AppRunner VPC connection with Copilot

Rafael Mosca, Associate Solutions Architect, WWPS EMEA Central SA

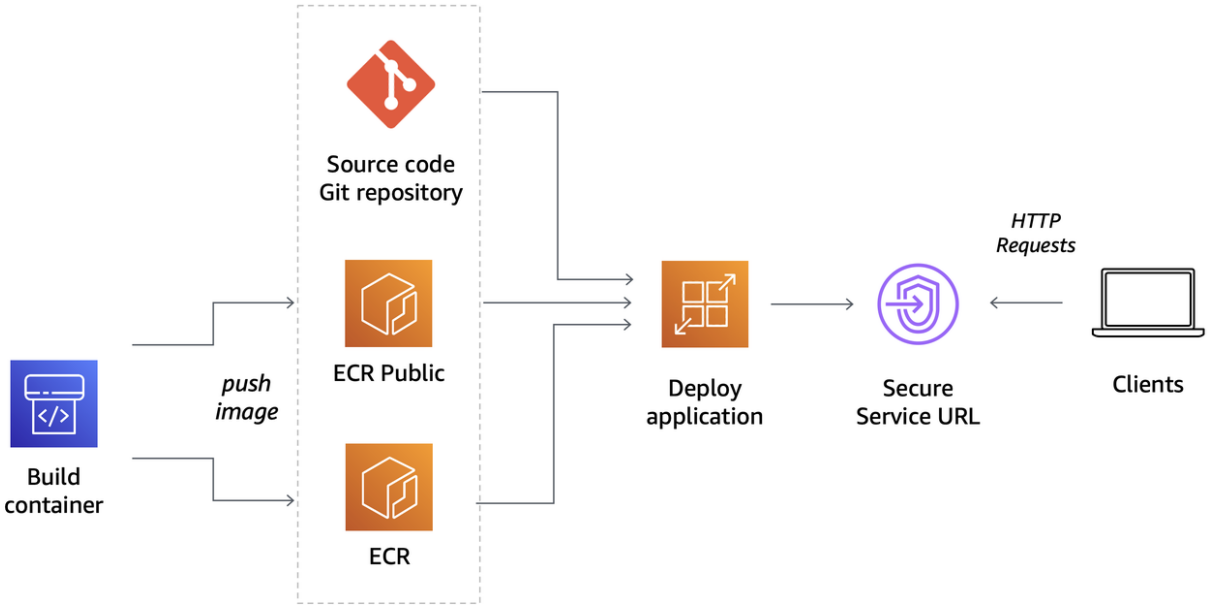
The AWS Copilot CLI is a tool that since its launch in 2020, developers have been using to build, manage, and operate Linux and Windows containers on Amazon Elastic Container Service (Amazon ECS), AWS Fargate, and AWS App Runner.

Not so long ago, we announced VPC Support for AWS App Runner. This means web applications and APIs that you deploy using AWS App Runner, can now communicate with databases running in services like Amazon Relational Database Service (RDS), and other applications running Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Service (EKS), Amazon Elastic Compute Cloud (Amazon EC2) that are hosted in an Amazon Virtual Private Cloud (VPC).

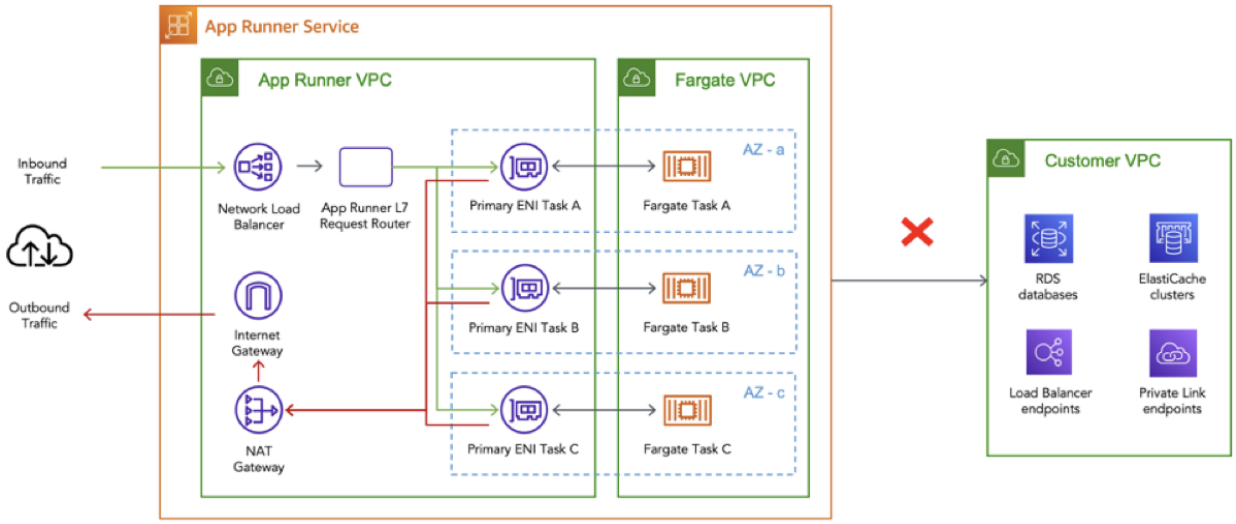
Although you can enable VPC access from the AWS App Runner console following the steps described in the documentation, during this blog we will see how it is actually possible to enable VPC access and connect to an Amazon Aurora database in a very easy way by using the AWS Copilot CLI.

Context

AWS App Runner is a fully managed service designed to make it easy for developers to run HTTP-based applications such as web and API servers. You don't need prior infrastructure or experience needed, you simply provide the source code or a container image, and App Runner will build and deploy your application containers in the AWS Cloud, automatically scaling and load-balancing requests across them behind the scenes. All you see is a service URL against which HTTPS requests can be made.



When you create a service, behind the scenes, App Runner deploys your application containers as AWS Fargate tasks orchestrated by Amazon Elastic Container Service (ECS) in an App Runner-owned VPC. By default, all outbound traffic initiated by the application is routed to the internet via a NAT Gateway and an internet gateway provisioned in the App Runner VPC. Therefore in this mode, applications hosted on App Runner can only connect to public endpoints on the internet and cannot reach private endpoints within your VPC.

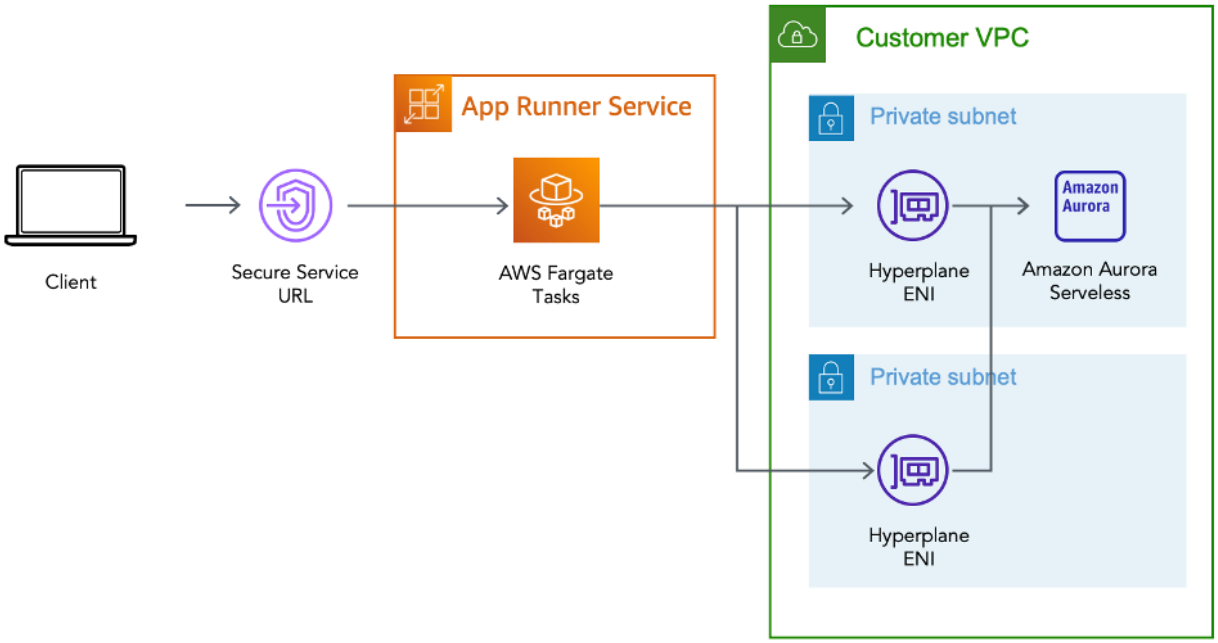


With the announcement of VPC support for App Runner services, applications can now connect to private resources in your VPC, such as an Amazon RDS database, Amazon ElastiCache cluster, or other private services hosted in your VPC.

To understand what happens under the hood, check out this dive deep post on VPC networking for the AWS App Runner. In simple words, by creating a VPC connector, the AppRunner service creates a uni-directional Elastic Network Interface (ENI) on the Customer-owned VPC that can be used for outbound traffic only (i.e. you cannot send any inbound requests to the ENI's IP address). This ENI has a security group that you can modify to associate your own custom outbound rules that allow communication to the desired destination endpoints.

Walkthrough

To demonstrate the capabilities of AWS Copilot CLI and VPC Support for AWS App Runner we are going to implement an architecture similar to the one you can find in the following diagram.



The steps we will follow will be the following:

- We will create a Customer VPC by using Copilot's notion of environment.
- We will create an Amazon Aurora PostgreSQL database inside a private network of this VPC.
- We will connect our service running on AWS App Runner to the database residing inside the VPC using the VPC connector and with the appropriate security group rules.
- We will verify the connection works by writing onto the database by using SQLAlchemy, a popular ORM for the Python programming language.

Prerequisites

For this walkthrough, you should have the following prerequisites:

- An AWS account.
- Have the AWS Copilot CLI installed.
- Properly configured AWS Credentials using the AWS CLI or with environment variables.
- Docker is installed and up and running.
- Clone the sample repository
    - ```
      git clone https://gitlab.aws.dev/rafams/apprunner-vpc-copilot.git
      ```

Create an application and environment

As a first thing, we are going to create a logical group of related services, environments, and pipelines we might create. In the AWS Copilot terminology this is called an *application*.

```
copilot app init apprunner-vpc
```

Once we execute that command, Copilot will use a folder named `./copilot` to hold special YAML configuration files
```

called *manifests* that will help us to easily deploy containerised applications on the AWS cloud.

Our next step is to create an environment for the application where we will deploy our services. With AWS Copilot it is possible to create different environments that logically isolate the deployments of our applications in a very easy way. A common use case is to have a test environment and a separate production environment where applications are deployed only when they have been validated on the test environment. For the scope of this walkthrough, we will only deploy the services to a testing environment named *test* that we create with the following command:

```
copilot env init \
    --app apprunner-vpc \
    --name test \
    --region 'eu-west-1' \
    --default-config
```

Once you press enter on the command, you will be asked to select AWS credentials that will be used to create the necessary infrastructure to host our services. Once the credentials have been selected, Copilot will start to create the resources on your behalf. This process may take a while so stretch a bit while this process is completed.
For every environment you create, AWS Copilot will create a separate networking stack (VPC).

Create a service running on AWS App Runner

AWS Copilot gives us several abstractions that we can use to deploy different types of services, in this case we are going to use the Copilot pattern called Request-Driven Web Service which deploys an AWS App Runner service that autoscales based on incoming traffic and scales down to a baseline when there's no traffic. This option is more cost effective for HTTP services with sudden bursts in request volumes or low request volumes.

```
copilot svc init \
    --app apprunner-vpc  \
    --svc-type "Request-Driven Web Service" \
    --name demo-service\
    --port 5000 \
    --dockerfile "demo-service/Dockerfile"
```

As usual, AWS Copilot generates a `manifest.yml` file that we can use to further customise the resources before we proceed to the actual deployment. To enable VPC access all that is needed is to uncomment the following section from the manifest file:

```
network:
  vpc:
    placement: private
```

Database on the VPC

To verify that we can connect to a private resource in our VPC, we are going to create an Amazon Aurora PostgreSQL database named `demoDb` using AWS Copilot.

```
copilot storage init \
    --name demoDb \
    --storage-type Aurora\
    --workload demo-service \
    --engine PostgreSQL \
    --initial-db demo
```

This will create a file under `./copilot/demo-service/addons/demoDb.yml` which contains configuration of the Amazon Aurora Serverless database that will be deployed using the AWS Copilot CLI.

If you explore the manifest file you will see that the database will be created by default inside of the private subnets associated to our Copilot application and environment VPC.

It is worth noticing that Copilot will use AWS Secrets Manager to generate the database password, so to be able to access the database password, AWS Copilot will inject an environment variable with the secret ARN so we can later use the SDK to retrieve the password (in our case, since we are using Python, we will use boto3 and the environment variable injected will be named `DEMO_DB_SECRET_ARN`), so in order to retrieve the password in Python we need to write something like:

```
sm_client = boto3.client('secretsmanager')

secret_arn = os.getenv("DEMO_DB_SECRET_ARN")
response = sm_client.get_secret_value(SecretId=secret_arn)
secret = json.loads(response['SecretString'])
```

`secret` will then be a dict made with the following information: `{dbClusterIdentifier, password, dbname, engine, port, host, username}` which we can use to connect to the database.

In this case, we are going to leverage SQLAlchemy, a popular Object Relation Mapper (ORM) tool for Python that lets you query and manipulate data from a database using an object-oriented paradigm instead of the low-level SQL queries.

To connect to the database we need to construct a database connection string as specified in the documentation. In our case we can get all the needed parameters with the secret dictionary.

```
DB_URI = f"postgresql://{secret['username']}:{secret['password']}@{secret['host']}/{se
```

Deploy sample application

Now that we have everything ready let's deploy our sample application which allows to insert new users into a private database:

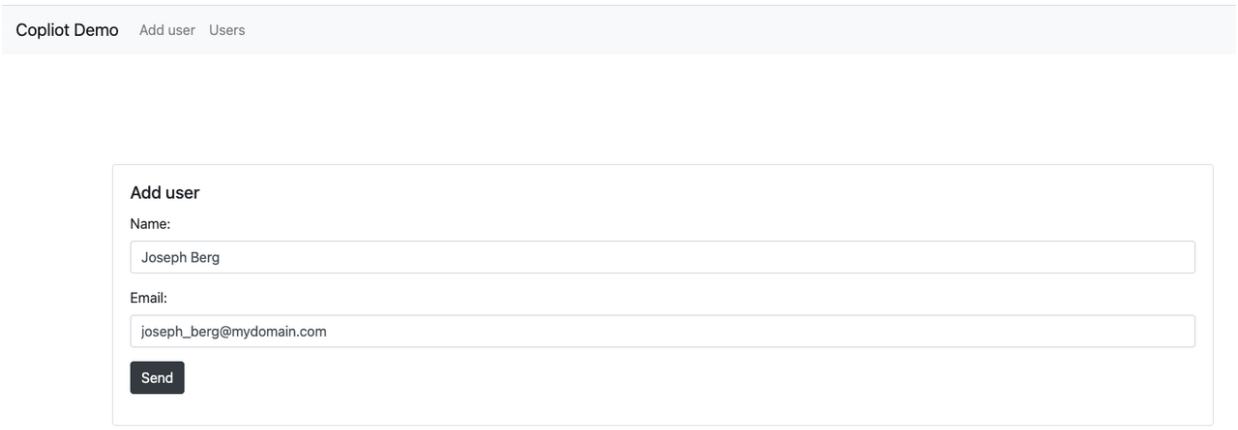
```
copilot svc deploy  --name demo-service --env test
```

Copilot will build the container image using the Docker Daemon, so if the daemon is not running an error will pop-up. Copilot will then start to create all the needed resources and when the provisioning is done, you will get a secure URL through which you can access the application. It should look something similar to this: https://rxrwprcxdp.eu-west-1.awsapprunner.com/

In case you want to retrieve this URL in future occasions you can run `copilot svc show  --name demo-service` and copy the value of the URL under the *Routes* section.

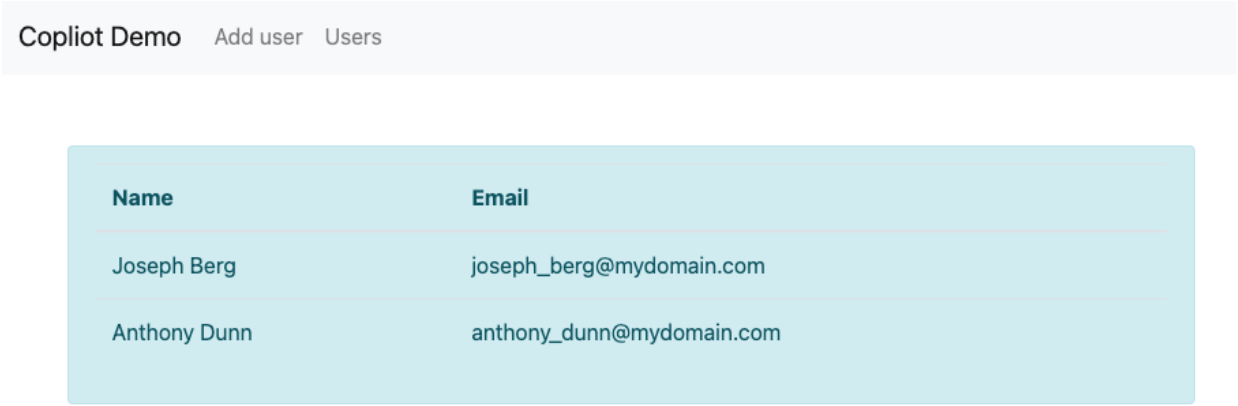
Verify it works

Every time you refresh the page a new name and email will be generated, but you are able to modify the fields if you want.

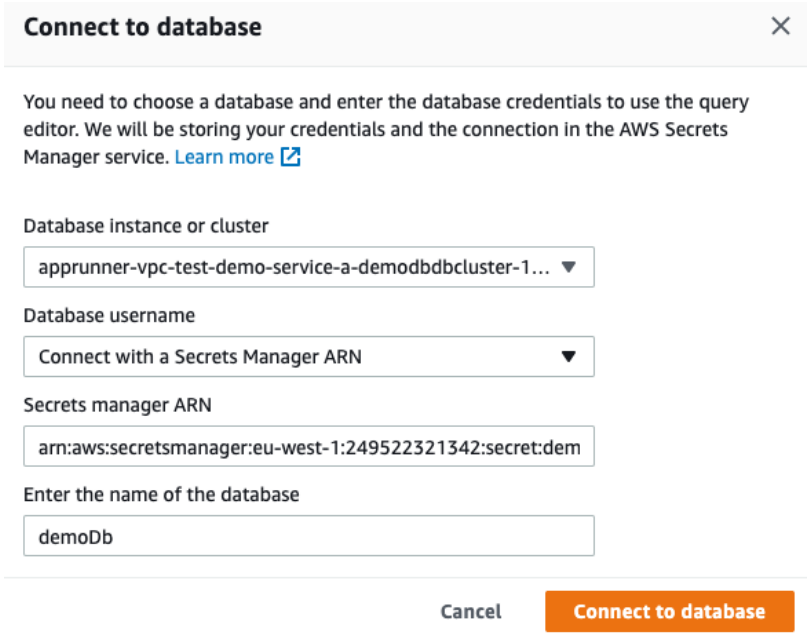


By clicking send, SQLAlchemy will perform a SQL Insert the user into the database we have created (`demo`) and into the table '`users`' that we have specified into the Python code with (`__tablename__ = 'users'`).

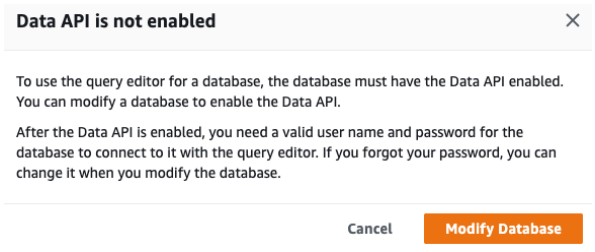
Add several users by using the front-end. To verify the users have been saved on the database you can go to the Users section of the front-end and see they are there.

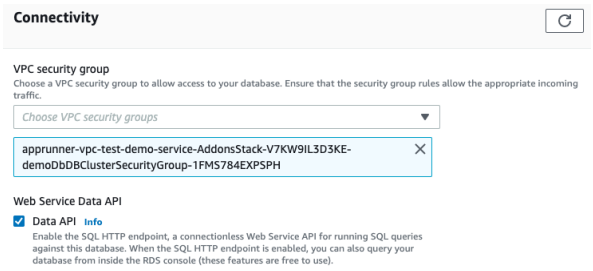


To further verify the users are really in the database, we can use the query editor in the RDS console which allows you to run Data Definition Language (DDL) and Data Manipulation Language (DML) using standard SQL statements.

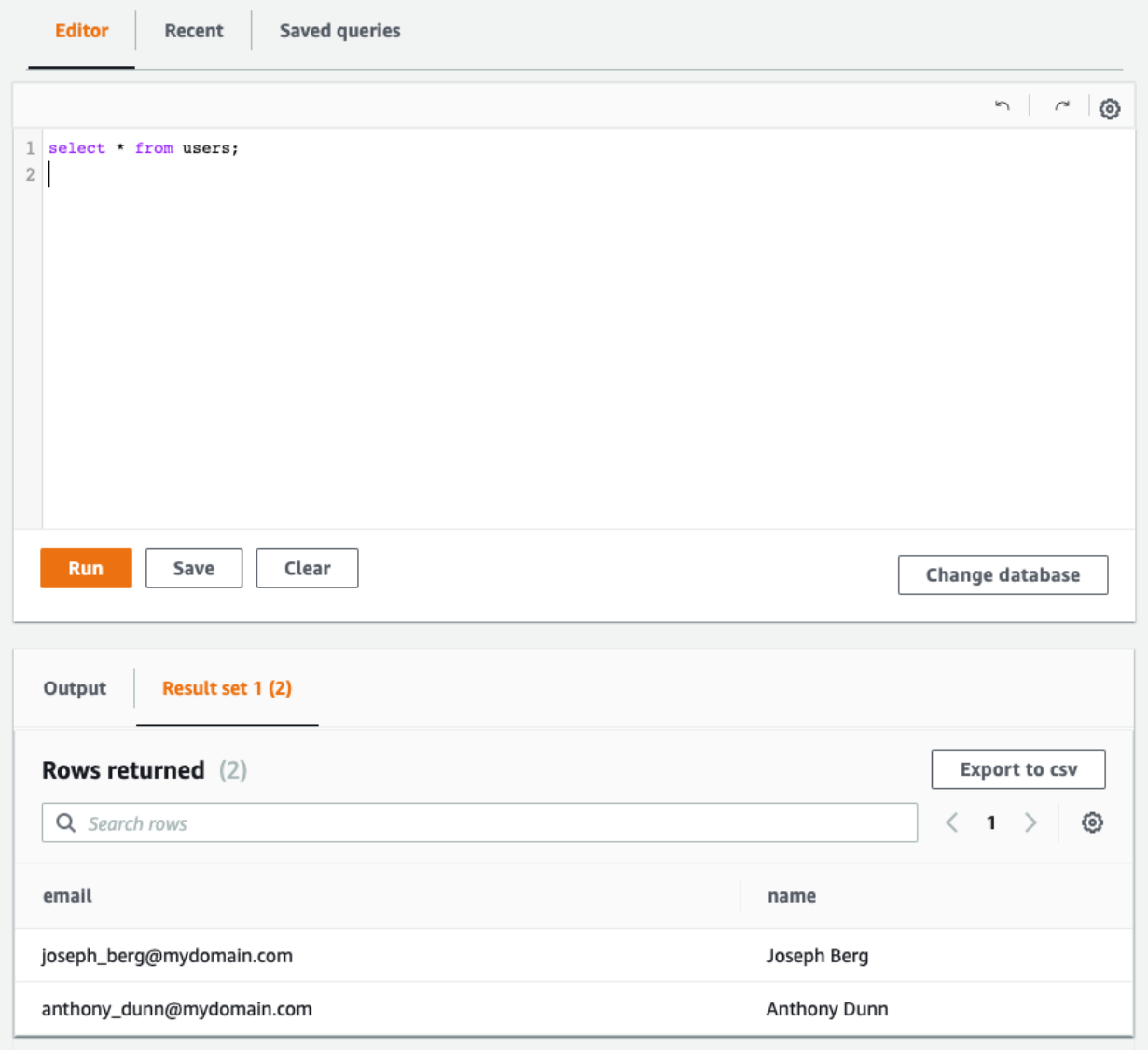


By default this functionality is not activated and requires you to activate it manually by enabling the Data API.





After enabling this functionality we can run SQL statements like a SELECT to verify the inserted data is indeed inside the database.



Cleaning up

To avoid incurring future charges, delete the resources. If you created everything correctly, you should be able to run the command

`copilot app delete apprunner-vpc`

and all the services and related infrastructure created for this demo will be deleted.

Conclusion

We saw how we can connect to private VPC resources by using AWS Copilot and the new VPC connection feature.