David Ezeji
12/15/2019

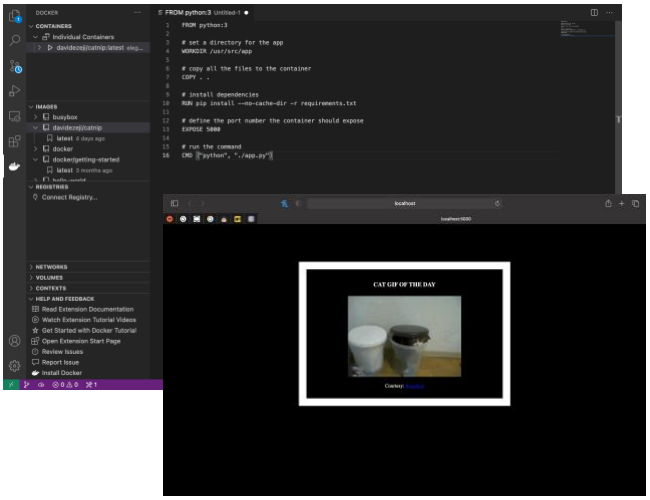
AWS Projects

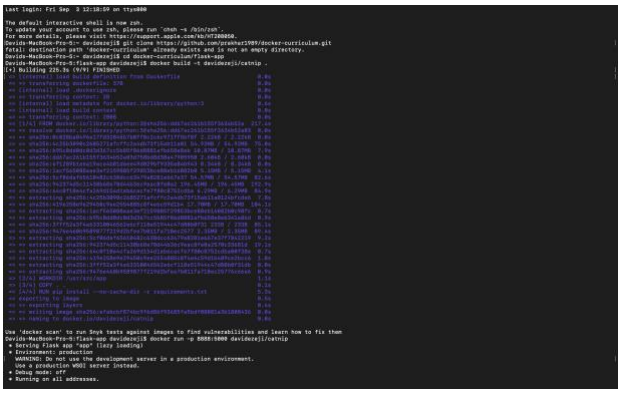
Project #1

Overview: Set up a Python-based AWS Fargate application (cat gif) within ECS. It was then configured to use an Application Load Balancer so that the application could scale.

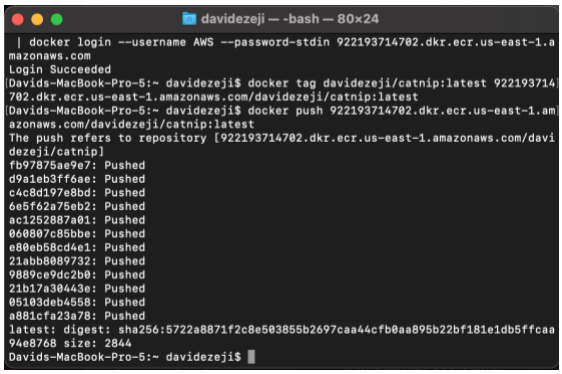
Steps:

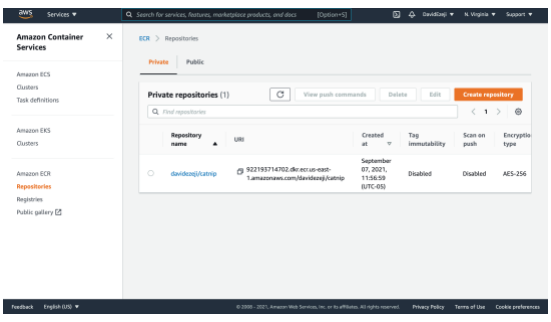
1. Created a docker image



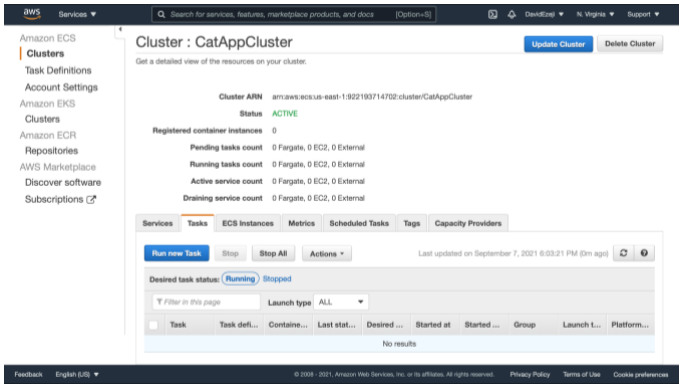


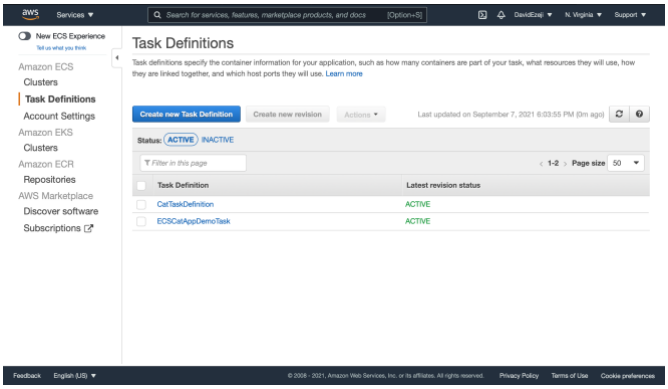
2. Pushed docker image to AWS using ECR as a repository



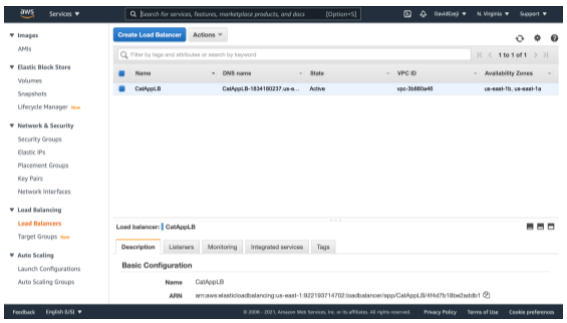


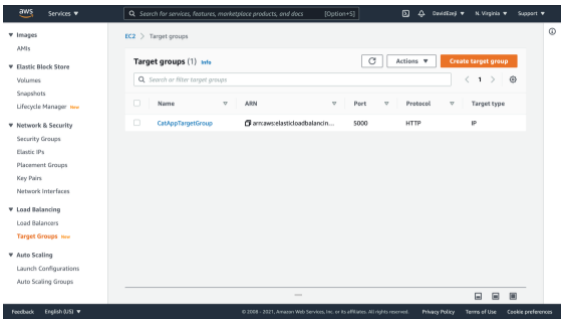
3. Created an ECS Cluster and task definition



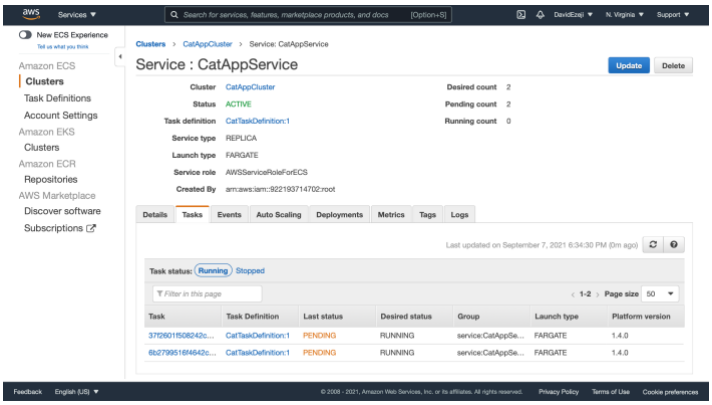


4. Configured an Application Load Balancer/Target Group

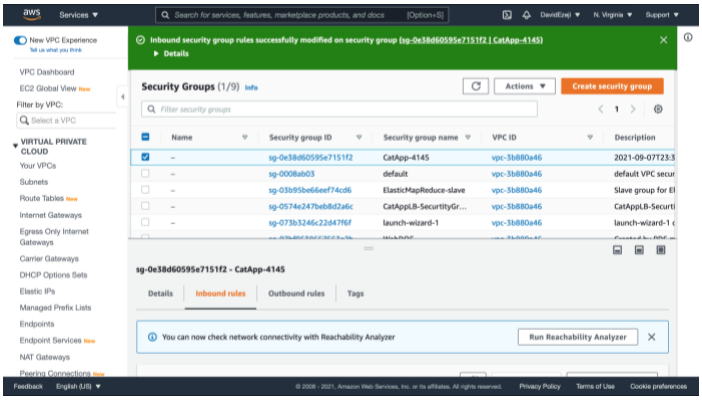




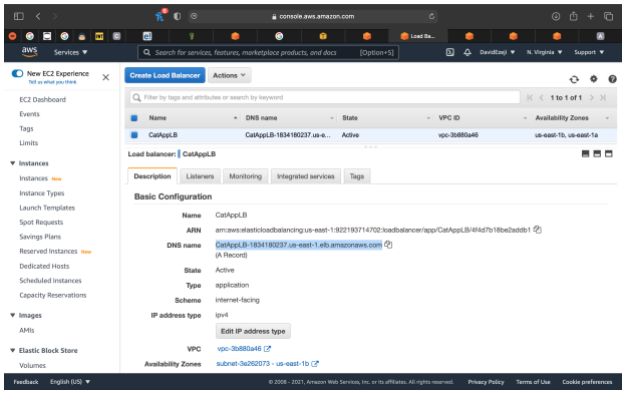
5. Created Fargate service

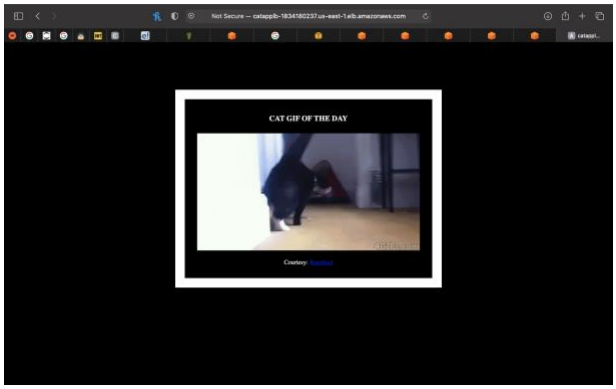


6. Modified Inbound Security Group rules for access to application



7. Docker image is successfully hosted via Fargate behind an Application Load Balancer



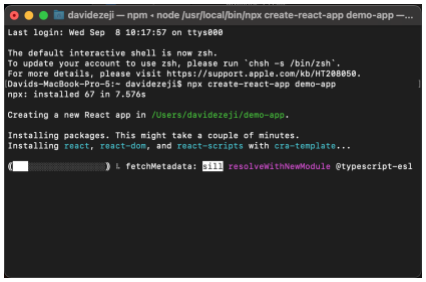


Project #2

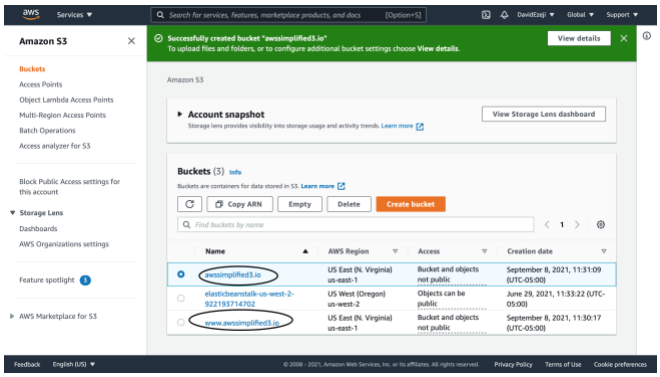
Overview: Deployed a React application to AWS using S3 with static hosting and distributed it via CloudFront.

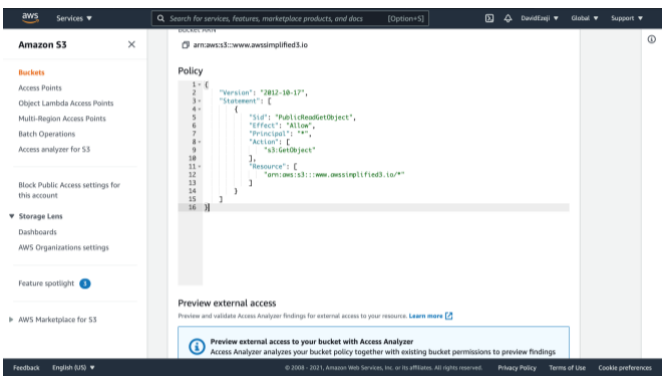
Steps:

1. Created React application

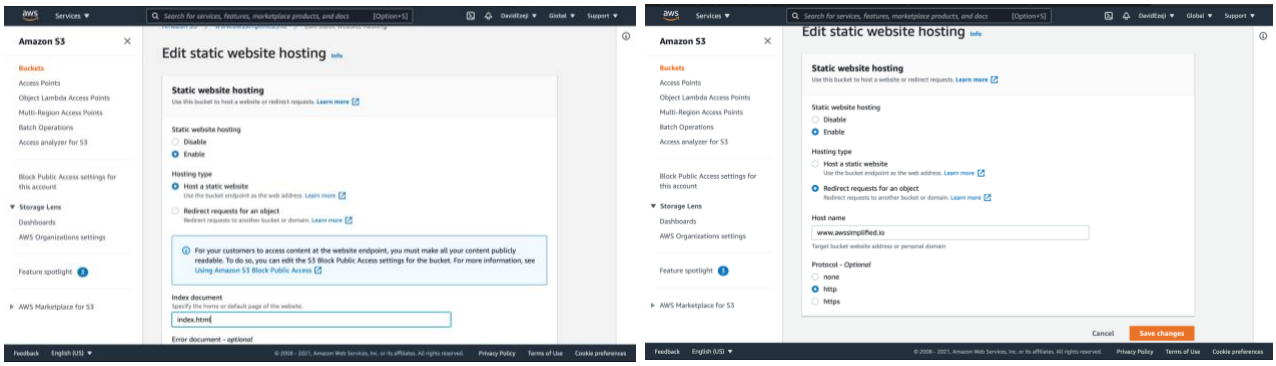


2. Created two s3 buckets

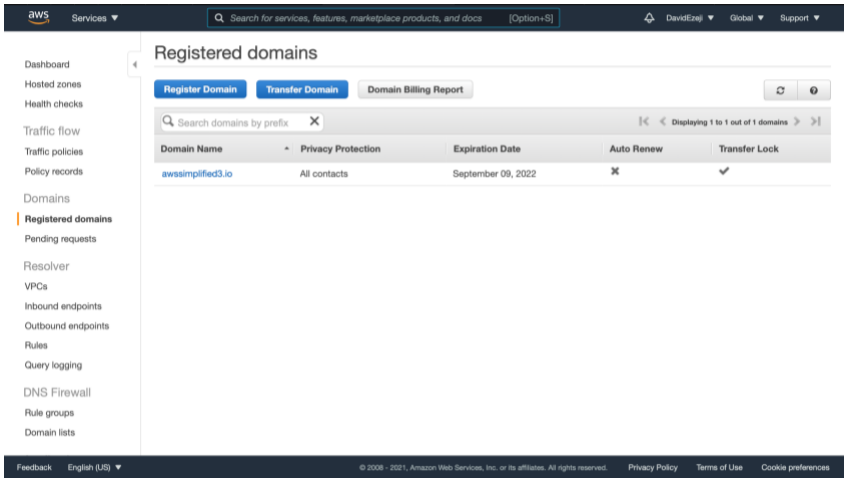




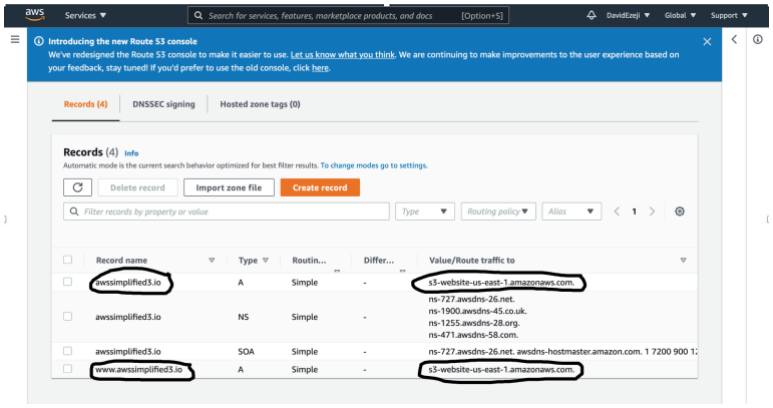
3. Configured one s3 bucket to host a static website (www.awssimplified3.io bucket), while using the other one (awssimplified3.io) to redirect requests to the other bucket



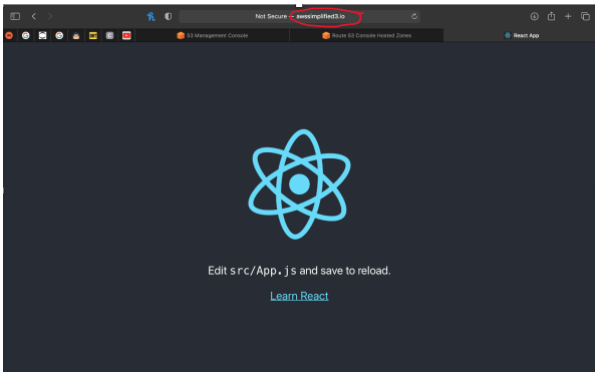
4. Registered a domain using Route53



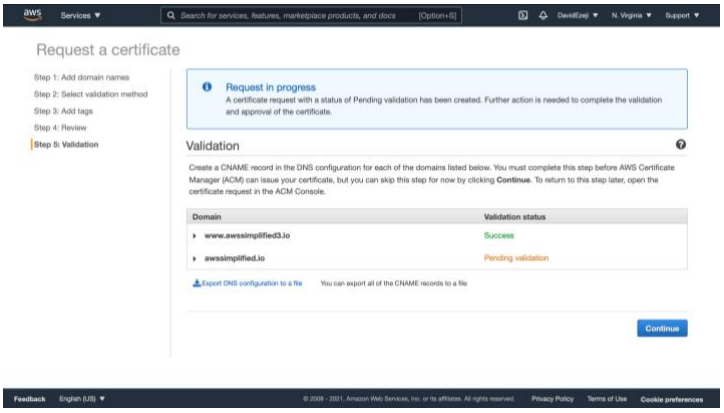
5. Created two records so that when users search the domain name, they are directed to the contents of the s3 bucket



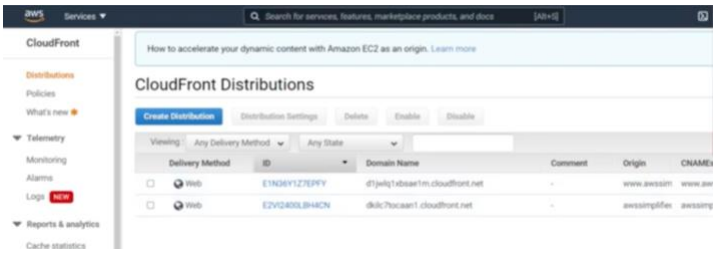
6. React app now shows up when domain name is entered, as domain directs traffic to the contents of the s3 buckets



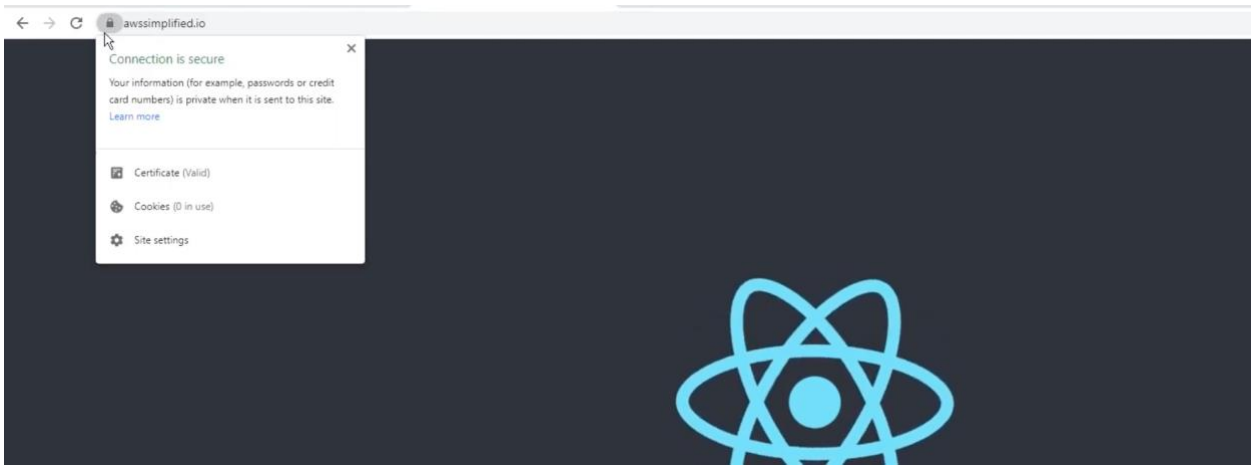
7. Set up HTTPS certifications for the application (using AWS Certificates Manager) so that it could be distributed by CloudFront



8. Set up CloudFront distributions



9. The React application is now distributed with CloudFront and is providing a secure connection for users as HTTPS has been enabled

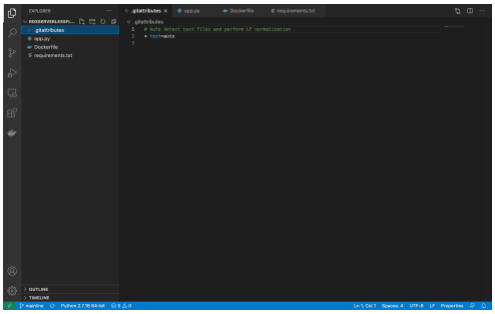
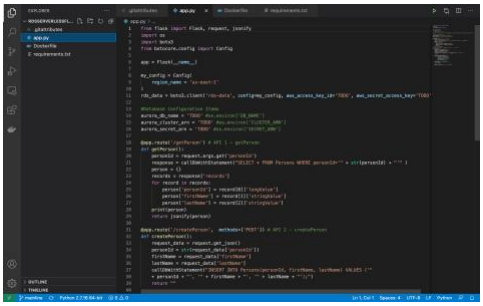
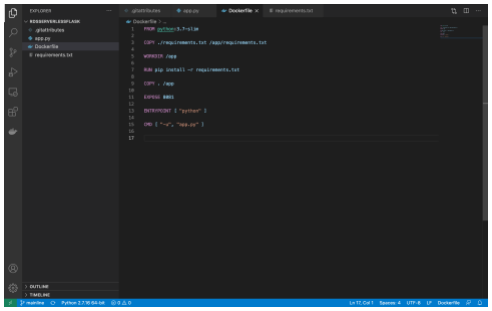
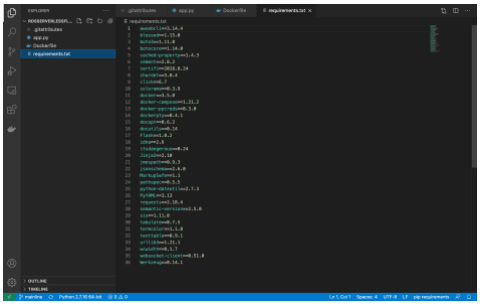


Project #3

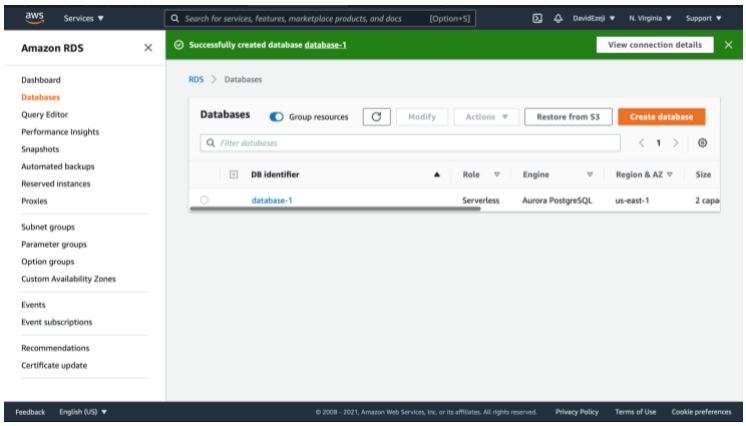
Overview: Built a Docker based flask application (sourced from GitHub) that uses an Application Programming Interface (API) to interact with an Aurora Serverless instance on AWS.

Steps:

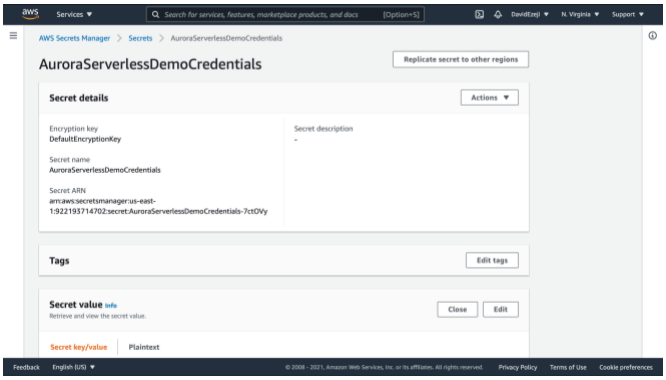
1. Sourced the code for the flask application from Github

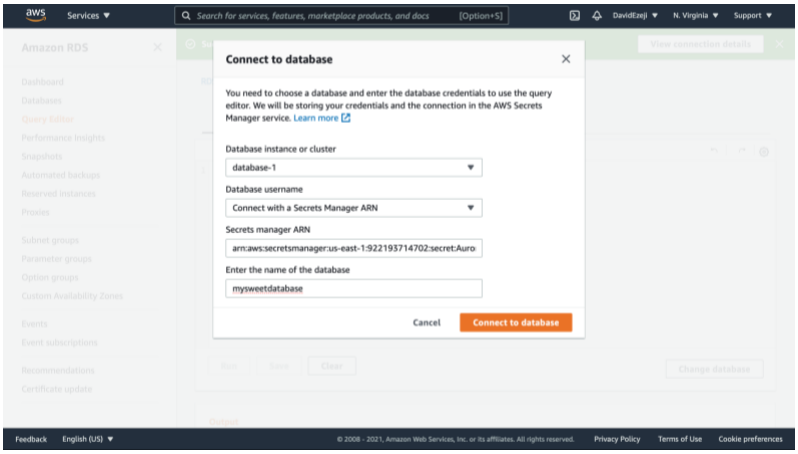
2. Set up a RDS database



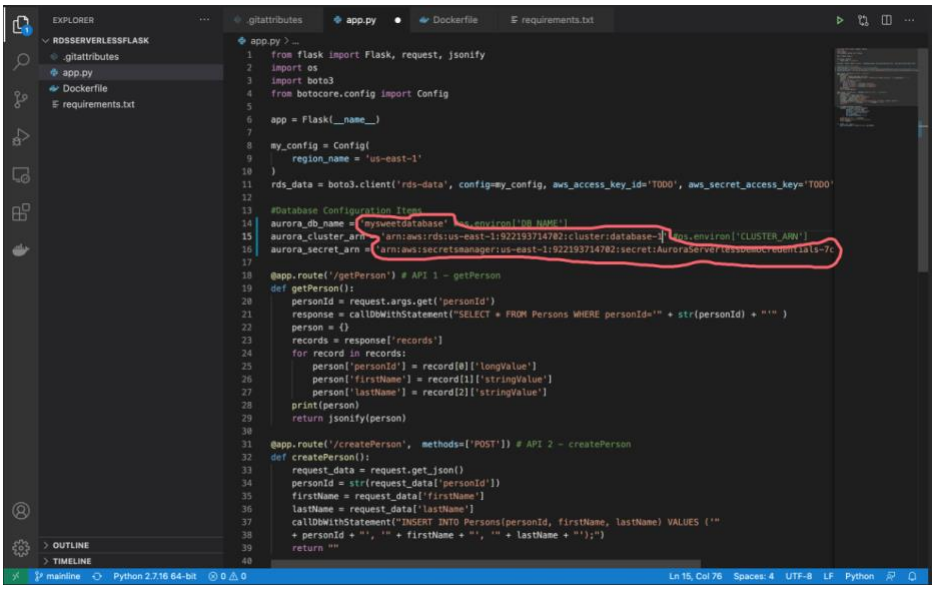
3. Created a secret key that stored my credentials so that I could interact with the database without having to enter my username and password directly (I did this for security purposes)



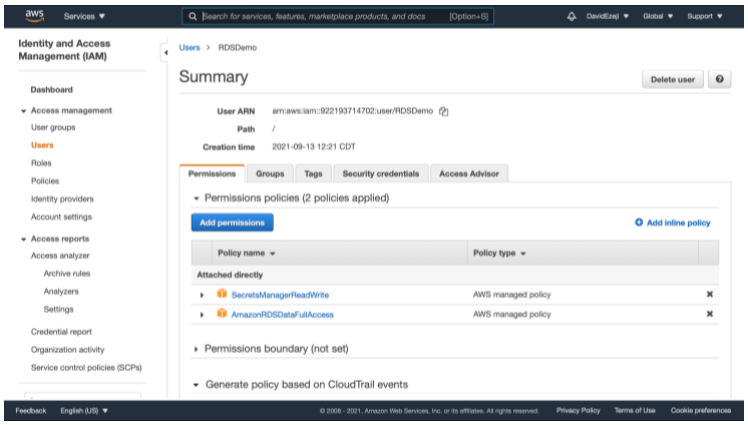
4. Used the query editor for RDS to connect my secrets credentials to my database



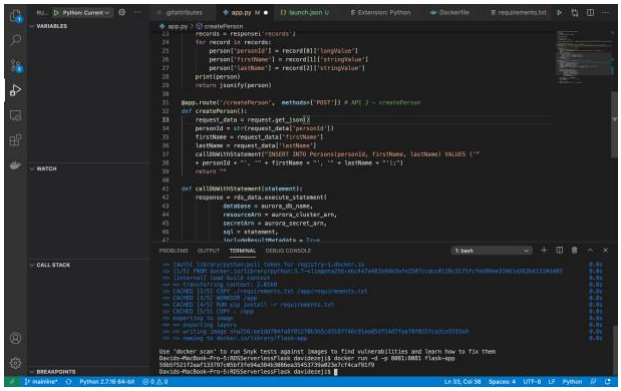
5. Updated the code with info that related to the RDS database as well as Secrets Manager

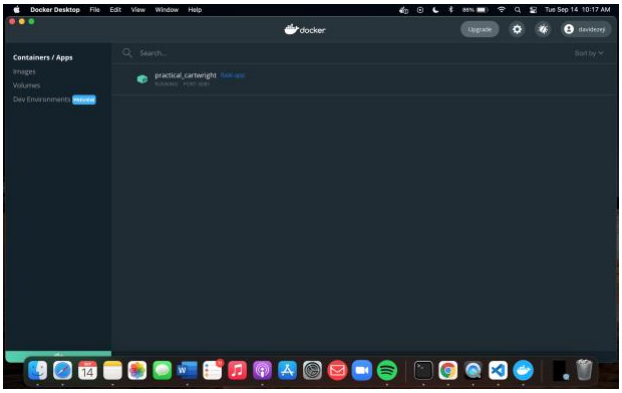


6. Created an IAM user who has access to RDS Aurora (specifically the data API) and Secrets Manager

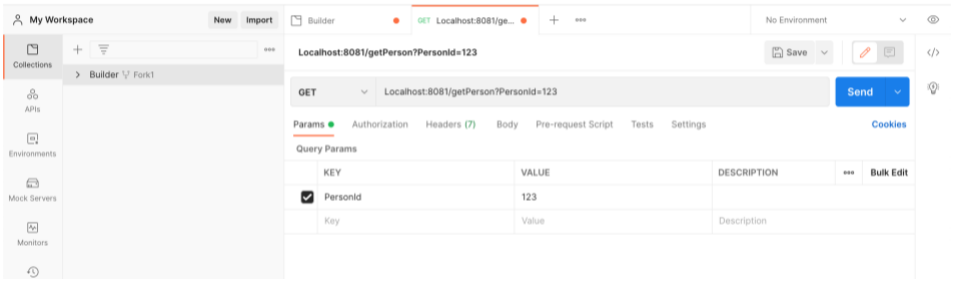


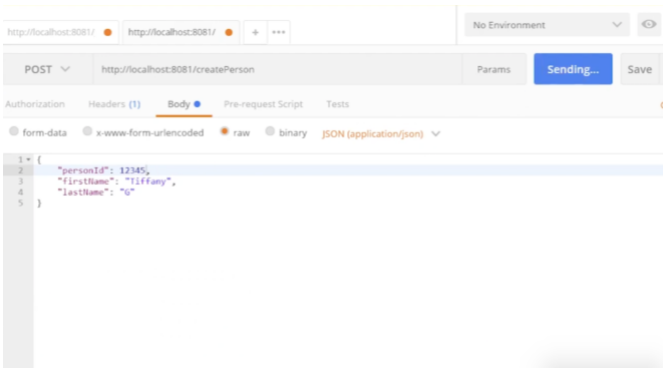
7. Built/ran Docker container





8. Used Postman service to test my API





9. Used the query editor in RDS to confirm that my flask application was now connected to the Aurora database and was responsive