic from an existing set of containers, which run the previous version of your application, to the new set of containers, which run the latest version. After traffic is rerouted to the new containers, the existing containers can be terminated. With a blue/green deployment, you can test the new application version before sending production traffic to it.

References

- The AWS Academy Cloud Architecting and AWS Academy Cloud Developing courses include hands-on labs that explore CodePipeline features.
- AWS CodePipeline User Guide

1. In the CodePipeline console, create a *customer* pipeline with the following settings:
   - **Pipeline name:** Enter `update-customer-microservice`
   - **Service role:** Choose the ARN for **PipelineRole**.
   - **Source provider:** Choose **AWS CodeCommit**.
     - **Repository name:** Choose **deployment**.

     Note: You have defined two CodeCommit repositories. The *deployment* repository contains the Amazon ECS task definition files and CodeDeploy AppSpec files that your pipeline will need, so that is the one you choose here.
     - **Branch name:** Choose **dev**.

     Note: *Skip* the build stage.
   - **Deploy provider: Amazon ECS (Blue/Green)**
     - **Region:** US East (N. Virginia).
     - **AWS CodeDeploy application name:** microservices
     - **AWS CodeDeploy deployment group:** microservices-customer
     - Under **Amazon ECS task definition**:
       - Set a **SourceArtifact** with a value of `taskdef-customer.json`
     - Under **AWS CodeDeploy AppSpec file**:
       - Set a **SourceArtifact** with a value of `appspec-customer.yaml`

         Note: Leave the *Dynamically update task definition image* fields *blank* for now.

   Note: After you create the pipeline, it will immediately start to run and will eventually fail on the *Deploy* stage. Ignore that for now and continue to the next step.
2. Edit the *update-customer-microservice* pipeline to add another *source*.
   - In the **Edit: Source** section, choose **Edit stage**, then add an action with these details:
     - **Action name:** `Image`
     - **Action provider:** Amazon ECR

        - **Repository name:** customer
        - **Image tag:** latest
        - **Output artifacts:** `image-customer`

3. Edit the *deploy* action of the *update-customer-microservice* pipeline.
    - Edit the **update-customer-microservice** pipeline
        - In the **Edit: Deploy** section, choose **Edit stage**, then add an input artifact as described below:
            - On the **Deploy Amazon ECS (Blue/Green)** card, choose the edit (pencil) icon.
            - Under **Input artifacts**, choose **Add** and then choose **image-customer**.

          Note: You should now have *SourceArtifact* and *image-customer* as listed input artifacts.
        - Under **Dynamically update task definition image**, for **Input artifact with image details**, choose **image-customer**.
        - For **Placeholder text in the task definition**, enter `IMAGE1_NAME`

Analysis: Recall that in a previous phase, you entered the *IMAGE1_NAME* placeholder text in the *taskdef-customer.json* file before you pushed it to CodeCommit. In this current task, you configured the logic that will replace the placeholder text with the actual image name that the source phase of the CodePipeline returns.

Task 8.3: Test the CI/CD pipeline for the *customer* microservice

Return to table of contents

In this task, you will test that the CI/CD pipeline for the *customer* microservice functions as intended.

Important: If you are repeating this task, confirm that all target groups are still associated with the Application Load Balancer. The Reassociate Target Groups with Load Balancer section of the appendix provides more detail.

1. Launch a deployment of the *customer* microservice on Amazon ECS on Fargate.
    - Navigate to the CodePipeline console.
    - On the **Pipelines** page, choose the link for the pipeline that is named **update-customer-microservice**.
    - To force a test of the current pipeline settings, choose **Release change**, and then choose **Release**.

      Note: By invoking the pipeline, you created a new revision of the task definition.

      Wait for the two *Source* tasks to show a status of *Succeeded - just now*.
    - In the **Deploy** section, wait for a **Details** link to appear, and then click the link.

      A CodeDeploy page opens in a new browser tab.
2. Observe the progress in CodeDeploy.
    - Scroll to the bottom of the page, and notice the **Deployment lifecycle events** section.

      Tip: If you see a "Primary task group must be behind listener" error, refer to the Reassociate Target Groups with Load Balancer section in the appendix.

      Within a few minutes, if everything was configured correctly, all of the deployment lifecycle events should succeed. Don't wait for that to happen—move to the next step. Keep this page open.
3. Load the *customer* microservice in a browser tab and test it.
    - Locate the **DNS name** value of the **microservicesLB** load balancer, and paste it into a new browser tab.
    - The customer microservice loads. If it doesn't load, add `:8080` to the end of the URL and try again.

Analysis: Recall that your load balancer has two listeners: one on port 80 and another on port 8080. Port 8080 is where the replacement task set will run for the first 5 minutes. Therefore, if you load the :80 URL within the first 5 minutes, the customer microservice page might not load, but you should already see the page at 8080. Then, after 5 minutes, you should see that the microservice is available at both ports.

- In the café web application, choose **List of suppliers** or **Suppliers list**.

  The suppliers page loads. It should *not* have the edit or add supplier buttons because it is a customer page.

4. Observe the running tasks in the Amazon ECS console.
   - Navigate to the Amazon ECS console.
   - In the clusters list, choose the link for **microservices-serverlesscluster**.

     On the **Services** tab, notice that the *customer-microservice* service appears. The **Deployments and tasks** status will change as the blue/green deployment advances through its lifecycle events.
   - Choose the **Tasks** tab.

     Here you can see the actual tasks that are running. You might have more than one task running per service that you defined.
   - Choose the link for one of the listed tasks. You might only have one.

     Here you can see the actual container details and the configuration information, such as the IP addresses that are associated with the running container.
5. Return to the CodeDeploy page that is open in another browser tab.

   You should now see that all five steps of the deployment succeeded and the replacement task set is now serving traffic.
6. Observe the load balancer and target group settings.
   - In the Amazon EC2 console, choose **Target Groups**.

     You might notice that the *customer-tg-two* target group is no longer associated with the load balancer. This is because CodeDeploy is managing the load balancer listener rules and might have determined that some of the target groups are no longer needed.
   - Observe the HTTP:80 listener rules.

     The default rule has changed here. The default "if no other rule applies" rule previously pointed to *customer-tg-two*, but now it points to *customer-tg-one*. This is because CodeDeploy actively managed your Application Load Balancer.
   - Observe the HTTP:8080 listener rules.

     The two rules still forward to the "one" target groups.

Congratulations! You successfully deployed one of the two microservices to Amazon ECS on Fargate by using a CI/CD pipeline.

Task 8.4: Create a pipeline for the *employee* microservice

Return to table of contents

In this task, you will create the pipeline for the *employee* microservice.

1. Create a pipeline for the *employee* microservice with the following specifications:
   - Pipeline name: `update-employee-microservice`
   - Role ARN: **PipelineRole**
     - Source provider: **AWS CodeCommit**
       - Repository name: **deployment**

    - Branch name: **dev**
- Deploy provider: **Amazon ECS (Blue/Green)**
- AWS CodeDeploy application name: **microservices**
- AWS CodeDeploy deployment group: **microservices-employee**
- Amazon ECS task definition: **SourceArtifact**
    - Path: `taskdef-employee.json`
    - AWS CodeDeploy AppSpec file: **SourceArtifact**
    - Path: `appspec-employee.yaml`

2. Add another *source* to the employee microservice pipeline. Add an action with the following details:
    - Action name: `Image`
    - Action provider: **Amazon ECR**
    - Repository name: **employee**
    - Image tag: **latest**
    - Output artifacts: `image-employee`
3. Edit the Amazon ECS (Blue/Green) action in the deploy stage:
    - Add another *input artifact* and choose **image-employee**.
    - Under **Dynamically update task definition image**, for **Input artifact with image details**, choose **image-employee**.
    - For **Placeholder text in the task definition**, enter `IMAGE1_NAME`

Task 8.5: Test the CI/CD pipeline for the *employee* microservice

Return to table of contents

In this task, you will test the pipeline that you just defined for the *employee* microservice.

Important: If you are repeating this task, confirm that all target groups are still associated with the Application Load Balancer. The Reassociate Target Groups with Load Balancer section of the appendix provides detail if needed.

1. Launch a deployment of the *employee* microservice on Amazon ECS on Fargate.
    - Use the *release change* feature to force a test of the pipeline.
    - Follow the progress in CodeDeploy. Within a few minutes, if everything was configured correctly, all of the *Deployment lifecycle events* should succeed.
2. Load the *employee* microservice in a browser tab.
    - In the browser tab where the customer microservice is running, choose **Administrator link**.

      You are directed to a page in the format *http://microserviceslb-UNIQUE-NUMBER.us-east-1.elb.amazonaws.com/admin/suppliers*.

      The employee microservice loads. If it doesn't load, try adding `:8080` just after `amazonaws.com` in the URL.
    - Choose **List of suppliers** or **Suppliers list**.

      The suppliers page should load. This version of the page should *not* have the edit or add supplier buttons. All links in the café web application should now work because you have now deployed both microservices.
3. Observe the running tasks in the Amazon ECS console.

   The **Deployments and tasks** status will change as the blue/green deployment advances through its lifecycle events.

4. Return to the CodeDeploy page to confirm that all five steps of the deployment succeeded and the replacement task set is now serving traffic.

Congratulations! You successfully deployed the employee microservice to Amazon ECS on Fargate by using a CI/CD pipeline.

Task 8.6: Observe how CodeDeploy modified the load balancer listener rules

Return to table of contents

1. Observe the load balancer and target group settings.
   - In the Amazon EC2 console, choose **Target Groups**.

     Note: If you already had the page open, refresh it.

     Notice that the *customer-tg-two* target group is no longer associated with the load balancer. This is because CodeDeploy is managing the load balancer listener rules.

     Note: If you are repeating this step, the target groups that are currently attached and unattached might be different.
   - Observe the HTTP:80 listener rules.

     The default rule has changed here. For the default "If no other rule applies" rule, the "forward to target group" previously pointed to *customer-tg-two*, but now it points to *customer-tg-one*.
   - Observe the HTTP:8080 listener rules.

     The two rules still forward to the "one" target groups.

Phase 9: Adjusting the microservice code to cause a pipeline to run again

In this phase, you will experience the benefits of the microservices architecture and the CI/CD pipeline that you built. You will begin by adjusting the load balancer listener rules that are related to the *employee* microservice. You will also update the source code of the *employee* microservice, generate a new Docker image, and push that image to Amazon ECR, which will cause the pipeline to run and update the production deployment. You will also scale up the number of containers that support the *customer* microservice.

Task 9.1: Limit access to the *employee* microservice

Return to table of contents

In this task, you will limit access to the *employee* microservice to only people who try to connect to it from a specific IP address. By limiting the source IP to a specific IP address, only users who access the application from that IP can access the pages, and edit or delete supplier entries.

1. Confirm that all target groups are still associated with the Application Load Balancer.

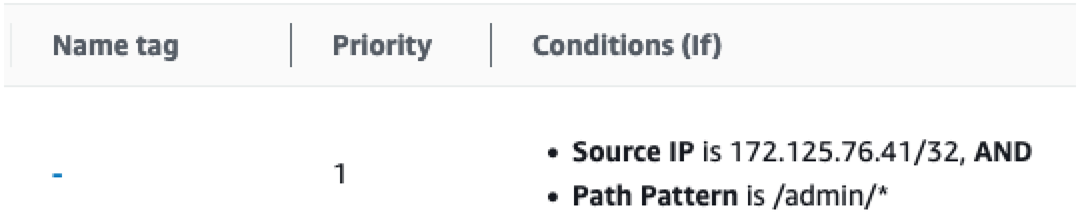
   In the Amazon EC2 console, check that all four target groups are still associated with the load balancer. Reassociate target groups as needed before going to the next step.

   Tip: For details, see Reassociating target groups with the load balancer in the appendix.
2. Discover your public IPv4 address.

   Tip: One resource you can use to do this is https://www.whatismyip.com.
3. Edit the rules for the **HTTP:80** listener.

For the rule that currently has "IF Path is /admin/*" in the details, add a second condition to route the user to the target groups only if the source IP of the request is your IP address.

Tip: For the source IP, paste in your public IPv4 address and then add `/32`. The following image shows an example:



Name tag	Priority	Conditions (If)
-	1	• **Source IP** is 172.125.76.41/32, **AND** • **Path Pattern** is /admin/*

4. Edit the rules for the **HTTP:8080** listener.

   Edit the rules in the same way that you edited the rules for the **HTTP:80** listener. You want access to the employee target groups to be limited to your IP address.

Task 9.2: Adjust the UI for the *employee* microservice and push the updated image to Amazon ECR

Return to table of contents

In this task, you will adjust the deployed microservices.

1. Edit the employee/views/**nav.html** file.
   - On line 1, change `navbar-dark bg-dark` to `navbar-light bg-light`
   - Save the change.
2. To generate a new Docker image from the *employee* microservice source files that you modified and to label the image, run the following commands:

```
docker rm -f employee_1
cd ~/environment/microservices/employee
docker build --tag employee .
dbEndpoint=$(cat ~/environment/microservices/employee/app/config/config.js | grep 'APP_DB_HOST' | cut -d
'"' -f2)
echo $dbEndpoint
account_id=$(aws sts get-caller-identity |grep Account|cut -d '"' -f4)
echo $account_id
docker tag employee:latest $account_id.dkr.ecr.us-east-1.amazonaws.com/employee:latest
```

3. Push an updated image to Amazon ECR so that the *update-employee-microservice* pipeline will be invoked.
   - In the CodePipeline console, navigate to the details page for the *update-employee-microservice* pipeline. Keep this page open.
   - To push the new employee microservice Docker image to Amazon ECR, run the following commands in your AWS Cloud9 IDE:

```
#refresh credentials in case needed
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin
$account_id.dkr.ecr.us-east-1.amazonaws.com
#push the image
docker push $account_id.dkr.ecr.us-east-1.amazonaws.com/employee:latest
```

- Confirm that the output is similar to the following.

  ```
  b20e9725db21: Pushed
  715b0f96c609: Pushed
  44141cf9260f: Layer already exists
  d81d715330b7: Layer already exists
  1dc7f3bb09a4: Layer already exists
  dcaceb729824: Layer already exists
  f1b5933fe4b5: Layer already exists
  ```

  At least one layer should indicate that it was pushed, which indicates that the image was modified since it was last pushed to Amazon ECR. You could also look in the Amazon ECR repository to confirm the last modified timestamp of the image that is tagged as the latest.

Task 9.3: Confirm that the *employee* pipeline ran and the microservice was updated

Return to table of contents

1. Observe the *update-employee-microservice* pipeline details in the CodePipeline console.

   Notice that when you uploaded a new Docker image to Amazon ECR, the pipeline was invoked and ran. Note that the pipeline might take a minute or two to notice that the Docker image was updated before the pipeline is invoked.

2. Observe the details in the CodeDeploy console.

Task 9.4: Test access to the *employee* microservice

Return to table of contents

In this task, you will test access to the *employee* microservice.

1. Test access to the *employee* microservice pages at `http://<alb-endpoint>/admin/suppliers` and `http://<alb-endpoint>:8080/admin/suppliers` from the same device that you have used for this project so far. Replace <*alb-endpoint*> with the DNS name of the *microservicesLB* load balancer.

   At least one of the two pages should load successfully.

   Notice that the banner with the page title is a light color now because of the change that you made to the nav.html file. Pages that are hosted by the customer microservice still have the dark banner. This demonstrates that by using a microservices architecture, you could independently modify the UI or features of each microservice without affecting others.

2. Test access to the same *employee* microservice pages from a different device.

   For example, you could use your phone to connect from the cellular network and not the same Wi-Fi network that your computer uses. You want the device to use a different IP address to connect to the internet than your computer.

   You should get a 404 error on any page that loads, and the page should say "Coffee suppliers" instead of "Manage coffee suppliers." This is evidence that you cannot successfully connect to the employee microservice from another IP address.

   Tip: If you don't have another network available, run the following command in the AWS Cloud9 terminal: `curl http://<alb-endpoint>/admin/suppliers`. The source IP address of the AWS Cloud9 instance is different than your browser's source IP. The result should include `<p class="lead">Sorry, we don't seem to have that page in stock</p>`.

This proves that the updated rules on the load balancer listener are working as intended.

Tip: If the update doesn't function as intended, you could go to the **Deployments** page in the CodeDeploy console within 5 minutes to stop the deployment and roll back to the previous version. You would choose **Stop and roll back deployment** and then choose **Stop and rollback**. This action would reroute production traffic to the original task set and then delete the replacement task set. You configured the 5-minute setting in phase 8, task 1.

Task 9.5: Scale the *customer* microservice

Return to table of contents

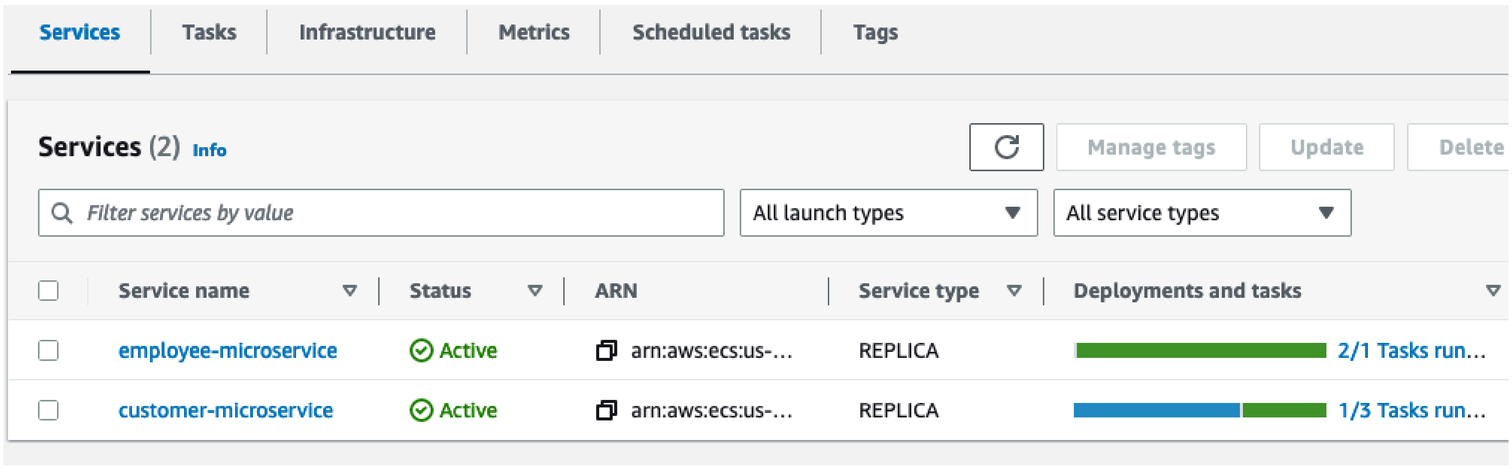
In this task, you will scale up the number of containers that run to support the *customer* microservice. You can make this change without causing the *update-customer-microservice* pipeline to run.

1. Update the *customer* service in Amazon ECS.
   - Run the following command:

     ```
     aws ecs update-service --cluster microservices-serverlesscluster --service customer-microservice --
     desired-count 3
     ```

     A large JSON-formatted response is returned.
   - Go to the Amazon ECS services view.

     It might look similar to the following image.



Service name	Status	ARN	Service type	Deployments and tasks
employee-microservice	Active	arn:aws:ecs:us-...	REPLICA	2/1 Tasks run...
customer-microservice	Active	arn:aws:ecs:us-...	REPLICA	1/3 Tasks run...

In this example, the *customer-microservice* shows 1/3 tasks running because you increased the desired count from 1 to 3, but the two new containers are still being started. If you wait long enough and look at the **Tasks** tab, you will see that three containers run to support the customer microservice. This demonstrates how you can scale microservices independently of one another.

Note: The *employee-microservice* might show 2/1 tasks running. This could happen because a replacement task set (which has one container) has been created, but the original task set remains active for the first 5 minutes in case you decide to roll back.

Ending your session

Reminder: This is a long-lived lab environment. Data is retained until you either use the allocated budget or the course end date is reached (whichever occurs first).

To preserve your budget when you are finished for the day, or when you are finished actively working on the assignment for the time being, do the following:

1. At the top of this page, choose **End Lab**, and then choose Yes to confirm that you want to end the lab.

   A message panel indicates that the lab is terminating.

   Note: Choosing **End Lab** in this environment will *not* delete the resource you have created. They will still be there the next time you choose Start lab (for example, on another day).
2. To close the panel, choose **Close** in the upper-right corner.

Appendix

Updating a test container that is running on AWS Cloud9

Return to table of contents

If you need to update the source code of a microservice *after* running the container, you need to (1) stop and delete the container, (2) build a new image, and (3) run the new image. The following sections provide a reference to complete these steps.

Updating the *customer* container that is running on AWS Cloud9

```
# Stop and delete the specified container (assumes that the container name is customer_1)
docker rm -f customer_1

# Must be in the directory that has the Dockerfile before you attempt to build a new image
cd ~/environment/microservices/customer

# Build a new image from the latest source files and overwrite any existing image
docker build --tag customer .

# Ensure the dbEndpoint variable is set in your terminal session
dbEndpoint=$(cat ~/environment/microservices/customer/app/config/config.js | grep 'APP_DB_HOST' | cut -d '"' -
f2)
echo $dbEndpoint

# Create and run a new container from the image (and pass the DB location to the container)
docker run -d --name customer_1 -p 8080:8080 -e APP_DB_HOST="$dbEndpoint"  customer

# URL to test the microservice running on the AWS Cloud9 test container
echo "Customer microservice test container running at http://"$(curl ifconfig.me):8080
```

Updating the *employee* container that is running on AWS Cloud9

```
# Stop and delete the specified container (assumes that the container name is employee_1)
docker rm -f employee_1

# Must be in the directory that has the Dockerfile before you attempt to build a new image
cd /home/ec2-user/environment/microservices/employee

# Build a new image from the latest source files and overwrite any existing image
docker build --tag employee .
```

```
# Ensure the dbEndpoint variable is set in your terminal session
dbEndpoint=$(cat ~/environment/microservices/customer/app/config/config.js | grep 'APP_DB_HOST' | cut -d '"' -
f2)
echo $dbEndpoint

# Create and run a new container from the image (and pass the DB location to the container)
docker run -d --name employee_1 -p 8081:8081 -e APP_DB_HOST="$dbEndpoint"  employee

# URL to test the microservice running on the AWS Cloud9 test container
echo "Employee microservice test container running at http://"$(curl ifconfig.me):8081/admin/suppliers
```

Reference: Other Docker commands

To find the commands that you need to run, see Use the Docker Command Line in the Docker documentation.

The following selected commands could be helpful:

```
docker container ls # List the running containers
docker rm -f <container> # Stop and delete the specified container
docker images # List the images
docker build --tag <tag-for-image> . # Create a new image from a Dockerfile (must be in the same directory as
the Dockerfile)
docker run -d --name <name-to-give-container> -p <port>:<port> <image-tag> # Run a container from an image
docker ps # List the running Docker containers
```

Reference: Network analysis and viewing Docker logs

```
# Helpful commands to see what is happening
sudo lsof -i :8080
sudo lsof -i :8081
ps -ef | head -1; ps -ef | grep docker

# Docker log viewing - returns container numbers
sudo ls /var/lib/docker/containers

# View the log for a container where you know the container number
sudo less /var/lib/docker/containers/<container-number>/*.log

# See the IP addresses that Docker containers use
docker inspect network bridge
```

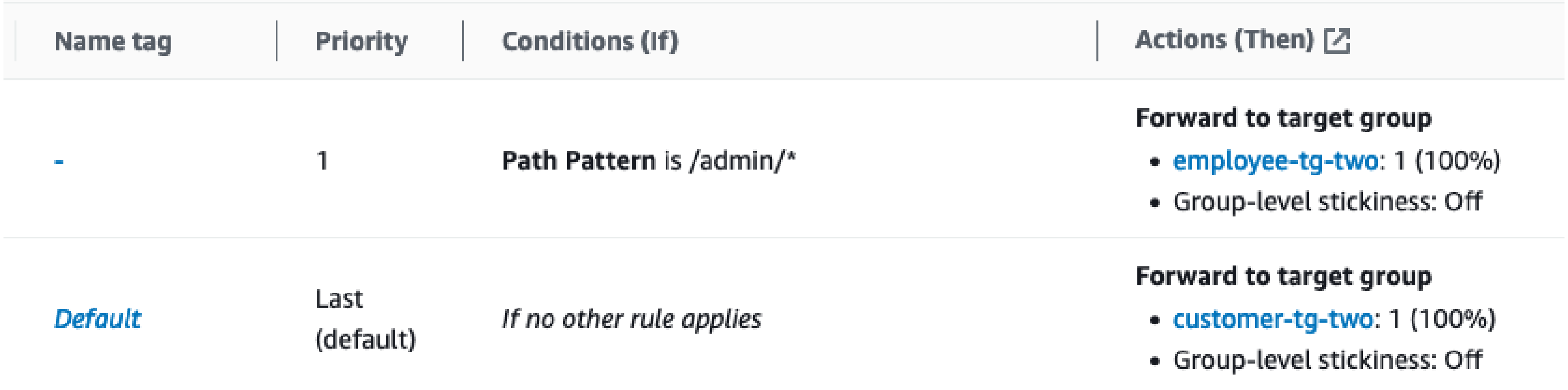
Reassociating target groups with the load balancer

Return to table of contents

Because CodeDeploy manages the Application Load Balancer rules, you might periodically find that some target groups are no longer associated with the load balancer. The following steps explain how to reassociate the target groups.

1. Note the original configuration that you made for the **HTTP:80** listener.

   The following image shows the original configuration for this listener:

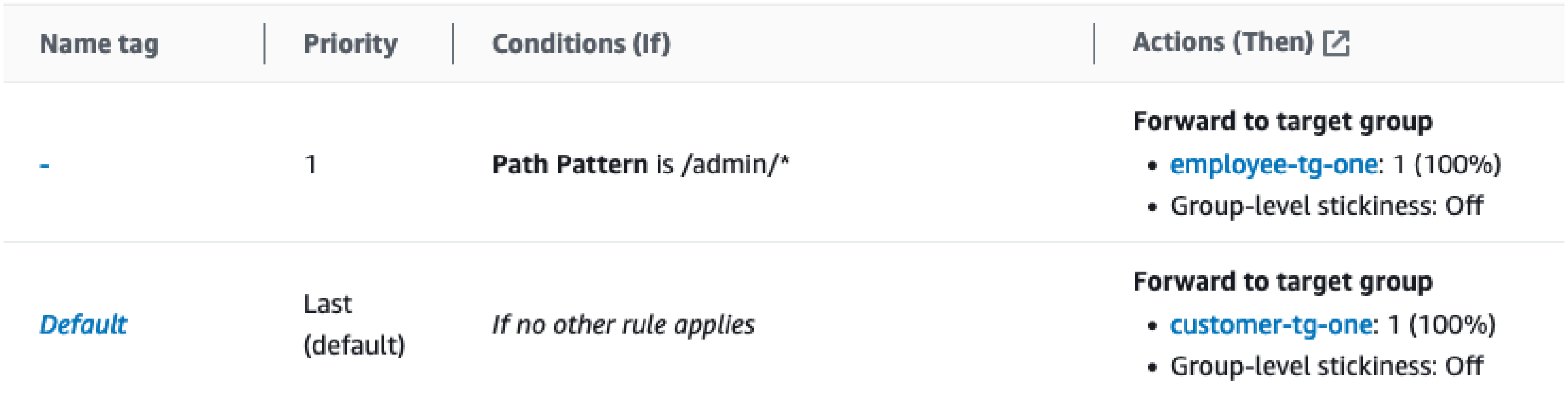


Name tag	Priority	Conditions (If)	Actions (Then)
-	1	**Path Pattern** is /admin/*	**Forward to target group** • employee-tg-two: 1 (100%) • Group-level stickiness: Off
Default	Last (default)	*If no other rule applies*	**Forward to target group** • customer-tg-two: 1 (100%) • Group-level stickiness: Off

If you successfully deployed the microservices previously, and the settings that are shown in the previous image are currently in use for the deployment (where the target groups in use by the HTTP:80 listener have "two" in their names), then this time you want to set the HTTP:80 listener to use target groups that have "one" in their names. Effectively, the green task set has become the blue task set, and the blue task set has become the green task set.

2. Edit the **HTTP:80** load balancer rules to make sure that they still match the previous image. If they don't, adjust them:
    - Navigate to the Amazon EC2 console.
    - In the navigation pane, choose **Load Balancers**.
    - Choose the link for the **microservicesLB** load balancer.
    - On the **Listeners and rules** tab, choose the **HTTP:80** link.
    - In the **Listener rules** panel, verify that the **Default** rule forwards to **customer-tg-one**. If it doesn't:
        - Select the rule.
        - From the **Actions** menu, choose **Edit rule**.
        - Set **Forward to target group** to **customer-tg-one**, and choose **Save changes**.
    - Still on the **HTTP:80** page, in the **Listener rules** panel, verify that the rule with **Path Pattern is /admin/** forwards to **employee-tg-one**. If it doesn't:
        - Select the rule.
        - From the **Actions** menu, choose **Edit rule**, and then choose **Next**.
        - In the **Actions** panel, set **Forward to target group** to **employee-tg-one**.
        - Choose **Next**, and then choose **Save changes**.
3. Note the original configuration that you made for the **HTTP:8080** listener.

    The following image shows the original configuration for this listener:



Name tag	Priority	Conditions (If)	Actions (Then)
-	1	**Path Pattern** is /admin/*	**Forward to target group** • employee-tg-one: 1 (100%) • Group-level stickiness: Off
Default	Last (default)	*If no other rule applies*	**Forward to target group** • customer-tg-one: 1 (100%) • Group-level stickiness: Off

If you successfully deployed the microservices previously, and the settings that are shown in the previous image are currently in use for the deployment (where the target groups in use by the HTTP:8080 listener have "one" in their names), then this time you want to set the HTTP:8080 listener to use target groups that have "two" in their names.

4. Edit the **HTTP:8080** load balancer rules to make sure that they still match the previous image. If they don't, adjust them:

- Navigate to the Amazon EC2 console.
- In the navigation pane, choose **Load Balancers**.
- Choose the link for the **microservicesLB** load balancer.
- On the **Listeners and rules** tab, choose the **HTTP:8080** link.
- In the **Listener rules** panel, verify that the **Default** rule forwards to **customer-tg-two**. If it doesn't:
    - Select the rule.
    - From the **Actions** menu, choose **Edit rule**.
    - Set **Forward to target group** to **customer-tg-two**, and choose **Save changes**.
- Still on the **HTTP:8080** page, in the **Listener rules** panel, verify that the rule with **Path Pattern is /admin/** forwards to **employee-tg-two**. If it doesn't:
    - Select the rule.
    - From the **Actions** menu, choose **Edit rule**, and then choose **Next**.
    - In the **Actions** panel, set **Forward to target group** to **employee-tg-two**.
    - Choose **Next**, and then choose **Save changes**.

5. Confirm that all target groups are reassociated.
    - In the navigation pane of the Amazon EC2 console, choose **Target Groups**.
    - Confirm that all four target groups are now associated with the *microservicesLB* load balancer.

Important: Each time that you run the pipelines, start by confirming that all four target groups are associated with the load balancer listeners and adjust as necessary.

In a production environment, you might choose to automate the steps that you just completed, where you want to check and adjust the Application Load Balancer configuration prior to each run of a pipeline. CodeDeploy provides a feature named AppSpec *hooks* that you can use to create a custom AWS Lambda function with code to accomplish tasks like this. You can then specify that the hook be run before, during, or after a specific lifecycle event by referencing it in your AppSpec definition. Authoring a custom Lambda function and using this feature is beyond the scope of this project. For more information, see AppSpec 'hooks' Section in the *AWS CodeDeploy User Guide*.

Troubleshooting tips

Return to table of contents

Issue: Records from the database don't display in the web application or microservice.

Check the following:

- If this happens when you deploy to Amazon ECS, look in the Amazon CloudWatch logs for error details. You might try rebooting the Amazon RDS database.
- If this happens when you test running a container on the AWS Cloud9 instance, verify that your AWS Cloud9 instance is running in *Public Subnet1*.

Issue: Invalid action configuration

Check the following:

- The artifact cannot be larger than 3 MB. Look at the **codepipeline-xxxx** bucket in Amazon S3. Browse to the **update-customer-micr/SourceArti** or **update-employee-micr/SourceArti** prefix (folder). Verify that no object is larger than 3 MB. Only push files to the CodeCommit repository if they were mentioned in these instructions. Avoid posting files other than the AppSpec and task definition files to this repository.

Issue: A message in **Deployments and tasks** in the Amazon ECS console says *service….-microservice is unable to consistently start tasks successfully*.

Check the following:

- Verify that all target groups are associated with the load balancer.

Issue: The CodeDeploy deployment status is stuck at step 1, or the application doesn't work when deployed to Amazon ECS.

Check the following:

- In the Amazon ECS console, choose the link for **microservices-serverlesscluster**. Choose the name of the service (**employee-microservice** or **customer-microservice**) that is stuck. Then, choose the **Deployments and events** tab. This view shows the progress of CodeDeploy deploying the task set. The **Configuration and tasks** tab can also provide helpful information because it shows whether any tasks (containers) have actually launched.
- In the AWS CloudTrail console, in the navigation pane, choose **Event history**. Look for events that might have resulted in errors. To observe details for an event, choose the **Event name** link for the event.
- In the CloudWatch console, in the navigation pane, choose **Logs** and then choose **Log groups**. Choose the link for the **awslogs-capstone** log group. To see detailed logs, choose the **Log stream** link for the latest event time.