OS-C ClimateScore API Service Installation Guide

I. Introduction

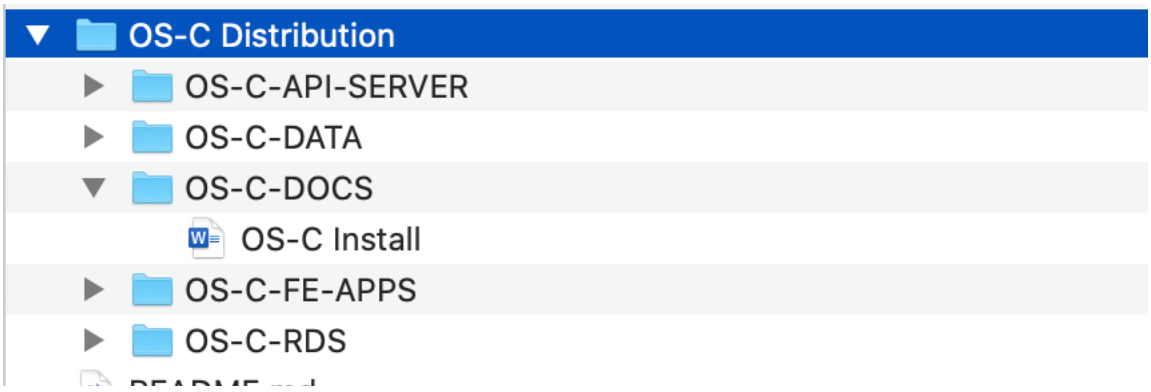
The purpose of this document is to guide a Cloud Engineer in installing the OS-C ClimateScore API Service ("ClimateScore Service") into the AWS Cloud Platform. The guide assumes the installer has some familiarity with the AWS Console, specifically AWS RDS, AWS ECS, AWS EC2 (SG and ALB), PgAdmin, Python and SQL.

The ClimateScore Service has three major components:

- An AWS RDS PostgreSQL server that hosts all the Jupiter contributed Climate Data.
- An AWS EC2 Fargate Cluster that instantiates containerized a Flask Application that provides the Business Logic for the ClimateScore Service. The Fargate Cluster is in turn connected to an Application Load Balancer.
- The third and final component is a suite of Front-End Applications, consisting of a Postman Collection, a Jupyter Notebook, and an Open API definition file, to serve as models for Application Developers who want to use the ClimateScore Service.

Manifest

All the artifacts necessary to install the ClimateScore Service is in git repo with the following structure:



OS-C-DATA contains all the Jupiter contributed data in CSV format. The data is delivered as seven CSV files, each containing data for one peril (combined flood, drought, fire, hail, heat, precipitation, and wind).

OS-C-RDS contains the Python scripts necessary to create tables for each of the Perils as well as corresponding indexes.

Both OS-C-DATA and OS-C-RDS artifacts are used in the "**Creating and Loading the Database**" section of this guide.

OS-C-API-SERVER contains the Python Source Code of the Flask Application Server that provides the business logic of the ClimateScore Service.

The OS-C-API-SERVER artifacts are used in the "**Installing, Configuring and Testing the API Application Server**" section of this guide.

OS-C-FE-APPS contains examples and documentation in Postman collections, Jupyter Notebooks, and Open API/Swagger format that is intended to help Developers and Data Scientists develop applications that utilize the ClimateScore Service.

Installation and use of these Applications are documented in "**Installing, Testing, and Making Use of example Front End Applications**"

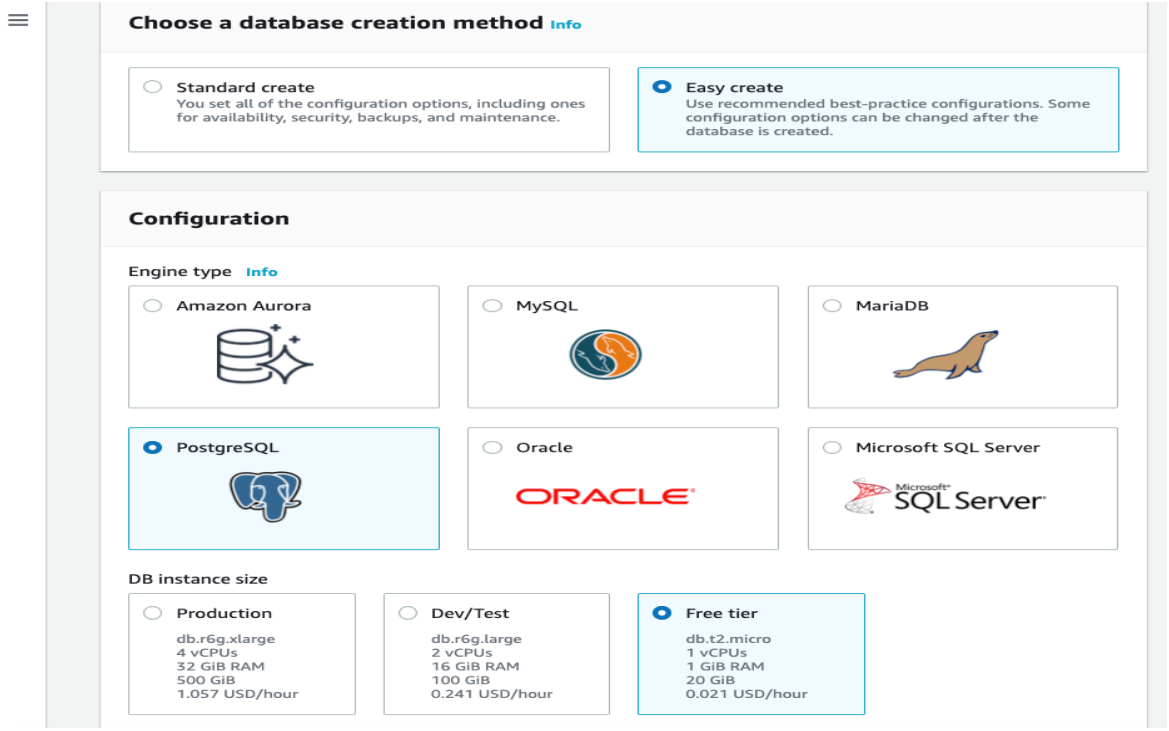
Important:
In this guide, we will assume that the ClimateScore service and all associated components will be hosted in AWS region "us-east-02". It assumes the installer has at least System Administrator rights in the account. It is also assumed that the installation will be done via AWS Console.

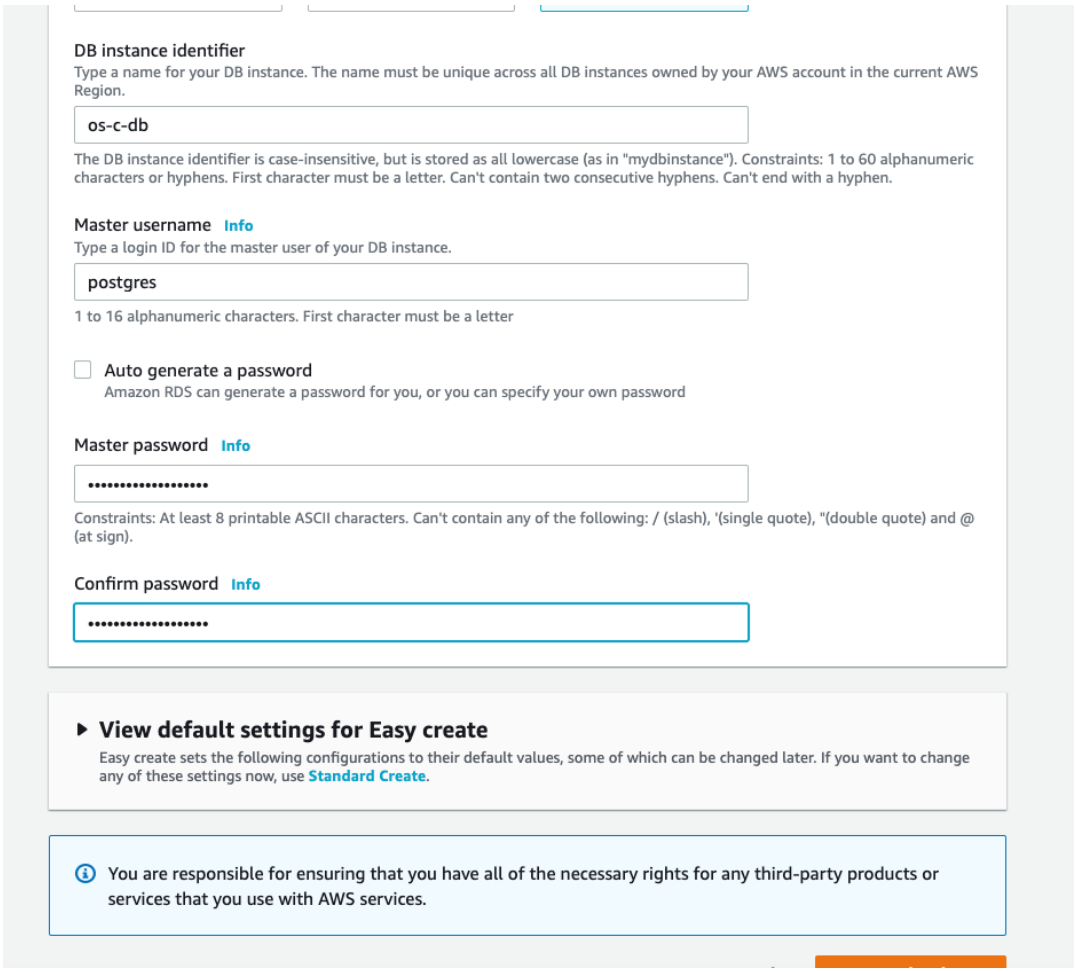
II. Creating and Loading the Database

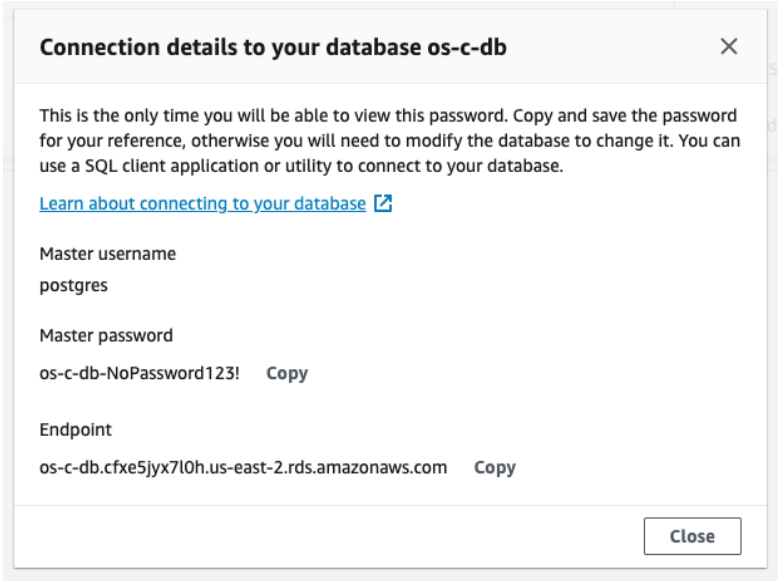
Step 1: Create the PostgreSQL database in the AWS RDS Service.

1. Select US-EAST-2 as the region.

2. Setting up the RDS. In AWS, select the RDS Service, and create a PostgreSQL t2.micro.instance. Set the credentials as you will.
   DB Instance identifier: os-c-db
   Master username: postgres
   Master password: os-c-db-NoPassword123!



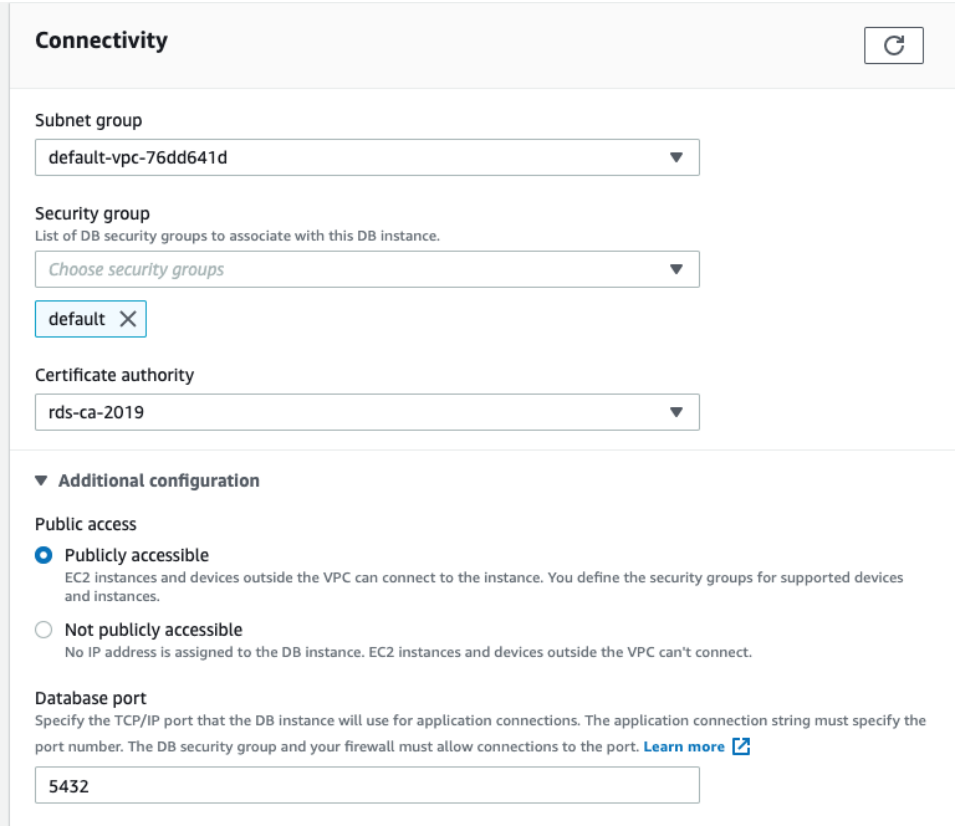

DB instance identifier
Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.
os-c-db
The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.
Master username Info
Type a login ID for the master user of your DB instance.
postgres
1 to 16 alphanumeric characters. First character must be a letter
Auto generate a password
Amazon RDS can generate a password for you, or you can specify your own password
Master password Info
Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).
Confirm password Info
View default settings for Easy create
Easy create sets the following configurations to their default values, some of which can be changed later. If you want to change any of these settings now, use Standard Create.
You are responsible for ensuring that you have all of the necessary rights for any third-party products or services that you use with AWS services.

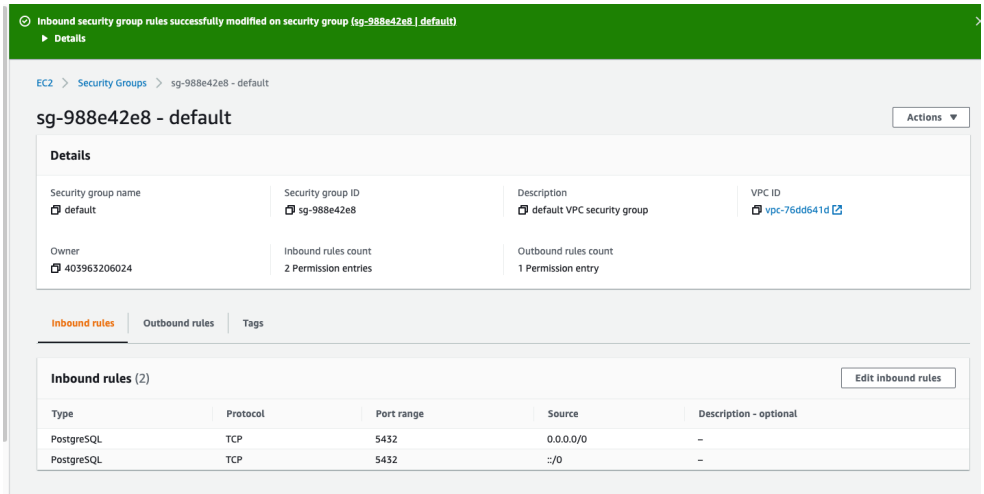

Connection details to your database os-c-db
This is the only time you will be able to view this password. Copy and save the password for your reference, otherwise you will need to modify the database to change it. You can use a SQL client application or utility to connect to your database.
Learn about connecting to your database
Master username
postgres
Master password
os-c-db-NoPassword123! Copy
Endpoint
os-c-db.cfxe5jyx7l0h.us-east-2.rds.amazonaws.com Copy
Close

Step 2: Tag the newly created database, and temporarily allow remote access to the database.

1. Tag the DB with “project”:”os-c”.

2. Temporarily allow your Desktop to access the database, by setting Public Accessibility to Yes, and the Security Group to allow Access on all Ports from your PC.





Step 3: Administer the database from your desktop via PgAdmin.

1. Connect to the AWS RDS instance from your PgAdmin using the credentials when you created your AWS RDS instance. Failure modes here include wrong or missing credential, database not accessible from your IP or any external IP.

2. Create a database called os-c-db

3. Create the tables and indices. Go to `OS-C Distribution/OS-C-RDS/ and modify the Python scripts so they point to the right RDS.`

```
import psycopg2
connection = psycopg2.connect(
    host="os-c-db.cfxe5jyx7l0h.us-east-2.rds.amazonaws.com",
port=5432, database="os-c-db", user="postgres", password="os-
c-db-NoPassword123!")
cursor = connection.cursor()
```

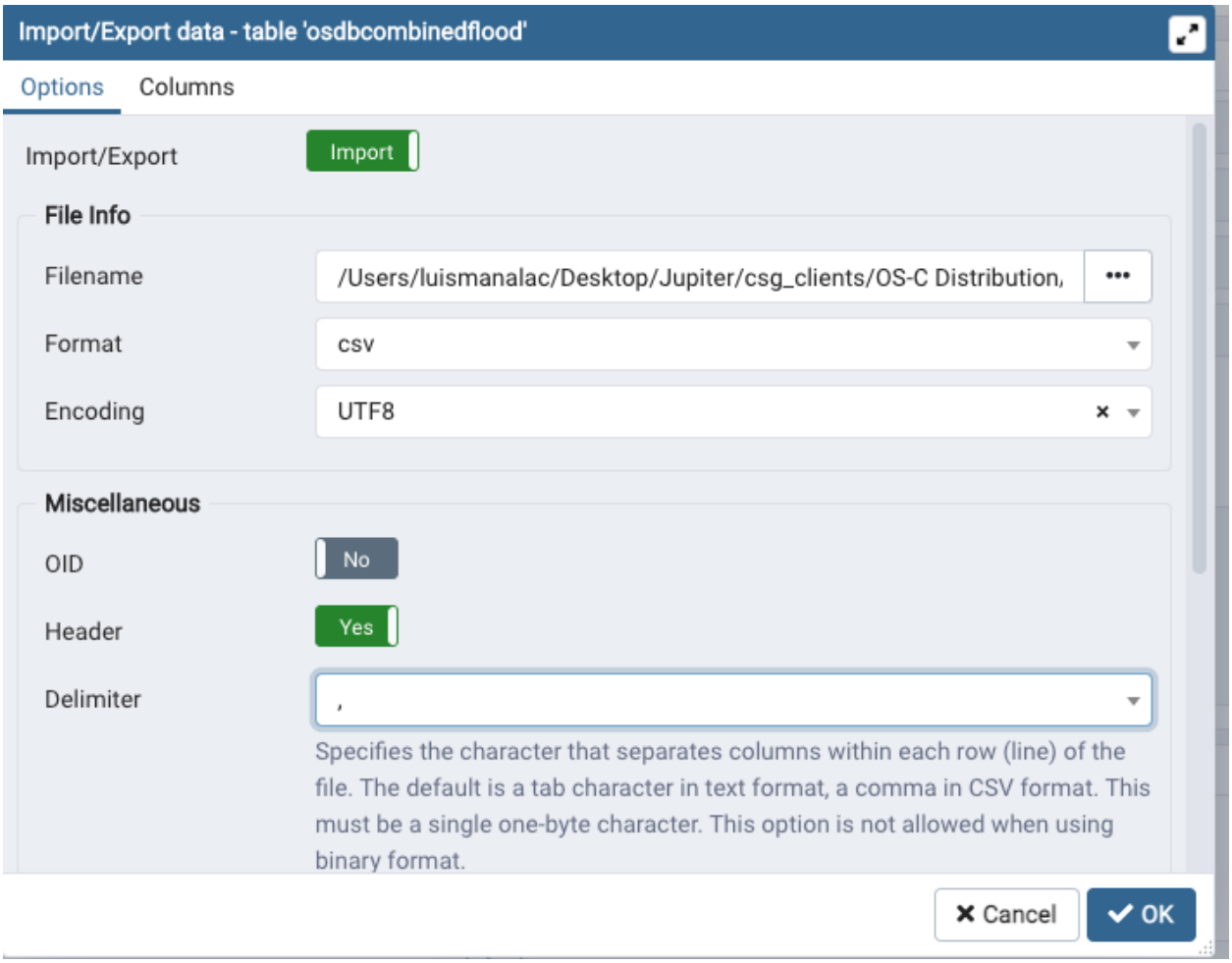
4. Using an environment with Python 3.7 and psychopg2 (e.g. psycopg2-binary), Run the Scripts (all 7 of them)

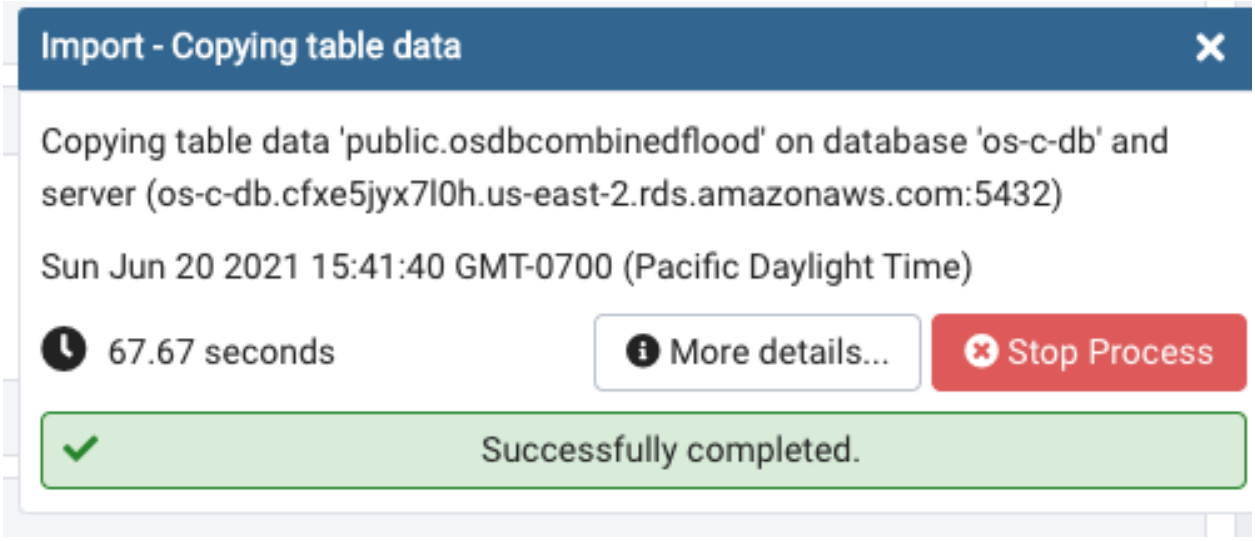
For example:

$ python OSCreateDB-CombinedFlood.py

The correct tables and indices will be created

Step 4: Use PgAdmin to import the CSVs from OS-C-DAT to the proper tables.





Step 5: Use etlcheck.sql to verify data was imported.

```
select * from osdbcombinedflood limit 10;
select * from osdbcombinedflood where key='64.00116.00';
select count(*) from osdbcombinedflood;
```

Repeat for all tables.

III. Installing, Configuring and Testing the API Application Server

Step 1: Test API-SERVER locally

1. Go to OS-C-API-SERVER. Activate an environment that contains Python 3.7.
2. Run

```
make install
```

3. In App.py change the connection string to point to the right DB

```
connection = psycopg2.connect(host="os-c-db.cfxe5jyx7l0h.us-
east-2.rds.amazonaws.com", port=5432, database="os-c-db",
user="postgres", password="os-c-db-NoPassword123!")
```

4. Run

```
make test
```

```
ython -m pytest -vv --cov=app test_app.py
========================================= test session starts ==========================================
latform darwin -- Python 3.8.5, pytest-6.2.4, py-1.9.0, pluggy-0.13.1 -- /opt/anaconda3/bin/python
achedir: .pytest_cache
ootdir: /Users/luismanalac/Desktop/Jupiter/csg_clients/OS-C Distribution/OS-C-API-SERVER
lugins: cov-2.12.1
ollected 1 item

est_app.py::test_index PASSED                                                                [100%]

---------- coverage: platform darwin, python 3.8.5-final-0 -----------
lame     Stmts   Miss  Cover
----------------------------
pp.py      322    260    19%
----------------------------
OTAL       322    260    19%


========================================== 1 passed in 1.53s ===========================================
```

Warning errors are acceptable.

5. Set environment variables

```
export db_host=os-c-db.cfxe5jyx7l0h.us-east-
2.rds.amazonaws.com
export db_port=5432
export db_database=os-c-db
export db_user=postgres
export db_password=os-c-db-NoPassword123!
export api_key=1234567890
```

6. Run

```
python app.py
```

to run os-api-server locally.py

```
(base) Luiss-MBP:OS-C-API-SERVER luismanalac$ python app.py
Connecting to : os-c-db.cfxe5jyx7l0h.us-east-2.rds.amazonaws.com
Connection successul
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
 * Restarting with fsevents reloader
Connecting to : os-c-db.cfxe5jyx7l0h.us-east-2.rds.amazonaws.com
Connection successul
 * Debugger is active!
 * Debugger PIN: 144-901-152
```

7. In a browser, type

   `localhost:8080/healthz`

   The service should return OK.

Step 2: Build Docker Image and Run Container Locally

1. In the OS-C-APP-SERVER directory, execute the Docker build and run commands:

```
docker system prune
docker build -t os-c-api-server .
docker run -p 8080:8080 -e api_key=1234567890 -e db_host=os-c-
db.cfxe5jyx7l0h.us-east-2.rds.amazonaws.com -e db_port=5432 -e
db_database=os-c-db -e db_user=postgres -e db_password=os-c-
db-NoPassword123! os-c-api-server
```

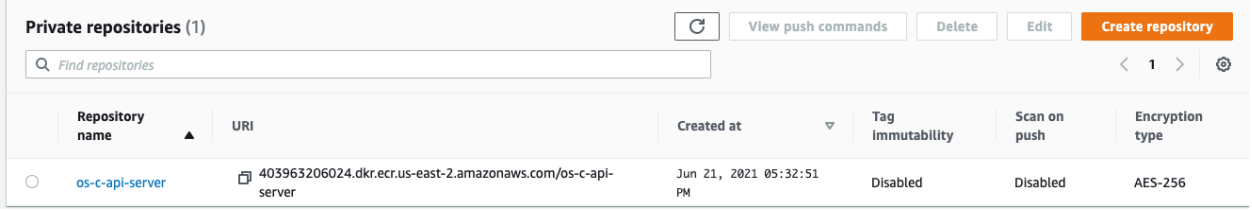
2. In a browser, type

```
localhost:8080/healthz
```

The service should return OK.

Step 3: Deploy and Run Image in AWS ECS Fargate with ALB

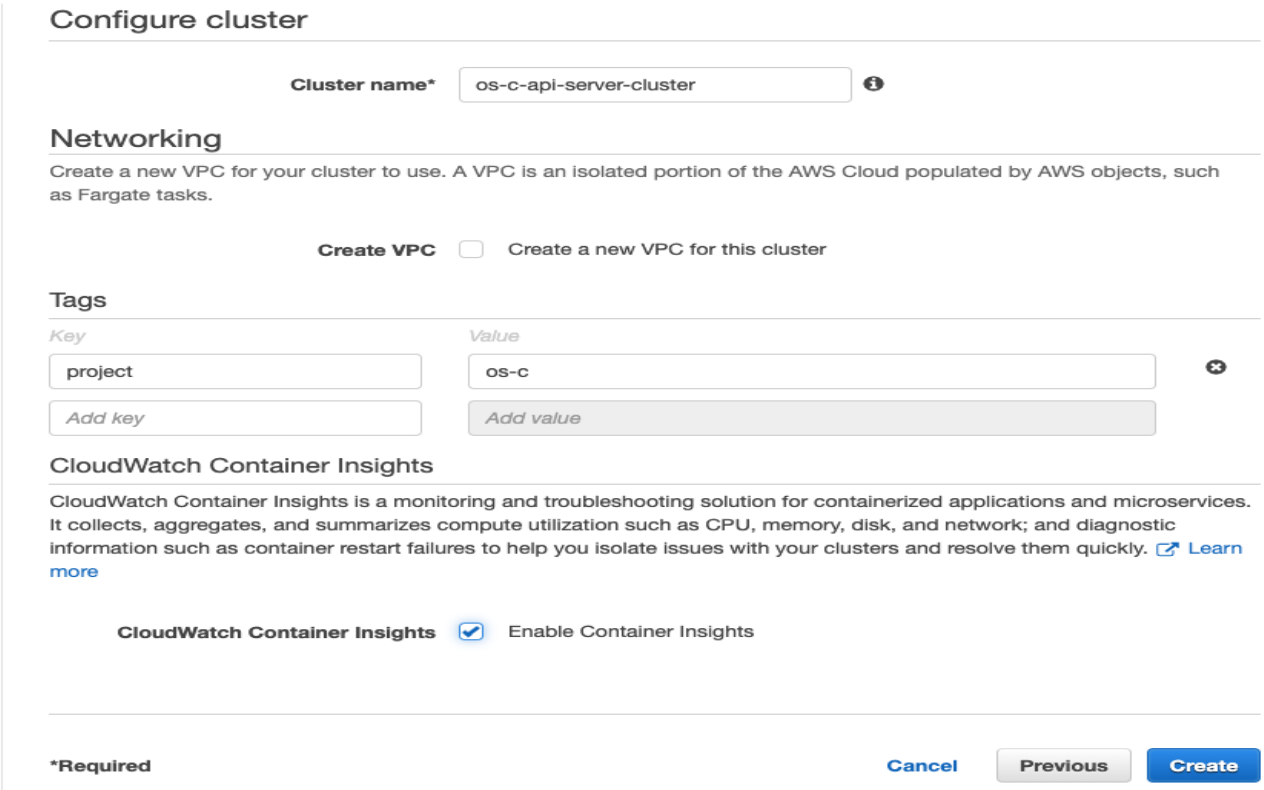
1. Create an Repository in US-EAST-02 ECR



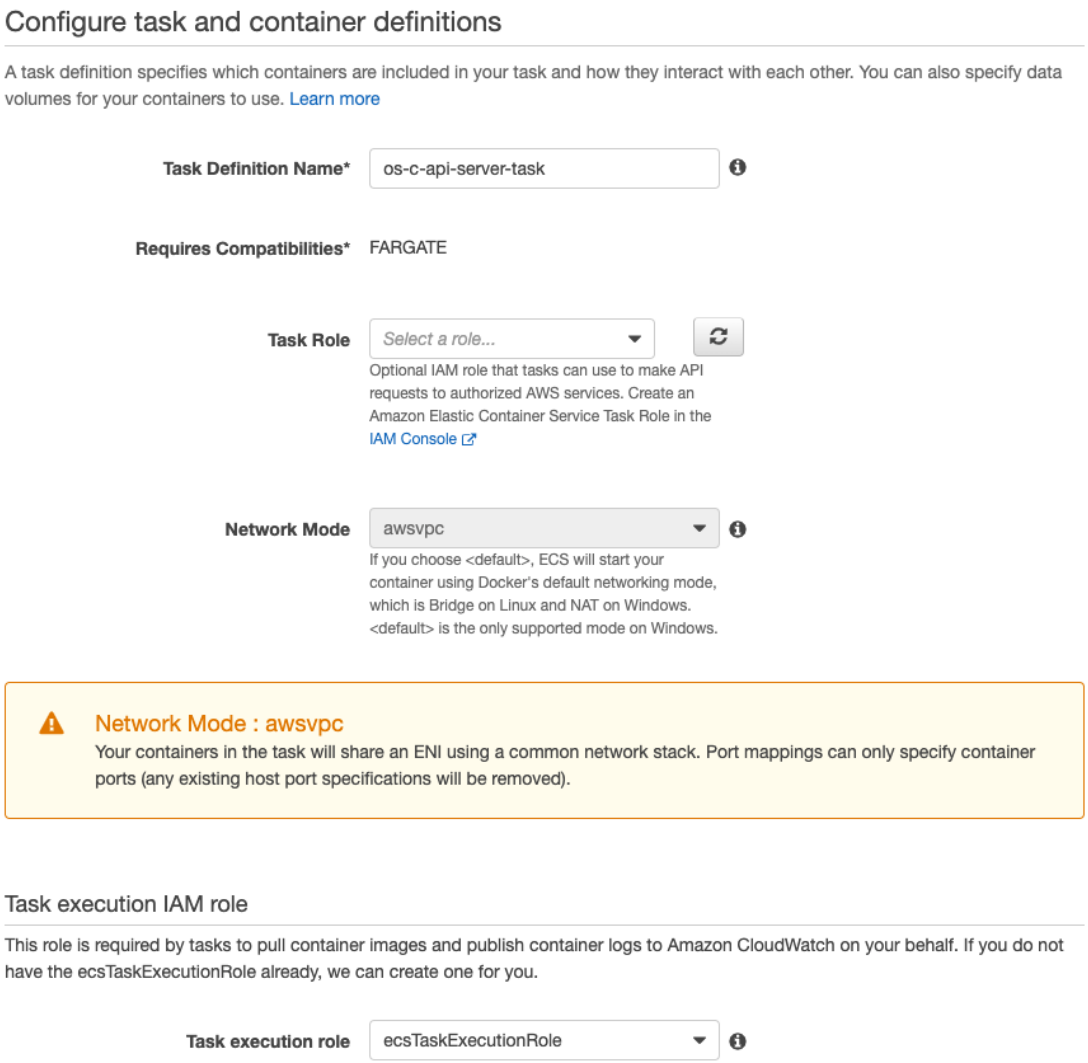
2. Push our locally built image into ECR

```
aws ecr get-login-password  --region us-east-2 | docker login
--username AWS --password-stdin 403963206024.dkr.ecr.us-east-
2.amazonaws.com

docker build -t os-c-api-server .

docker tag os-c-api-server:latest 403963206024.dkr.ecr.us-
east-2.amazonaws.com/os-c-api-server:latest
```

3. Create ECS Cluster. Make sure to expose port 8080 on the Container.



4. Create Task Definition



Task size

The task size allows you to specify a fixed size for your task. Task size is required for tasks using the Fargate launch type and is optional for the EC2 or External launch type. Container level memory settings are optional when task size is set. Task size is not supported for Windows containers.

Task memory (GB) 2GB

The valid memory range for 1 vCPU is: 2GB - 8GB.

Task CPU (vCPU) 1 vCPU

The valid CPU range for 2GB memory is: 0.25 vCPU - 1 vCPU.

Task memory maximum allocation for container memory reservation

0 2048 MiB

Task CPU maximum allocation for containers

0 1024 shared of 1024 CPU units

Container Definitions

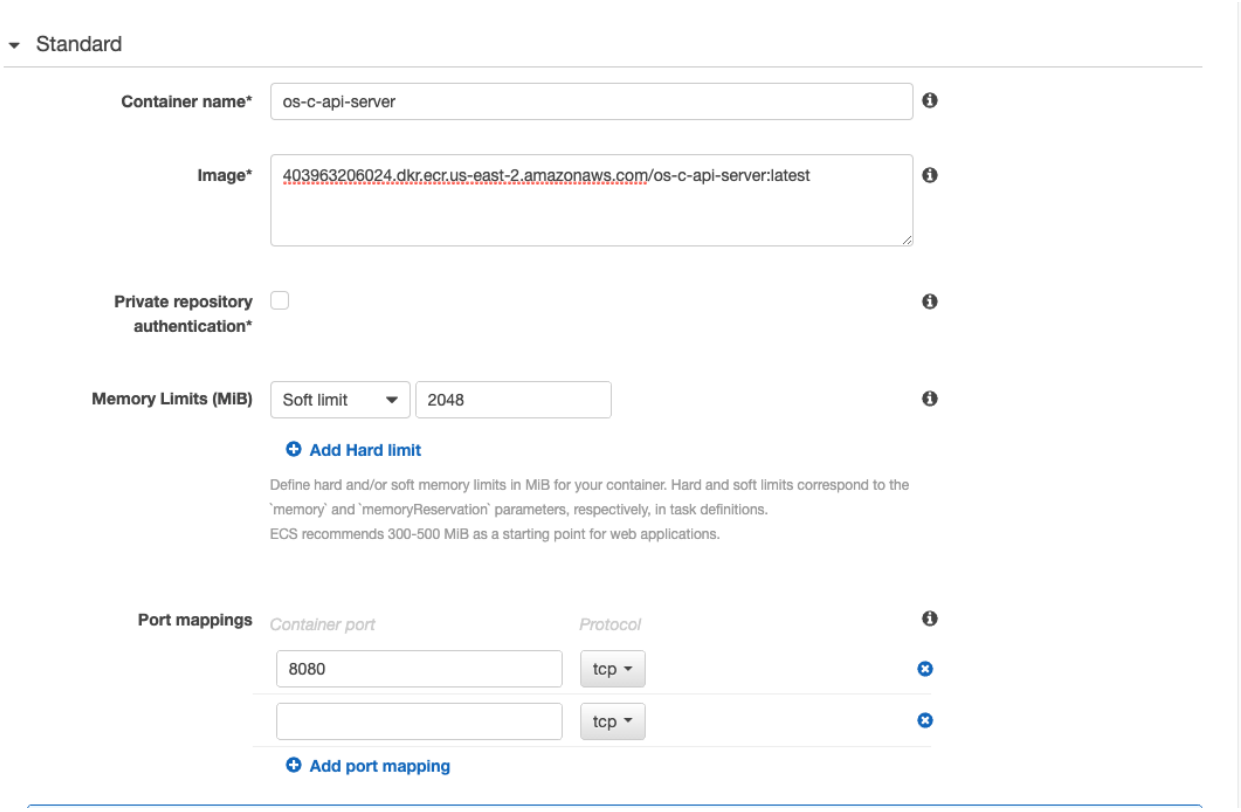
Add container

Container ...	Image	Hard/Soft ...	CPU Unit...	GPU	Inference A...	Essential ...	
os-c-ap...	4039632060...	--/2048				true	

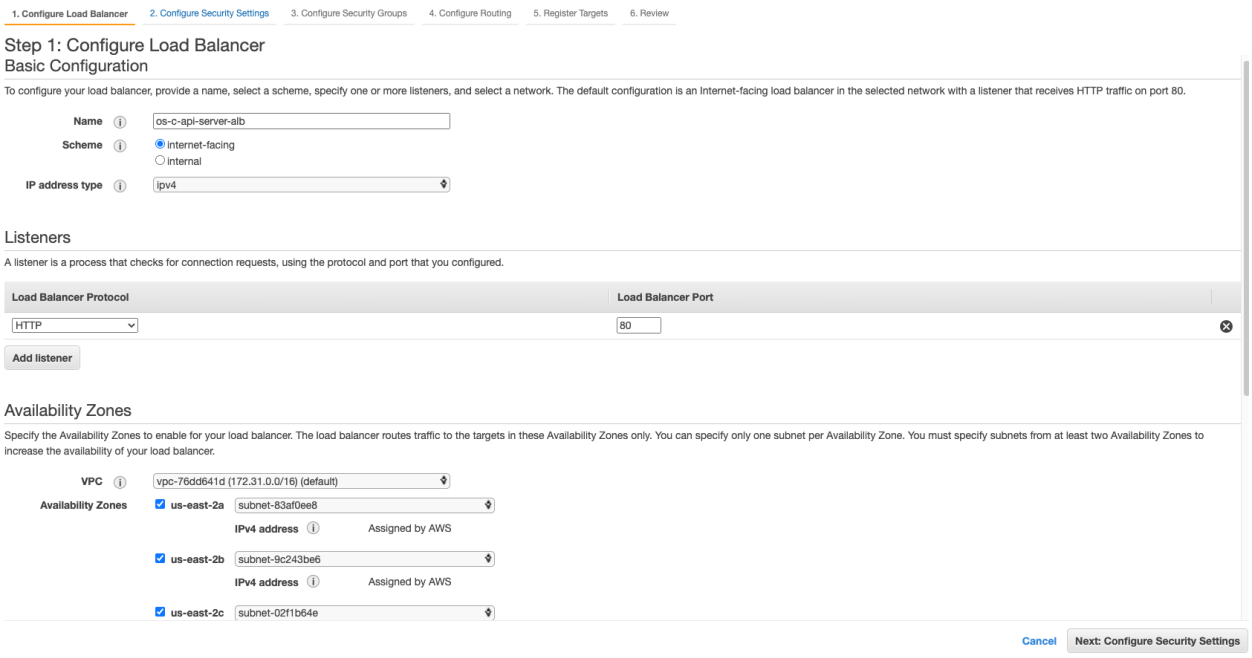
Service Integration

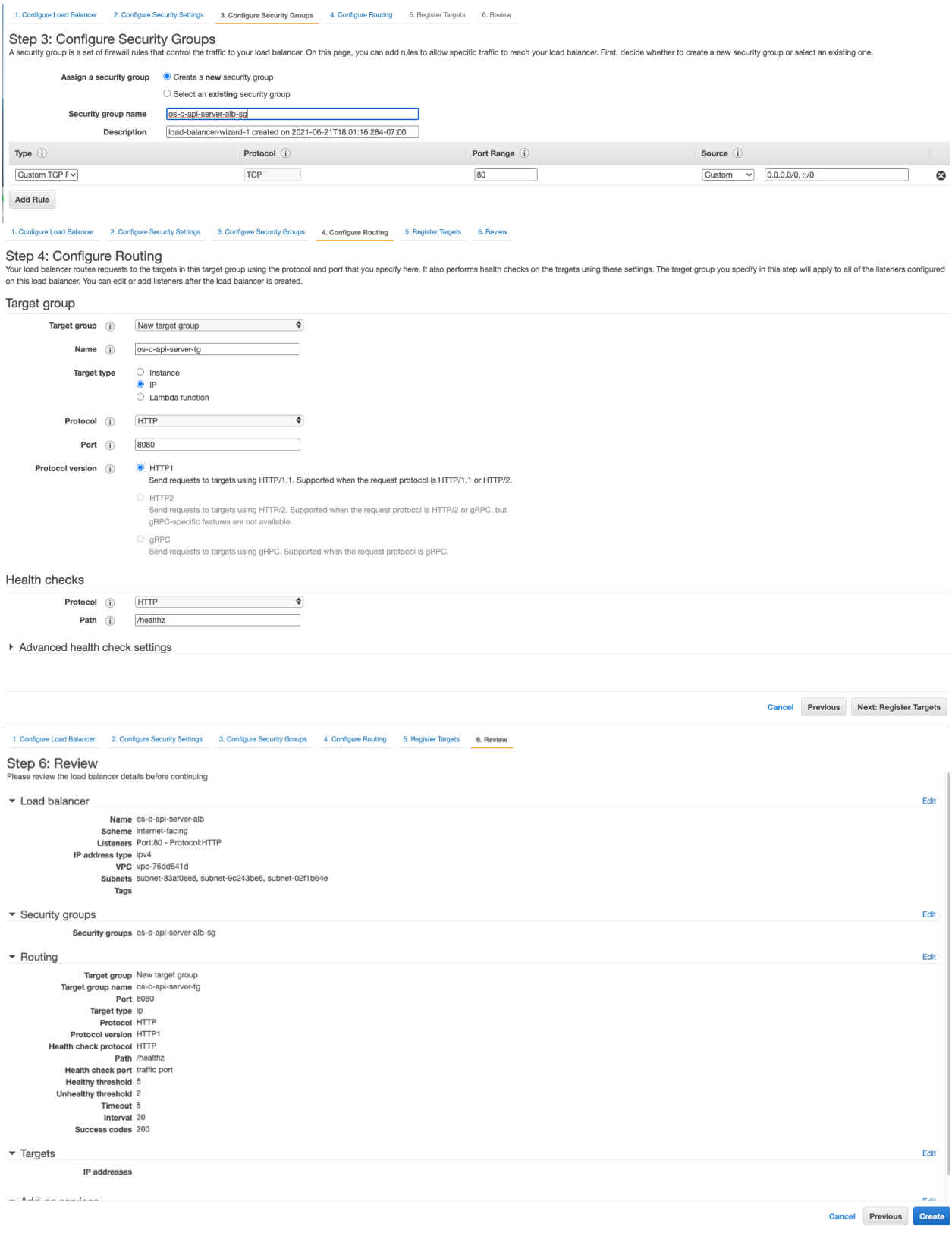
AWS App Mesh is a service mesh based on the Envoy proxy that makes it easy to monitor and control microservices. App Mesh standardizes how your microservices communicate, giving you end-to-end visibility and helping to ensure high-availability for your applications. To enable App Mesh integration, complete the following fields and then choose **Apply** which will auto-configure the proxy configuration. Learn more

Enable App Mesh integration ☐

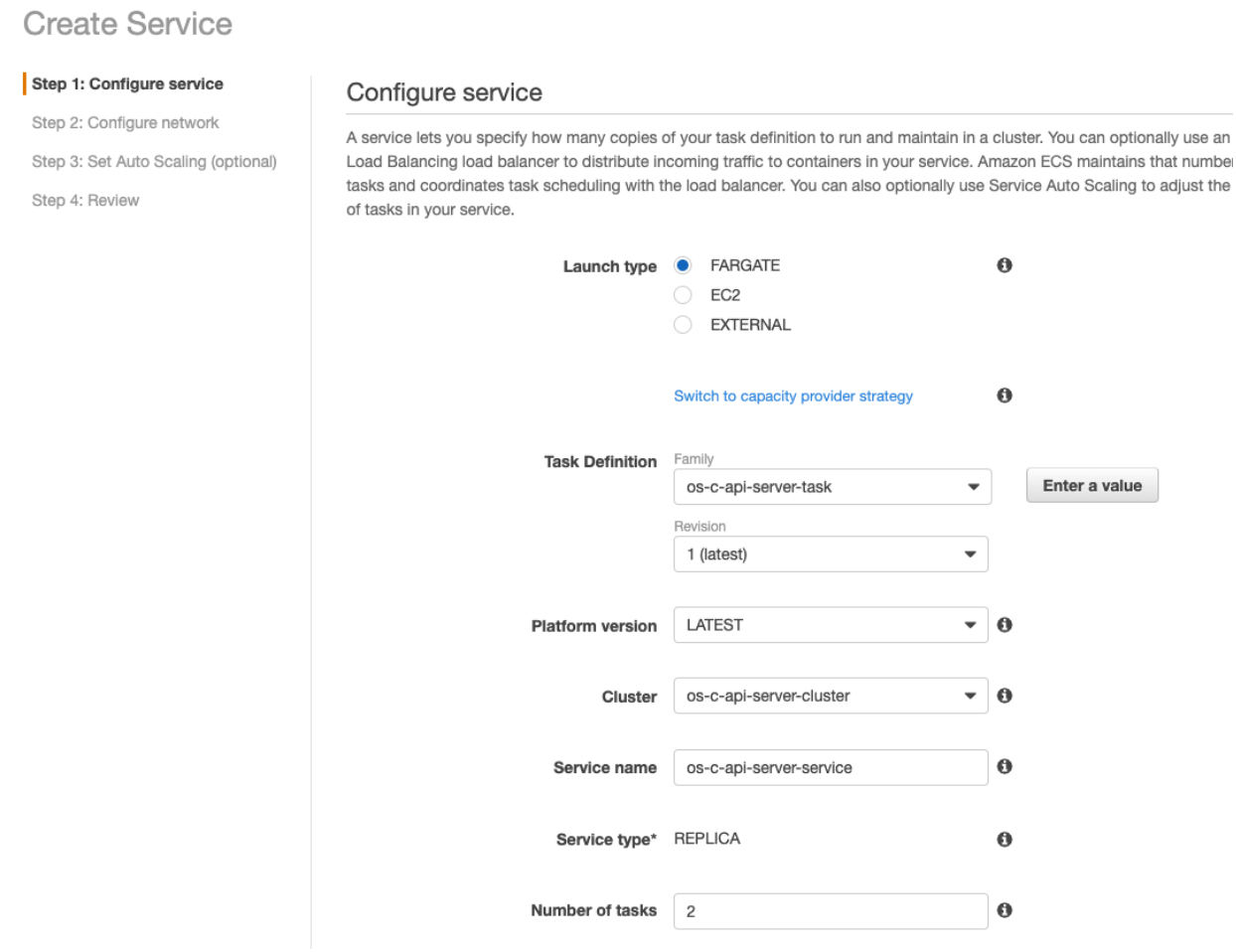


5. Create Application Load Balancer




1. Configure Load Balancer
2. Configure Security Settings
3. Configure Security Groups
4. Configure Routing
5. Register Targets
6. Review
Step 3: Configure Security Groups
A security group is a set of firewall rules that control the traffic to your load balancer. On this page, you can add rules to allow specific traffic to reach your load balancer. First, decide whether to create a new security group or select an existing one.
Assign a security group
Create a new security group
Select an existing security group
Security group name
os-c-api-server-alb-sg
Description
load-balancer-wizard-1 created on 2021-06-21T18:01:16.284-07:00
Type
Protocol
Port Range
Source
Custom TCP F
TCP
80
Custom
0.0.0.0/0, ::/0
Add Rule
1. Configure Load Balancer
2. Configure Security Settings
3. Configure Security Groups
4. Configure Routing
5. Register Targets
6. Review
Step 4: Configure Routing
Your load balancer routes requests to the targets in this target group using the protocol and port that you specify here. It also performs health checks on the targets using these settings. The target group you specify in this step will apply to all of the listeners configured on this load balancer. You can edit or add listeners after the load balancer is created.
Target group
Target group
New target group
Name
os-c-api-server-tg
Target type
Instance
IP
Lambda function
Protocol
HTTP
Port
8080
Protocol version
HTTP1
Send requests to targets using HTTP/1.1. Supported when the request protocol is HTTP/1.1 or HTTP/2.
HTTP2
Send requests to targets using HTTP/2. Supported when the request protocol is HTTP/2 or gRPC, but gRPC-specific features are not available.
gRPC
Send requests to targets using gRPC. Supported when the request protocol is gRPC.
Health checks
Protocol
HTTP
Path
/healthz
Advanced health check settings
Cancel
Previous
Next: Register Targets
1. Configure Load Balancer
2. Configure Security Settings
3. Configure Security Groups
4. Configure Routing
5. Register Targets
6. Review
Step 6: Review
Please review the load balancer details before continuing
Load balancer
Edit
Name os-c-api-server-alb
Scheme internet-facing
Listeners Port:80 - Protocol:HTTP
IP address type ipv4
VPC vpc-76dd641d
Subnets subnet-83af0ee8, subnet-9c243be6, subnet-02f1b64e
Tags
Security groups
Edit
Security groups os-c-api-server-alb-sg
Routing
Edit
Target group New target group
Target group name os-c-api-server-tg
Port 8080
Target type ip
Protocol HTTP
Protocol version HTTP1
Health check protocol HTTP
Path /healthz
Health check port traffic port
Healthy threshold 5
Unhealthy threshold 2
Timeout 5
Interval 30
Success codes 200
Targets
Edit
IP addresses
Cancel
Previous
Create

6: Create ECS Service



Create Service

Step 1: Configure service

Step 2: Configure network

Step 3: Set Auto Scaling (optional)

Step 4: Review

Configure network

VPC and security groups

VPC and security groups are configurable when your task definition uses the awsvpc network mode.

Cluster VPC* vpc-76dd641d (172.31.0.0/16)

Subnets* subnet-02f1b64e
(172.31.32.0/20) - us-east-2c
assign ipv6 on creation: Disabled

subnet-9c243be6
(172.31.16.0/20) - us-east-2b
assign ipv6 on creation: Disabled

subnet-83af0ee8
(172.31.0.0/20) - us-east-2a
assign ipv6 on creation: Disabled

Security groups* os-c-a-7146 Edit

Auto-assign public IP ENABLED

Health check grace period

If your service's tasks take a while to start and respond to ELB health checks, you can specify a health check grace period of up to 2,147,483,647 seconds during which the ECS service scheduler will ignore ELB health check status. This grace period can prevent the ECS service scheduler from marking tasks as unhealthy and stopping them before they have time to come up. This is only valid if your service is configured to use a load balancer.

Load balancing

An Elastic Load Balancing load balancer distributes incoming traffic across the tasks running in your service. Choose an existing load balancer, or create a new one in the Amazon EC2 console.

Load balancer type*

- ○ None
  Your service will not use a load balancer.
- ◉ Application Load Balancer
  Allows containers to use dynamic host port mapping (multiple tasks allowed per container instance). Multiple services can use the same listener port on a single load balancer with rule-based routing and paths.
- ○ Network Load Balancer
  A Network Load Balancer functions at the fourth layer of the Open Systems Interconnection (OSI) model. After the load balancer receives a request, it selects a target from the target group for the default rule using a flow hash routing algorithm.
- ○ Classic Load Balancer
  Requires static host port mappings (only one task allowed per container instance); rule-based routing and paths are not supported.

Service IAM role Task definitions that use the awsvpc network mode use the AWSServiceRoleForECS service-linked role, which is created for you automatically. Learn more.

Load balancer name os-c-api-server-alb

Container to load balance

os-c-api-server : 8080

Remove ✖

Production listener port* 80:HTTP

Production listener protocol* HTTP

Target group name os-c-api-server-tg

Health check grace period

If your service's tasks take a while to start and respond to ELB health checks, you can specify a health check grace period of up to 2,147,483,647 seconds during which the ECS service scheduler will ignore ELB health check status. This grace period can prevent the ECS service scheduler from marking tasks as unhealthy and stopping them before they have time to come up. This is only valid if your service is configured to use a load balancer.

Health check grace period 0

Load balancing

An Elastic Load Balancing load balancer distributes incoming traffic across the tasks running in your service. Choose an existing load balancer, or create a new one in the Amazon EC2 console.

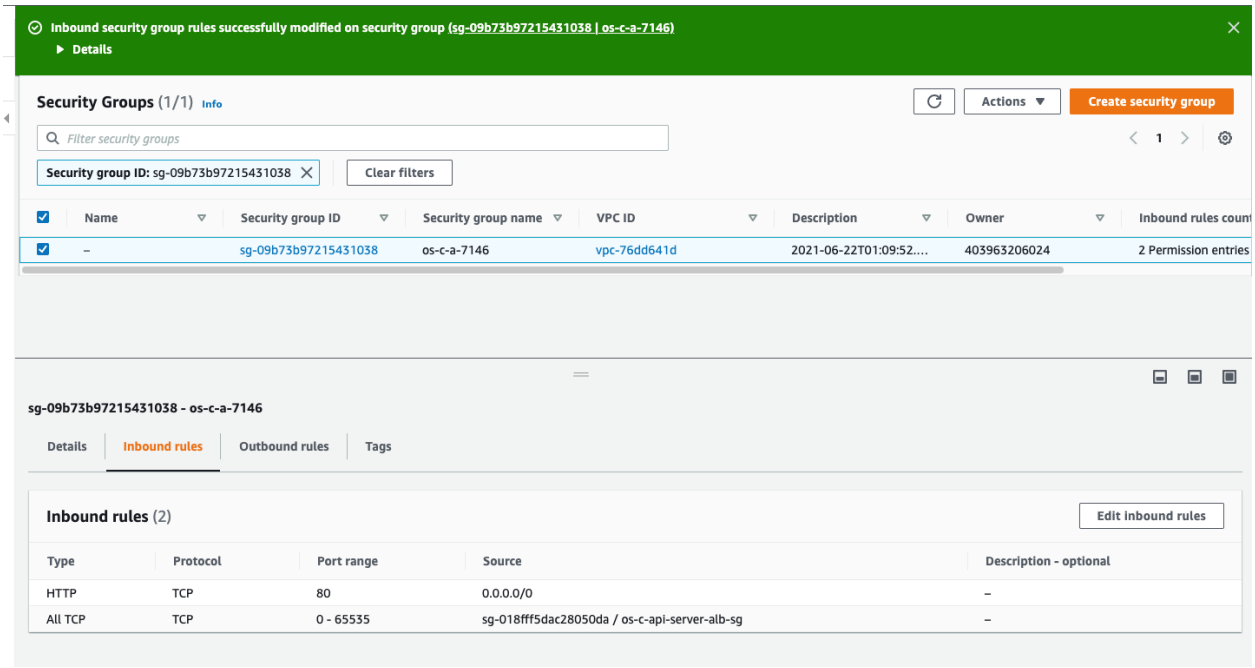
Load balancer type*

- None
  Your service will not use a load balancer.
- Application Load Balancer
  Allows containers to use dynamic host port mapping (multiple tasks allowed per container instance). Multiple services can use the same listener port on a single load balancer with rule-based routing and paths.
- Network Load Balancer
  A Network Load Balancer functions at the fourth layer of the Open Systems Interconnection (OSI) model. After the load balancer receives a request, it selects a target from the target group for the default rule using a flow hash routing algorithm.
- Classic Load Balancer
  Requires static host port mappings (only one task allowed per container instance); rule-based routing and paths are not supported.

Service IAM role Task definitions that use the awsvpc network mode use the AWSServiceRoleForECS service-linked role, which is created for you automatically. Learn more.

Load balancer name os-c-api-server-alb

8. Open ECS Security Group to ALB Security Group



9. In a browser, type the address of the ALB /healthz e.g.

   http://os-c-api-server-alb-1601253115.us-east-2.elb.amazonaws.com/healthz

   Should return OK.

IV. “**Installing, Testing, and Making Use of example Front End Applications**”

The third and final component of the ClimateScore Service, which can be found in OS-C-FE_APPS.

Step 1: Postman Collection and Environments

`csg_api_osc (RDS).postman_collection.json`

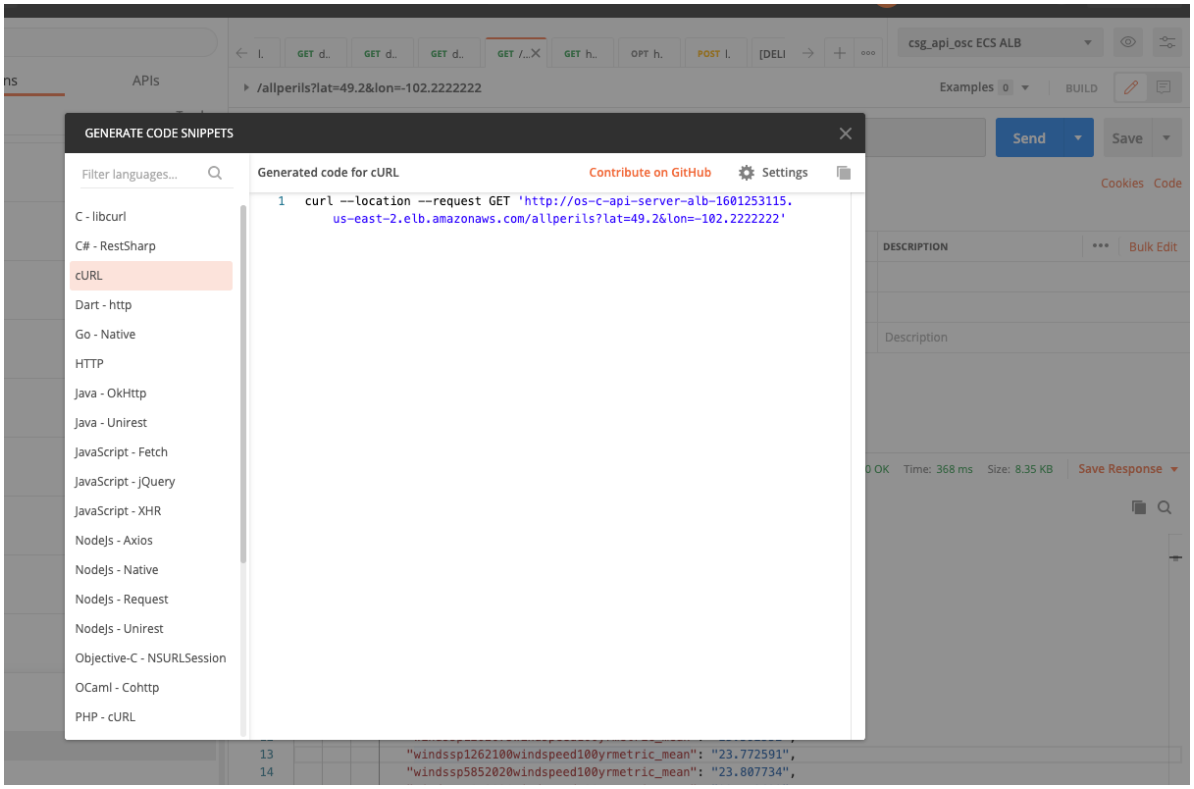
is a collection of postman tests to exercise the ClimateScore Service. It also provides good examples of how to access the service, by clicking on code (upper right) of a particular test and seeing a code snippet showing how that test is called.

`csg_api_osc ECS ALB.postman_environment.json`

`is a postman environment pointing to the deployed ALB for the ECS Cluster.`

`csg_api_osc (localhost:8080).postman_environment.json`

`is a postman environment pointing to the locally running version of the OS-C API Server. It is used for local testing.`



Step 2: Jupyter Notebook

Another include application is

```
os-c rds – test suite.ipynb
```

This application shows how filters can be passed to the ClimateScore Service, how to send multiple Locations (up to 100) in an array, and how to benchmark performance of the ClimateScore Service.

```
#url = "http://os-c-api-server-alb-1601253115.us-east-
2.elb.amazonaws.com//location"
#Num of Queries: 1000 Time Elapsed: 53.87759804725647
Queries Per Second: 18.56058986005449
#Num of Queries: 1000 Time Elapsed: 59.1816782951355
Queries Per Second: 16.89712135254191
#Num of Queries: 10000 Time Elapsed: 559.5767290592194
Queries Per Second: 17.870650226667507
```

Step 3: Open API/Swagger

The /location endpoint is also document in an Open API/Swagger compatible JSON file.

I. Security Best Practices

Step 1: Environment Variables

As part of Security Best Practices, critical parameters are not hard coded into the Application Source Code. Instead, they are injected at run-time through environment variables.

For local deployments, the following variables need to be set like so:

export db_host=os-c-db.cfxe5jyx7l0h.us-east-2.rds.amazonaws.com
```
export db_port=5432
export db_database=os-c-db
export db_user=postgres
export db_password=os-c-db-NoPassword123!
export api_key=1234567890
```

For containers, the variables are set in te JSON configuration of the Task Definition, like so:

```
"environment": [
        {
          "name": "api_key",
          "value": "1234567890"
        },
        {
          "name": "db_database",
          "value": "os-c-db"
        },
        {
          "name": "db_host",
          "value": "os-c-db.cfxe5jyx7l0h.us-east-
2.rds.amazonaws.com"
        },
        {
          "name": "db_password",
          "value": "os-c-db-NoPassword123!"
        },
        {
          "name": "db_port",
          "value": "5432"
        },
        {
          "name": "db_user",
          "value": "postgres"
        }
      ],
```

Step 2: API Key Authentication

A simple API Key Authentication system is implemented. Each call to the ClimateScore Service must include an X-Api-Key key in its header with a value corresponding to the one one set in the API Server environment variable, like so:

```
import requests

url = "https://api-phase3-osc.jupiterintel.com//location"

payload="{\n    \"locations\": [\n        {\"latitude\":
37.793871, \"longitude\": -122.395556}\n    ]\n}"
headers = {
  'X-Api-Key': '1234567890',
  'Content-Type': 'application/json'
}

response = requests.request("POST", url, headers=headers,
data=payload)
```

Step 3: Miscellaneous

- The API Server listens on port 8080.
- The ALB listens on port 443
- The API Server needs to access the AWS RDS at port 5432