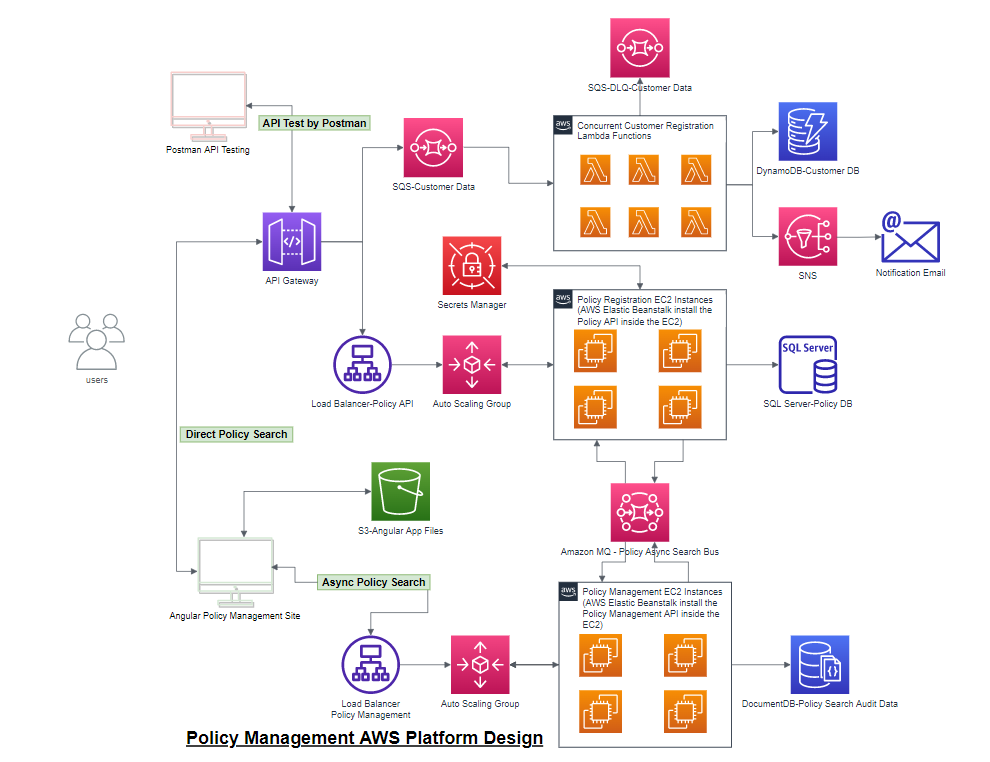
Policy Management System - AWS Cloud Platform Design

Source Code: <https://github.com/ericguousaca/PolicyManagementSystemSolution>

1. Cloud Design
   1. Architecture Diagram

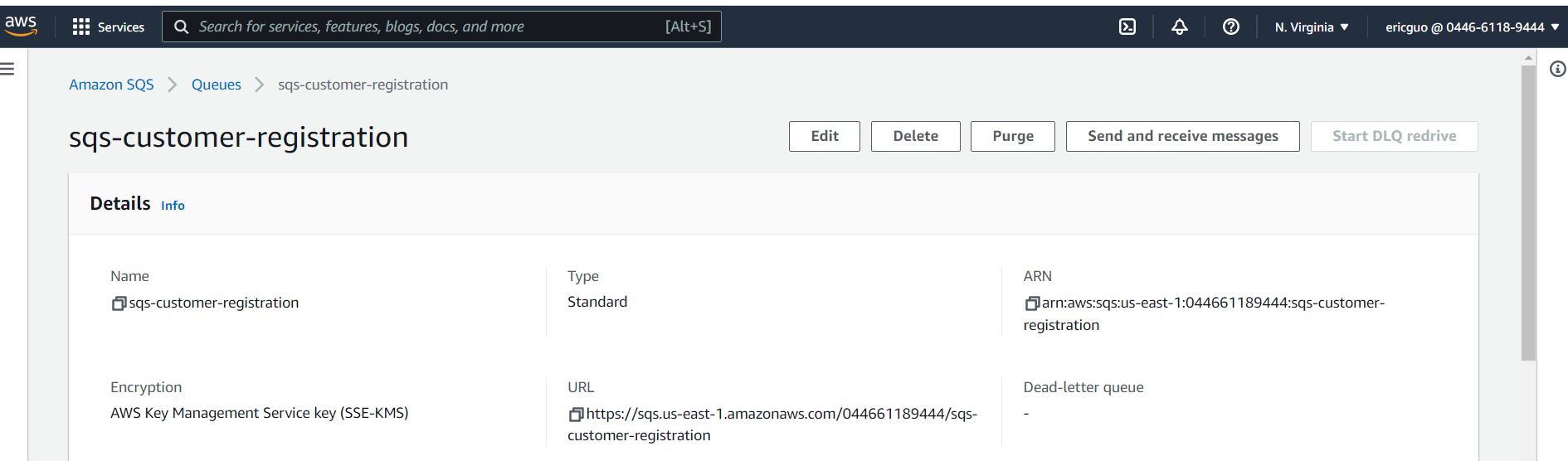


* 1. Main modifications for AWS Cloud Platform
     1. Migrating the micro services solution to AWS cloud platform:
     2. Will replace the Rabbit MQ by using Amazon MQ that is compatible with Rabbit MQ for least code changes in current projects
     3. Will replace the SQL Server Customer Database by using the AWS DynamoDB, but use AWS MS SQL Server Database for Policy DB.
     4. Will migrate the MongoDB to Amazon DocumentDB (with MongoDB compatibility) for least code changes in in current projects
     5. Will replace Customer API micro service by using AWS API Gateway + AWS Lambda + SQS + SNS.
     6. Will replace the Ocelot API gateway by using the AWS API Gateway.
     7. Will update the Policy API micro service to work with AWS SQL Server DB + AWS MQ and deploy to AWS Beanstalk service.
     8. Will update the Policy Management API micro service to work AWS DocumentDB + AWS MQ and deploy to AWS Beanstalk service.
     9. Will update the Angular app and deploy to AWS S3 static website service

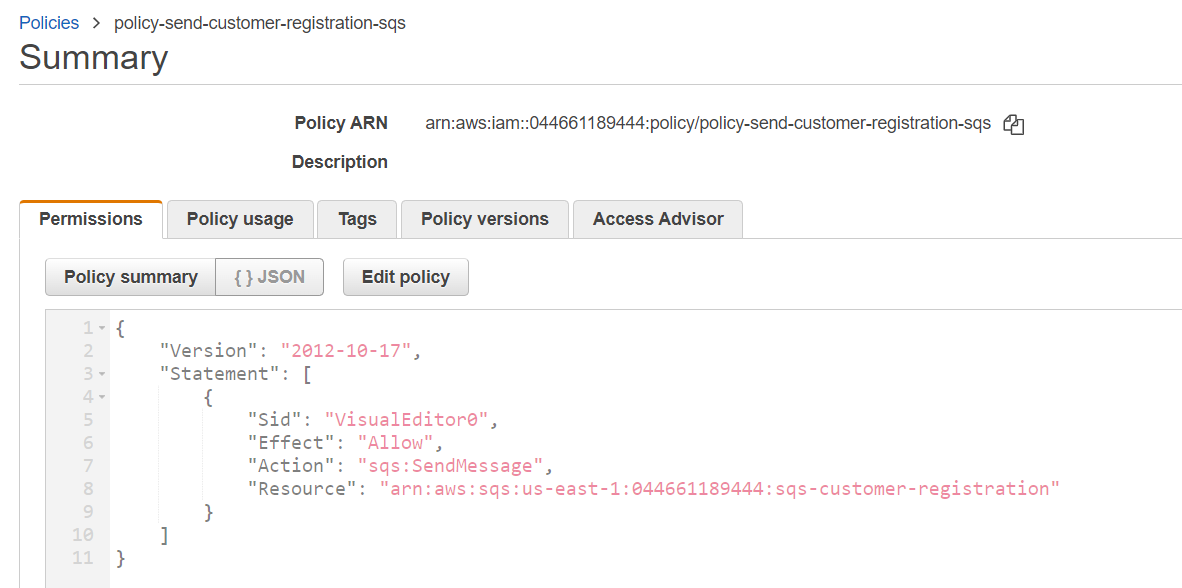
1. Customer API - AWS API Gateway + SQS + SNS + Lambda + DynamoDB Design

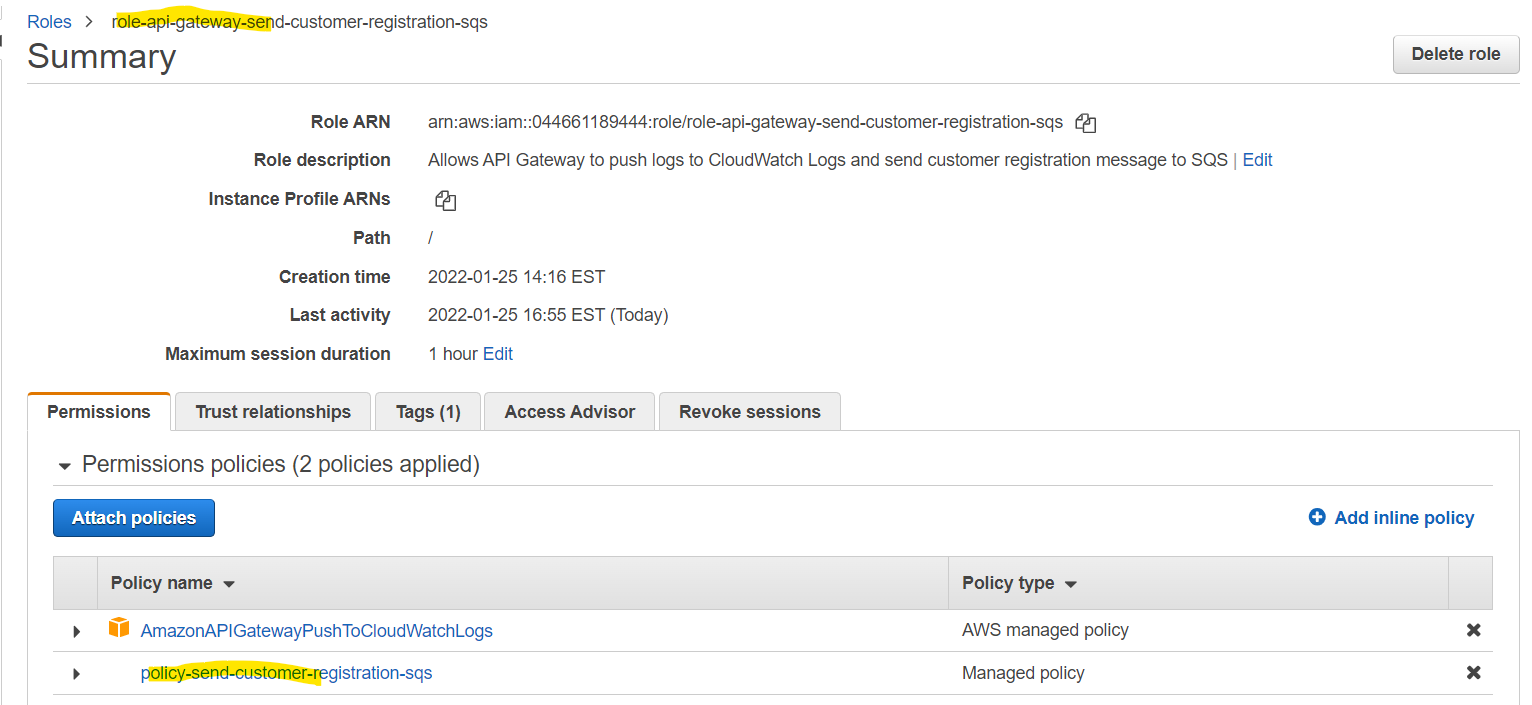
Use AWS Gateway, SQS, Lambda and DynamoDB to implement Customer API Service. Will design below main steps for the implementation:

* 1. Create a new standard AWS SQS to receive the posted Customer Registration data, for example:

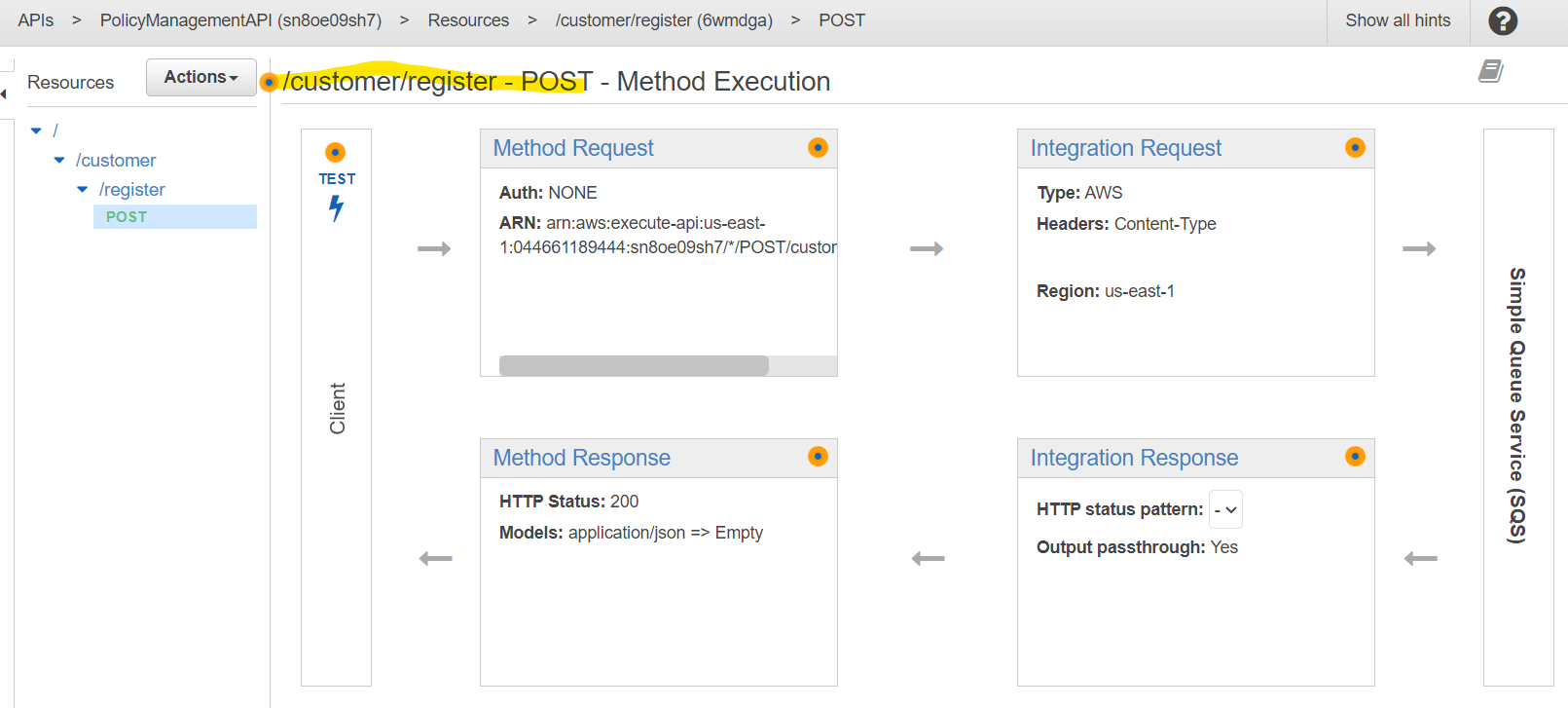


* 1. Add the policy and role that can write the message to above SQS, for example:



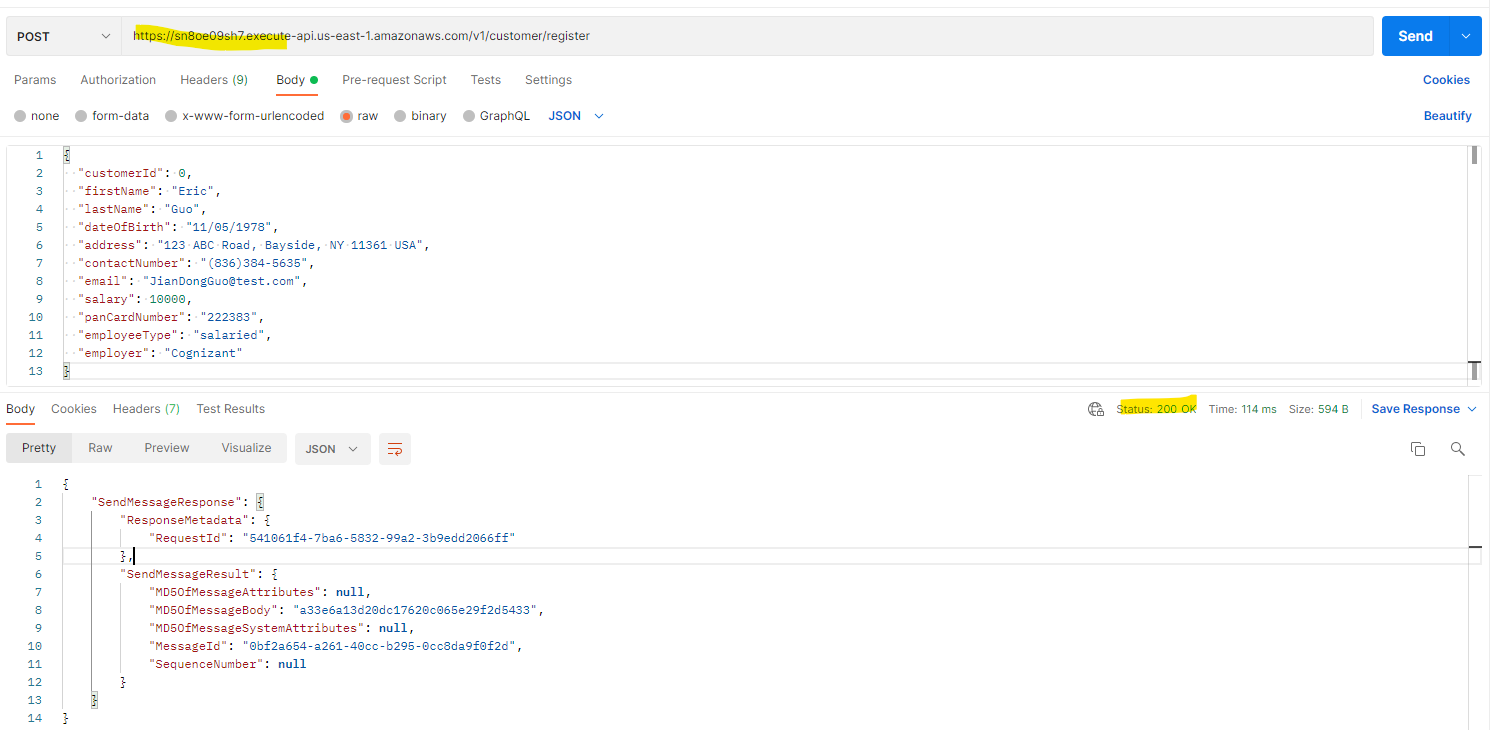


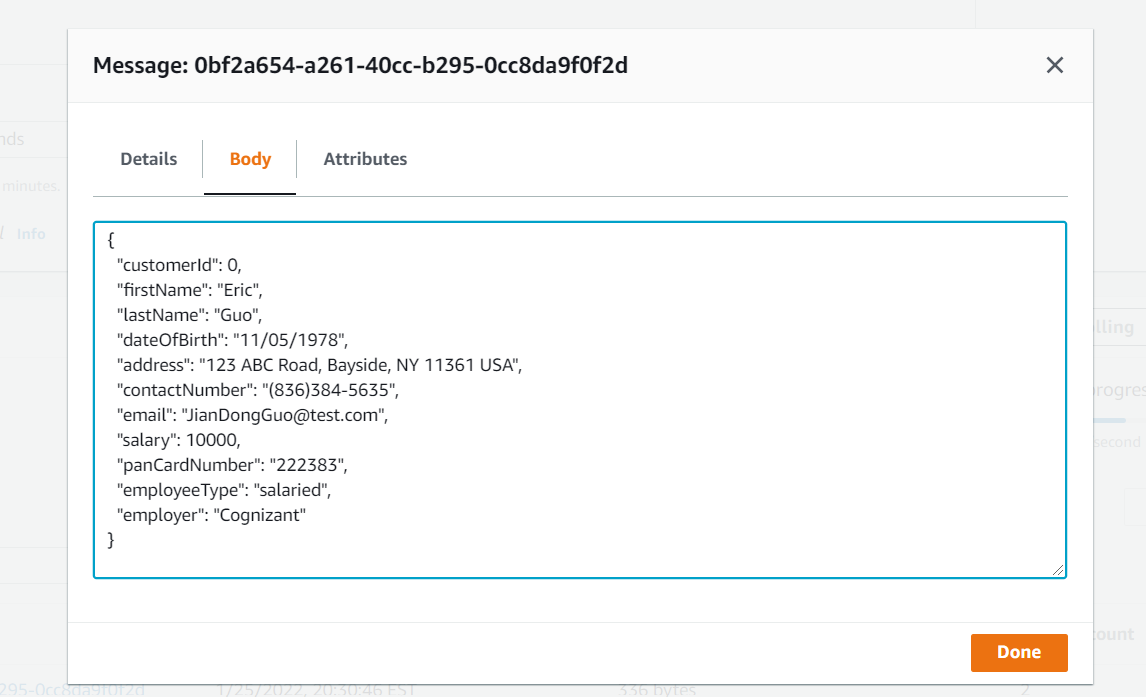
* 1. Create new AWS Gateway, resources and Post method to send new Customer registration data to above SQS queue, for example:





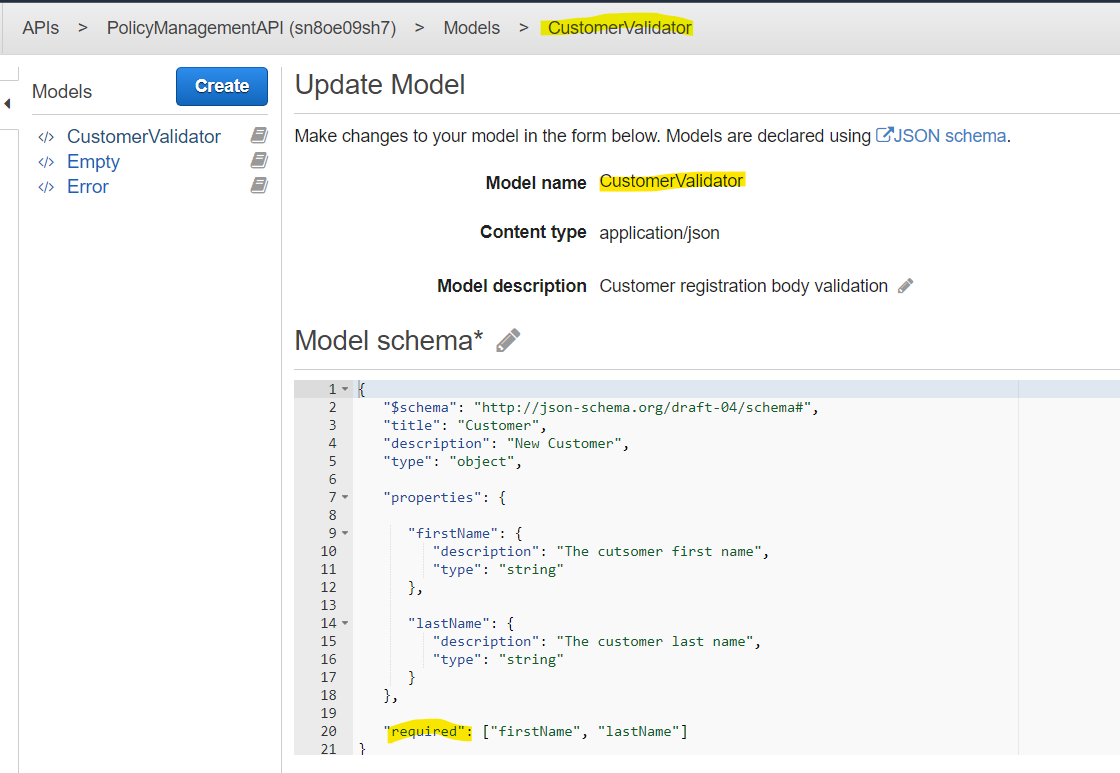
* 1. Make sure gateway can send the message to SQS, and SQS can receive the message, for example, send a message to SQS from Postman and poll the message from SQS:



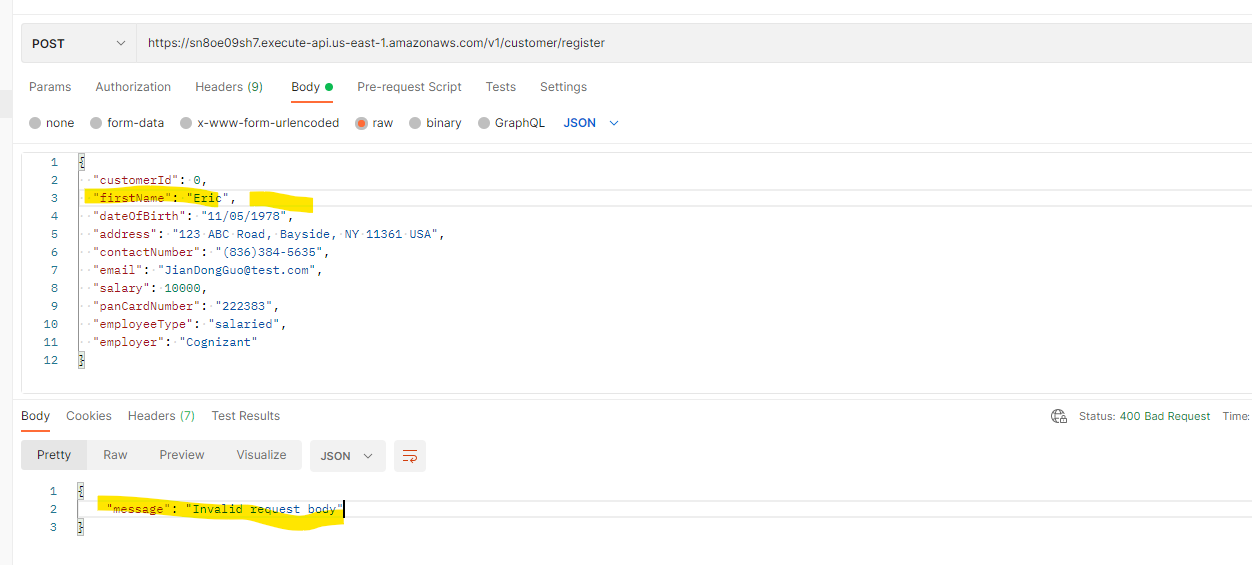


* 1. Since above design sending Customer JSON object to SQS directly, so better to add API Gateway request validation feature by using the JSON Schema object and test the validation properly.

For example, below defined a simple validation model to validate the presenting of “firstName” and “lastName” fields and configured API Gateway to use the validation model to validate the request body of New Customer’s JSON data before sending to SQS queue:



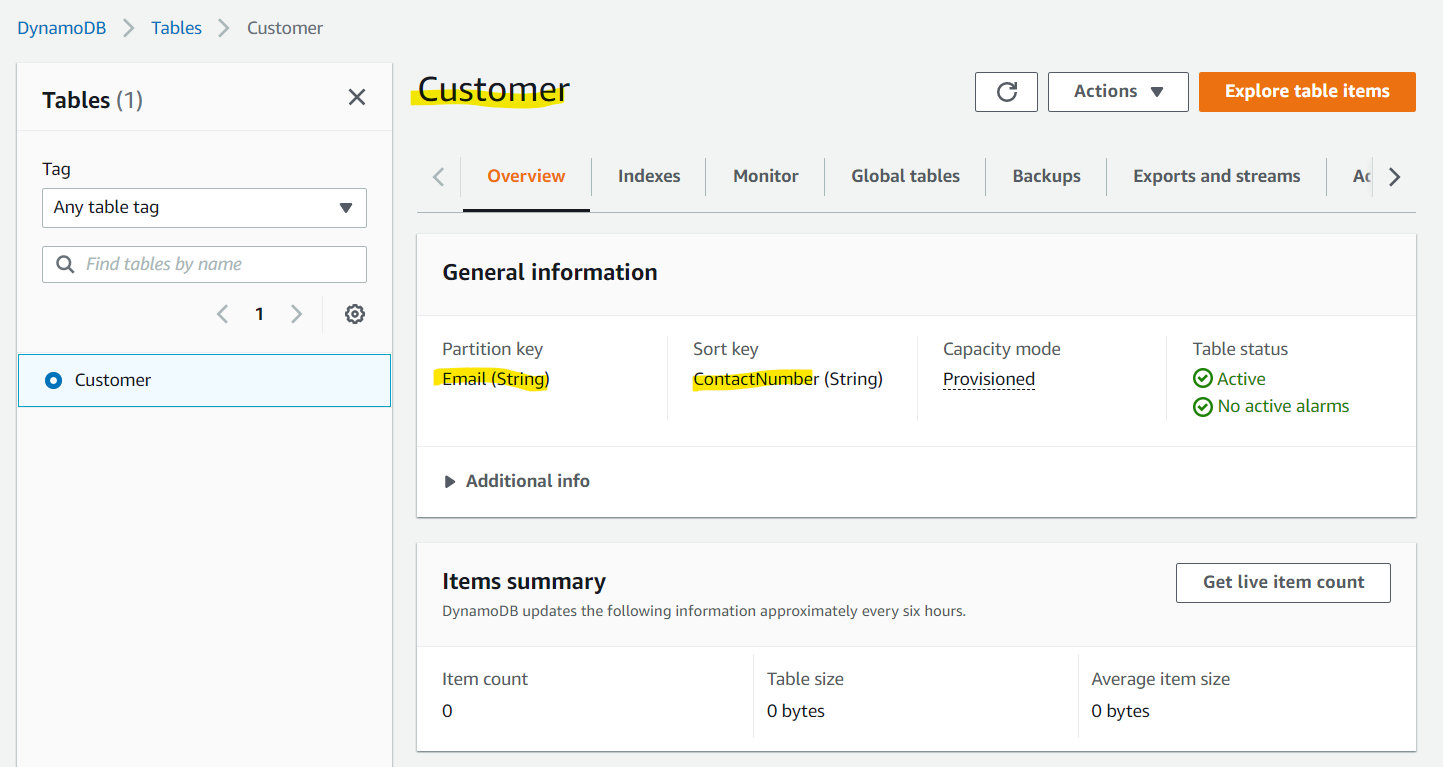


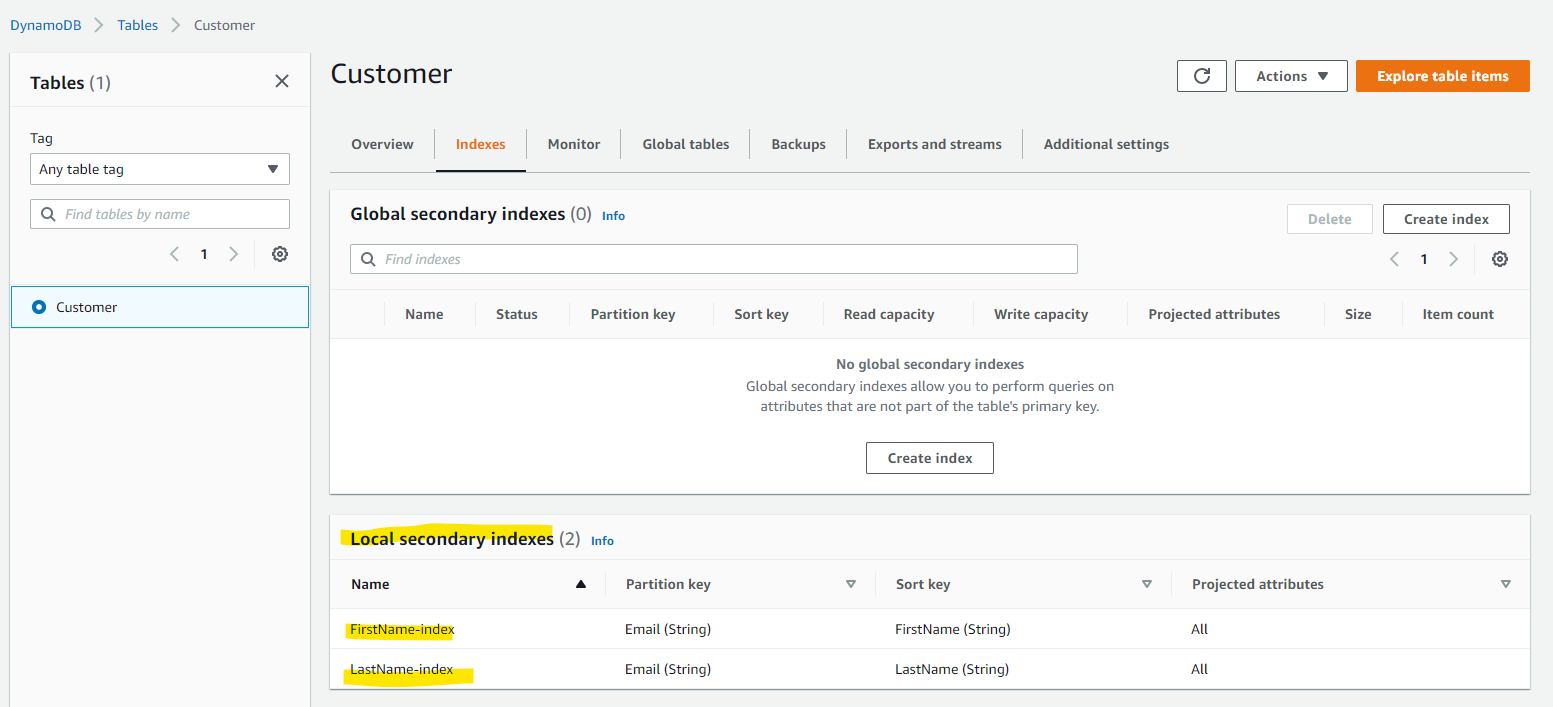


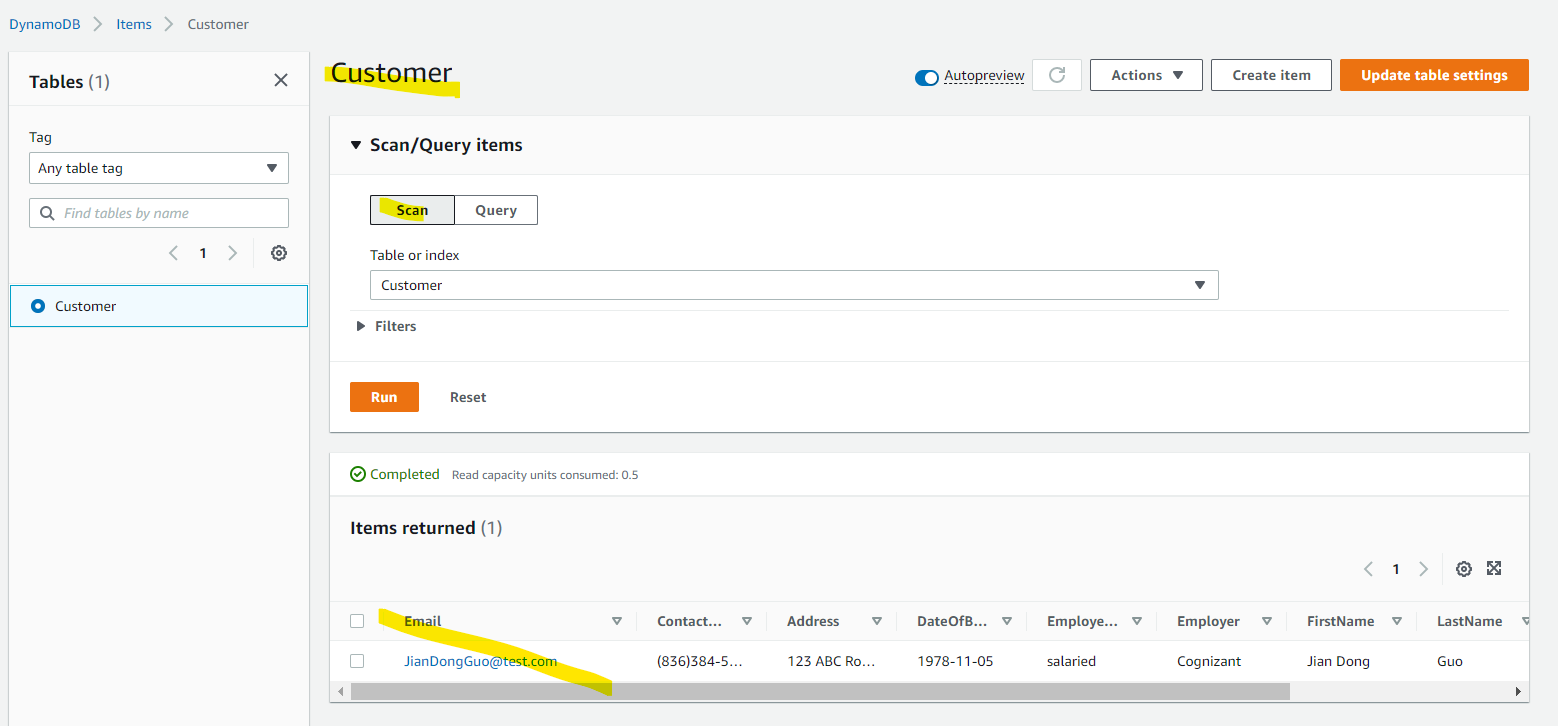
Reject request since lastName is missed

* 1. Or instead of sending request to SQS directly, we can create a middle Validation Lambda function to receive the request from API Gateway and validate it, if passing validation, then sends the request object to SQS from the Lambda, otherwise, reject the request.
  2. Will use AWS NoSQL DynamoDB to store Customer data.

For example, can create a Customer table in DynamoDB with below settings and sample data, when in development, may need to adjust the settings:



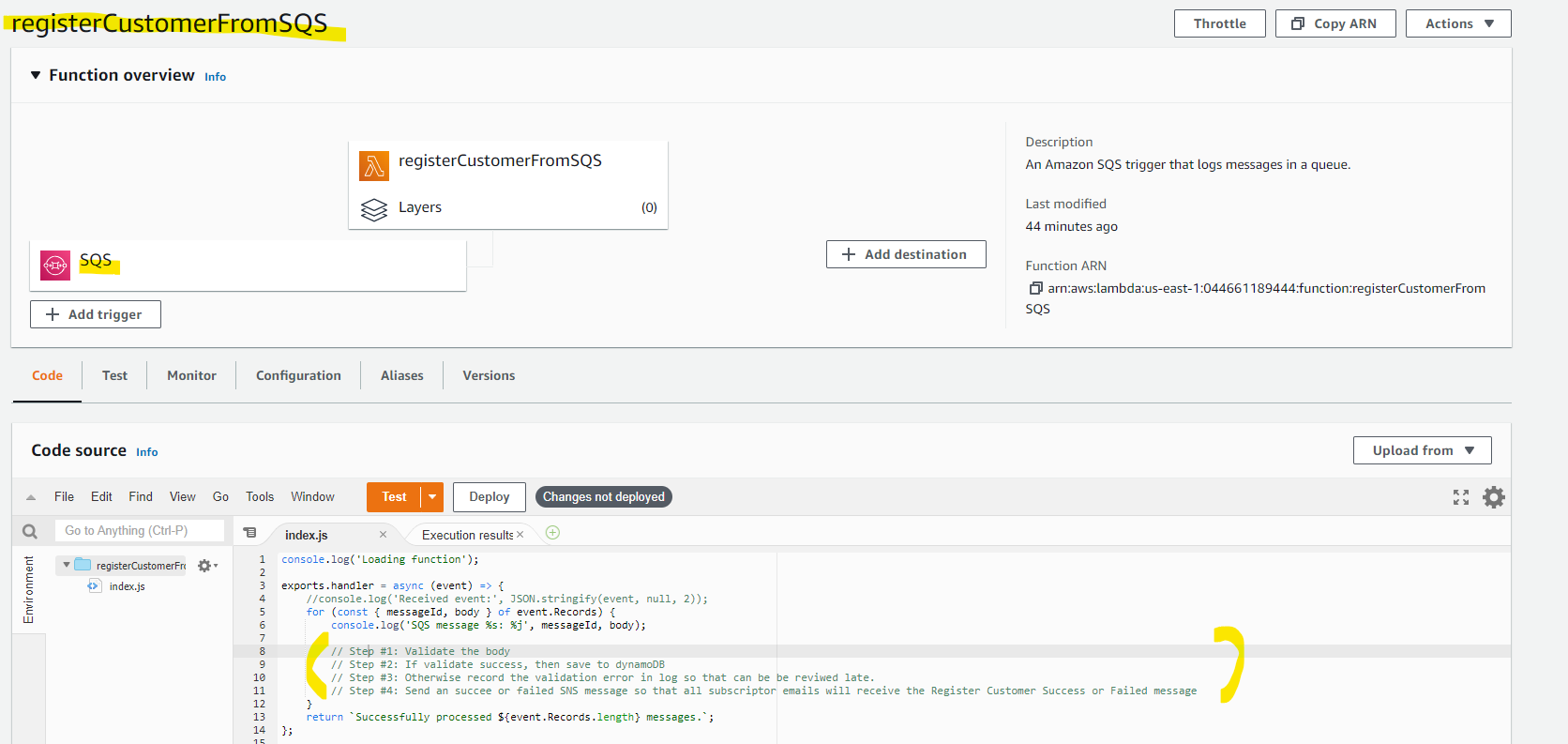




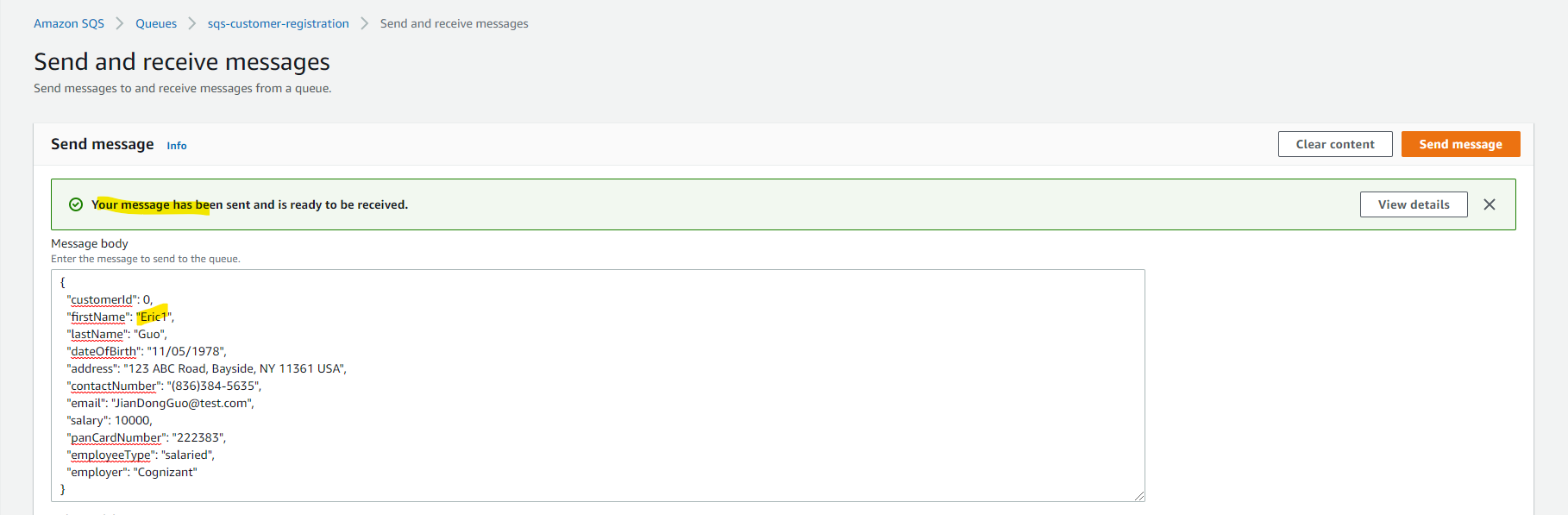
* 1. Create a Lambda function to process the SQS message. The Lambda function will poll the message from SQS in batch and process them one by one. For each message, will validate the body first, if pass the validation, then save to Customer table in DynamoDB, otherwise record the validation error and body data in the log or DB for future review, and then send a success or failure email message to AWS SNS topic, the SNS topic will deliver the message to all subscriptions (Emails).

For example, below is a simple Node.js Lambda function to process SQS message (In the implementation phase, the Lambda function can be created by VS 2019 with using Asp.Net Core technology + AWS SDK packages):

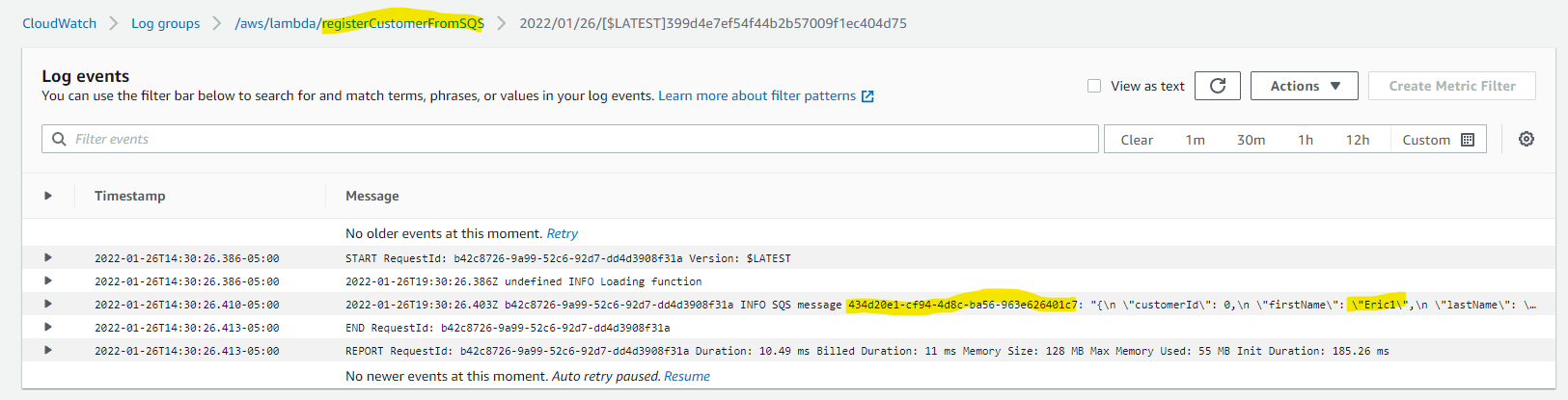
* A simple Nod.js Lambda registerCustomerFromSQS function to process the SQS message:



* Sent a test SQS Message:



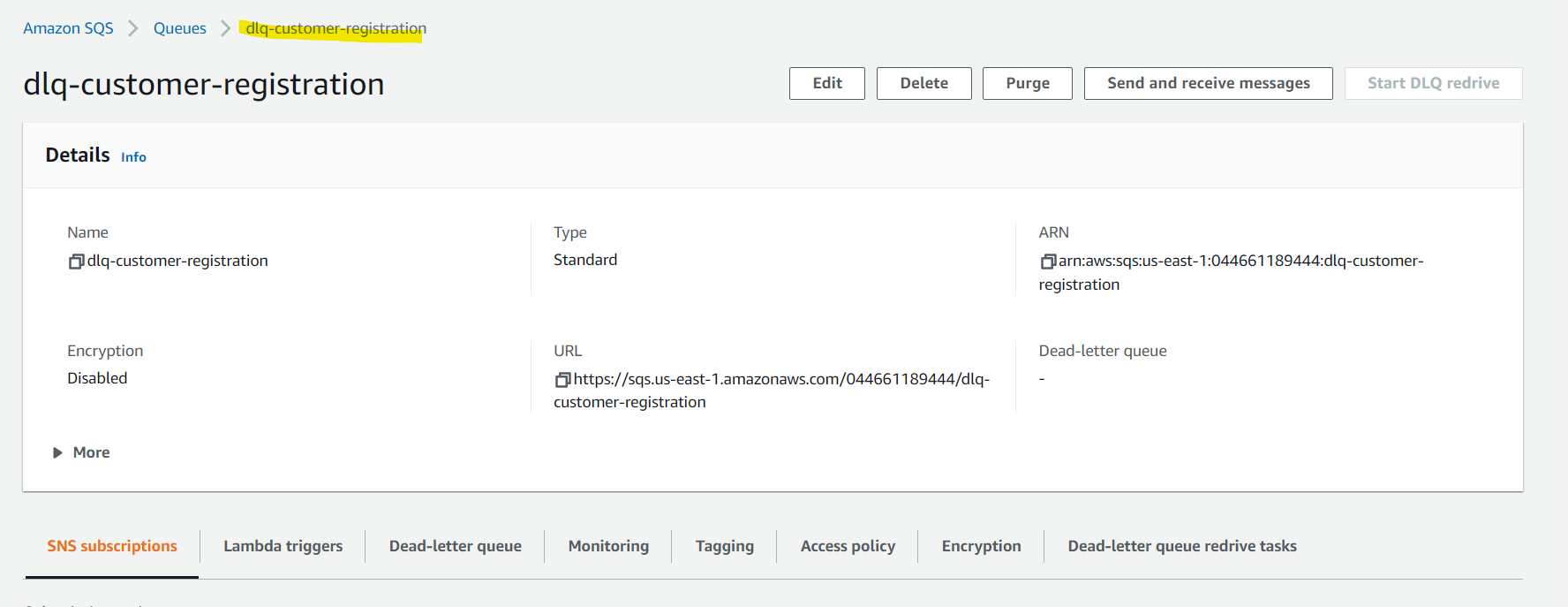
* Validate the message had been processed by the lambda function in Cloud Watch:



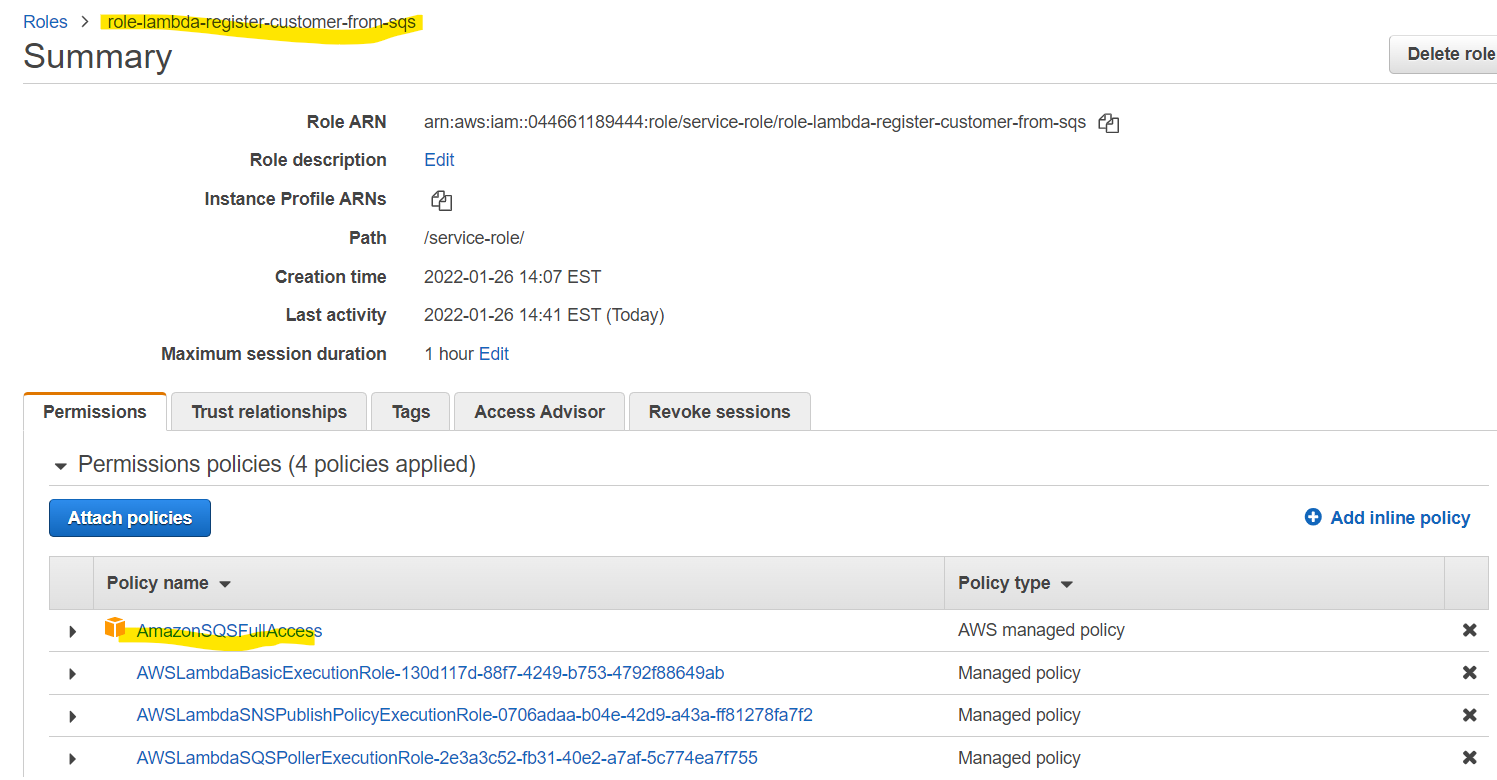
* 1. Since above approach will asynchronous invoke the Lambda function. By default, the Lambda function will retry 2 times if there are errors, so better define a SQS dead-letter queue (DLQ) so that Lambda function can move the failed message to DLQ for future review and process.

For example, create below Dead-Letter queue, give the Lambda role enough permission, and then attach the DLQ to the Lambda function:

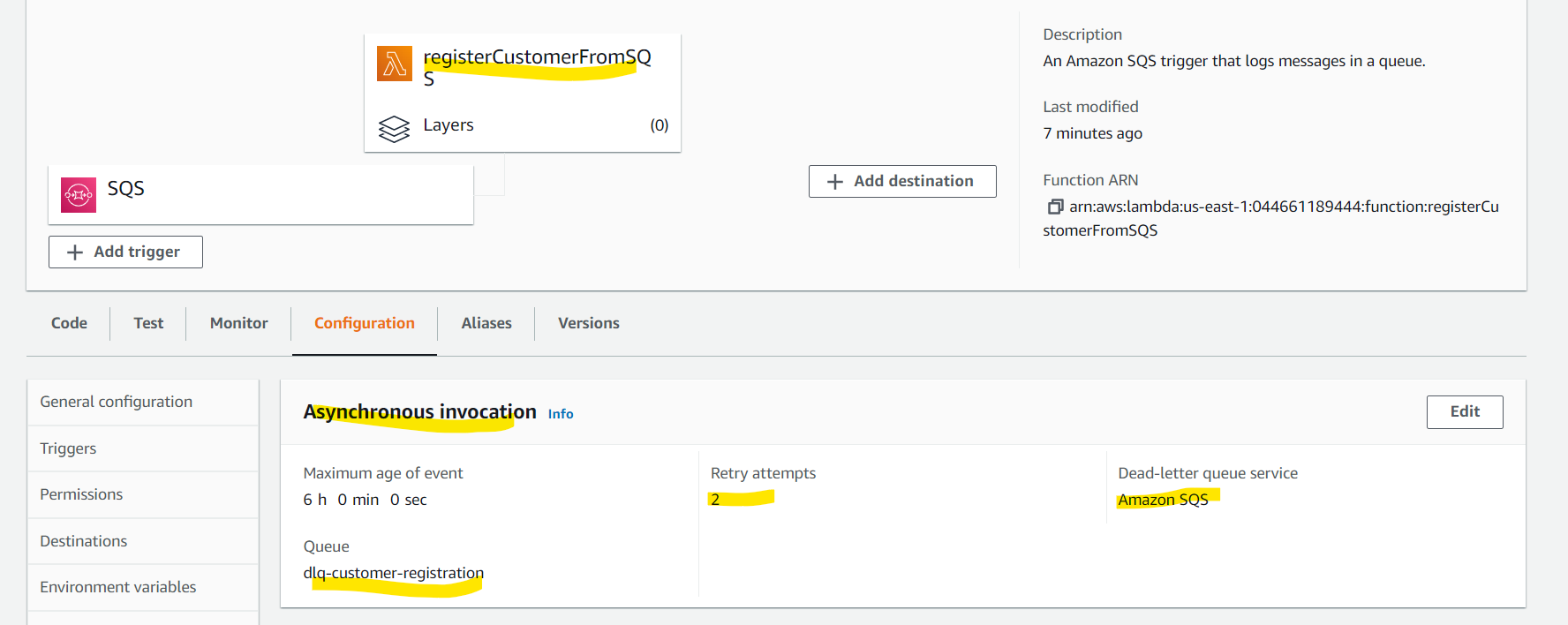
* Create a Dead-Letter queue:



* Just simple giving full access in design phase for Lambda role in this demo (shall not give full access in implementation phase):

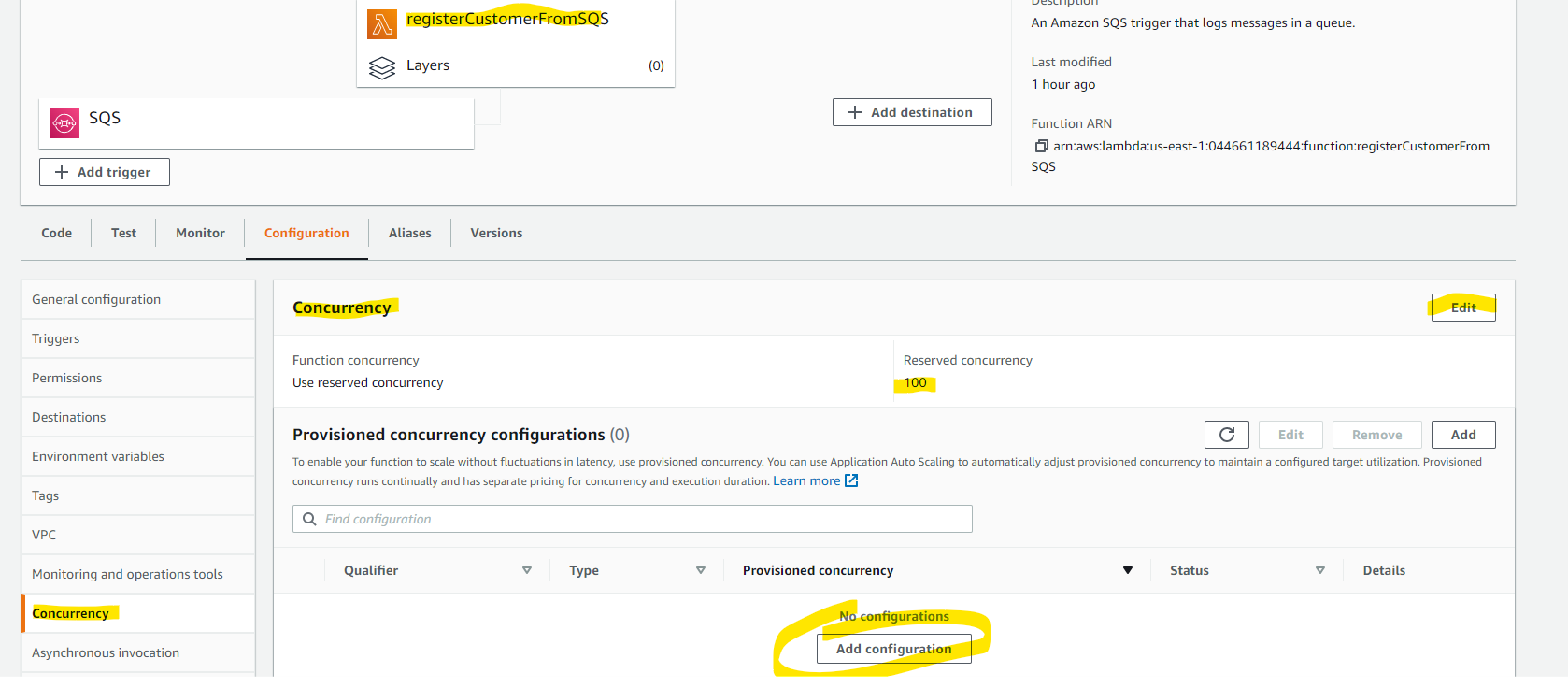


* Attach the DLQ to the Lambda function in the Asynchronous Invocation setting, in the implementation phase, make sure to test the Lambda + DLQ can work as expected:

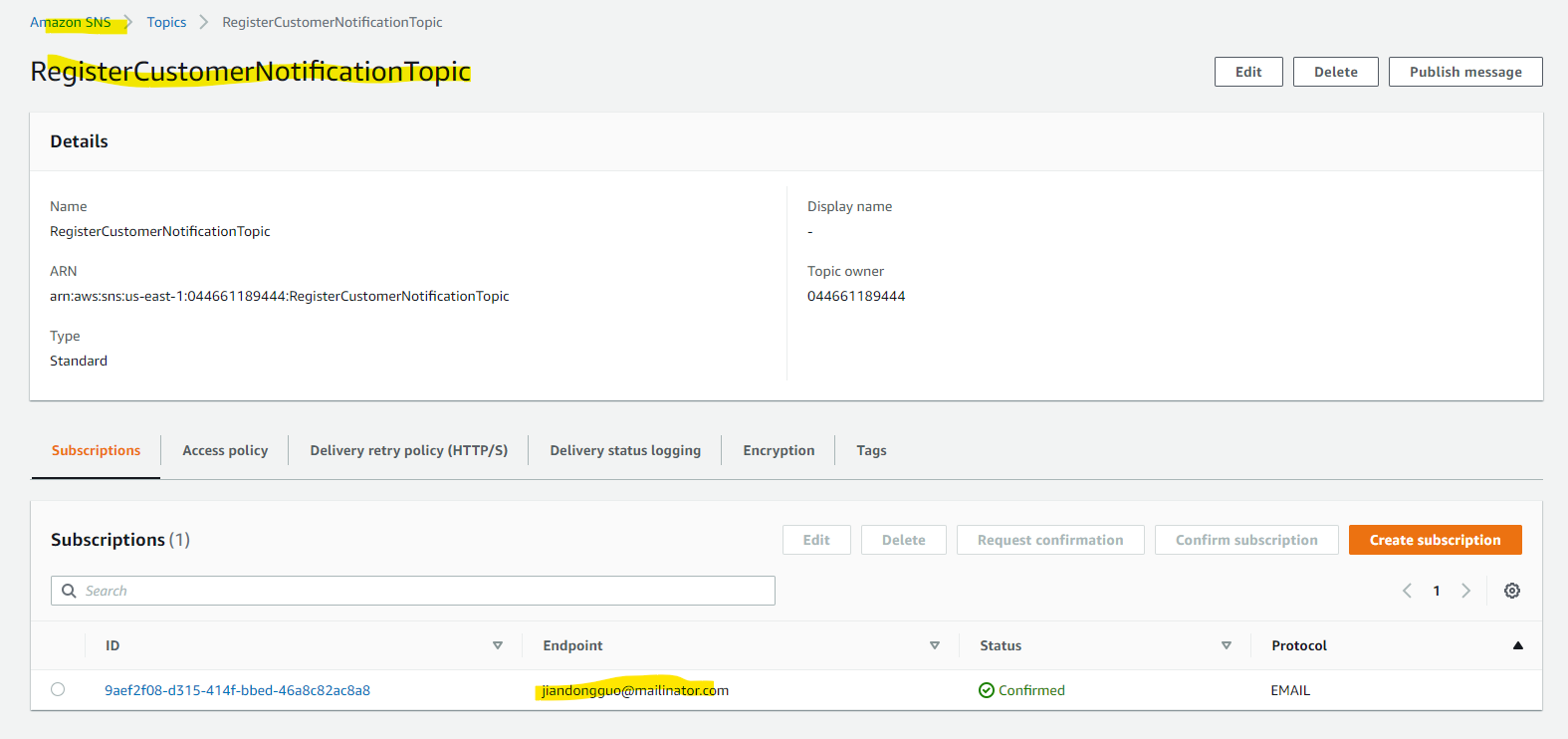


* 1. The Lambda service has default 1000 concurrency executions limit, so we shall set a fixed concurrent limit for the Customer Registration Lambda function to prevent it runs out all 1000 limit (in such case, other Lambda function will throw the throttling error since no Lambda function can be instanced)

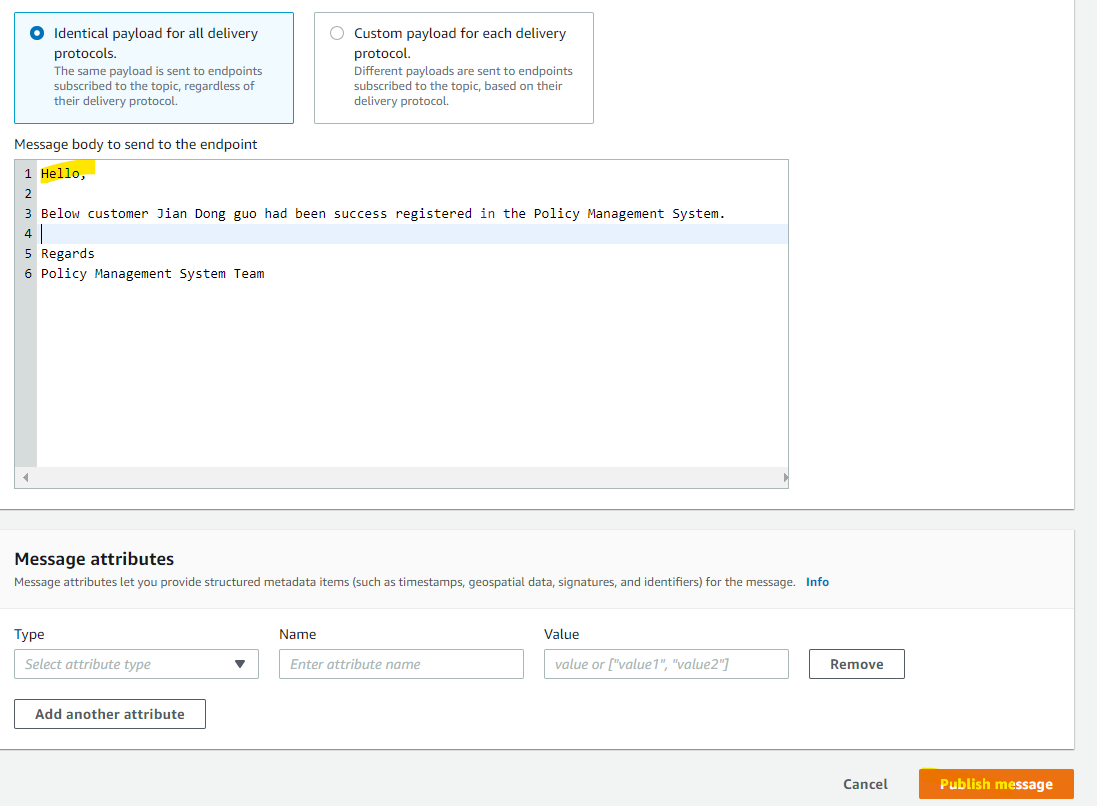
For example, we can set the 100 concurrency in blow setting, also in implementation phase we can set the provisioned concurrency too to speed up our Lambda function in cold start phase:

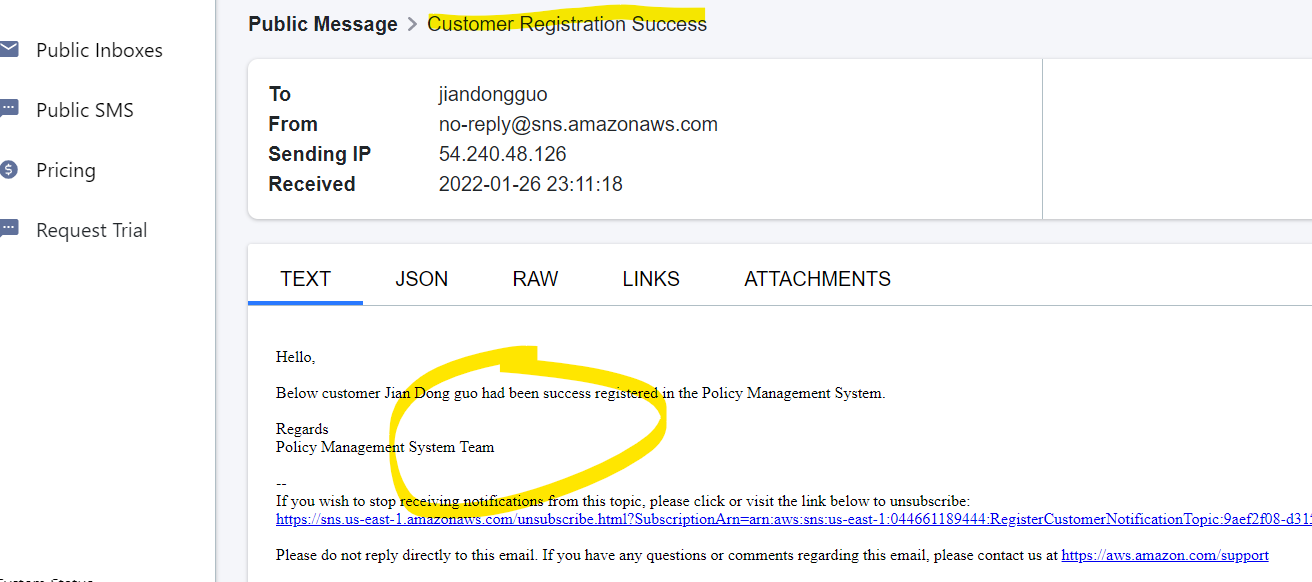


* 1. In order send the notification email to user, need to set up a SNS topic. For example, can set up below SNS Topic and test the email notification, in the implementation, The Lambda function will send the formal success or failure notification to the subscribed emails:
* Create a test SNS and email subscription:



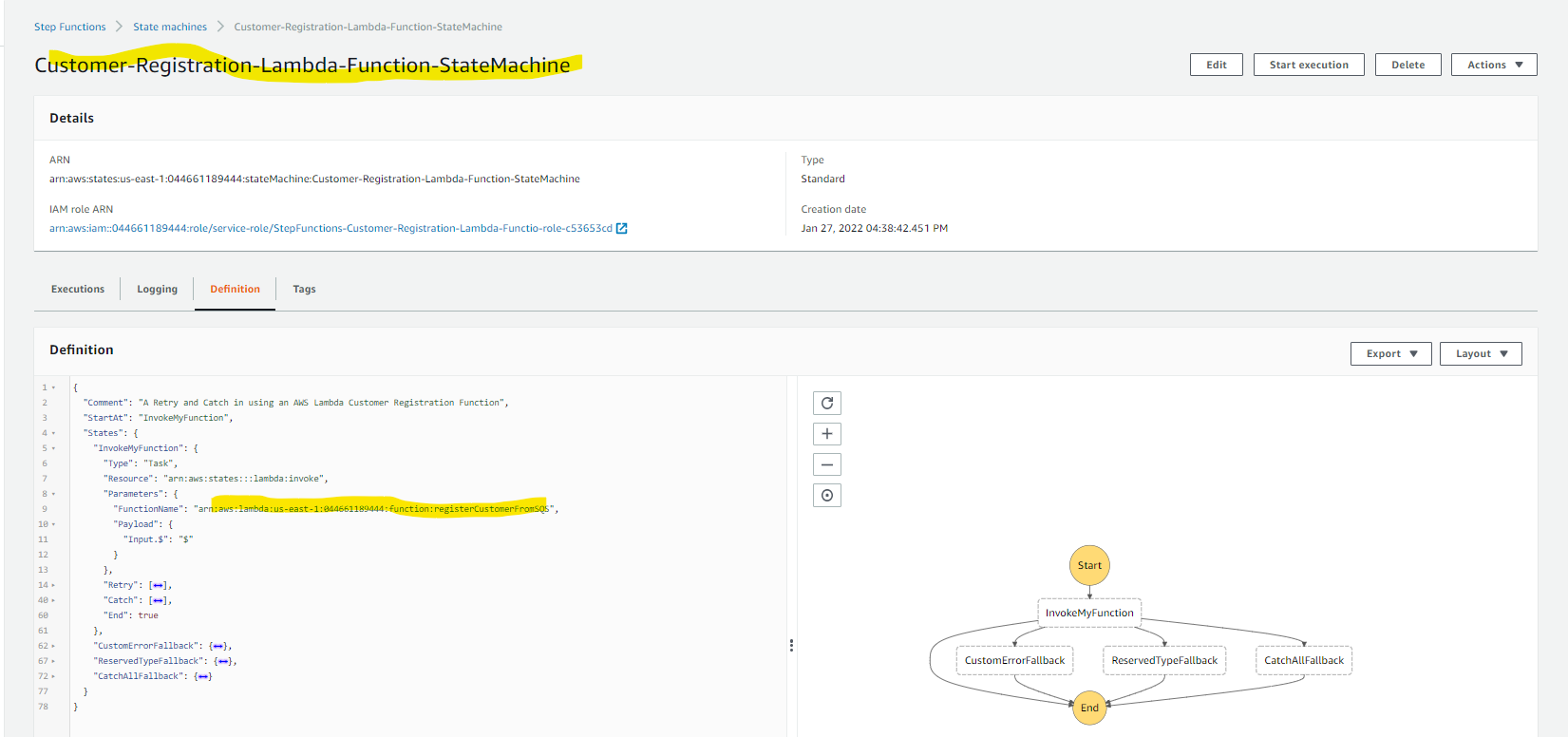
* Test send and receive sample success email:



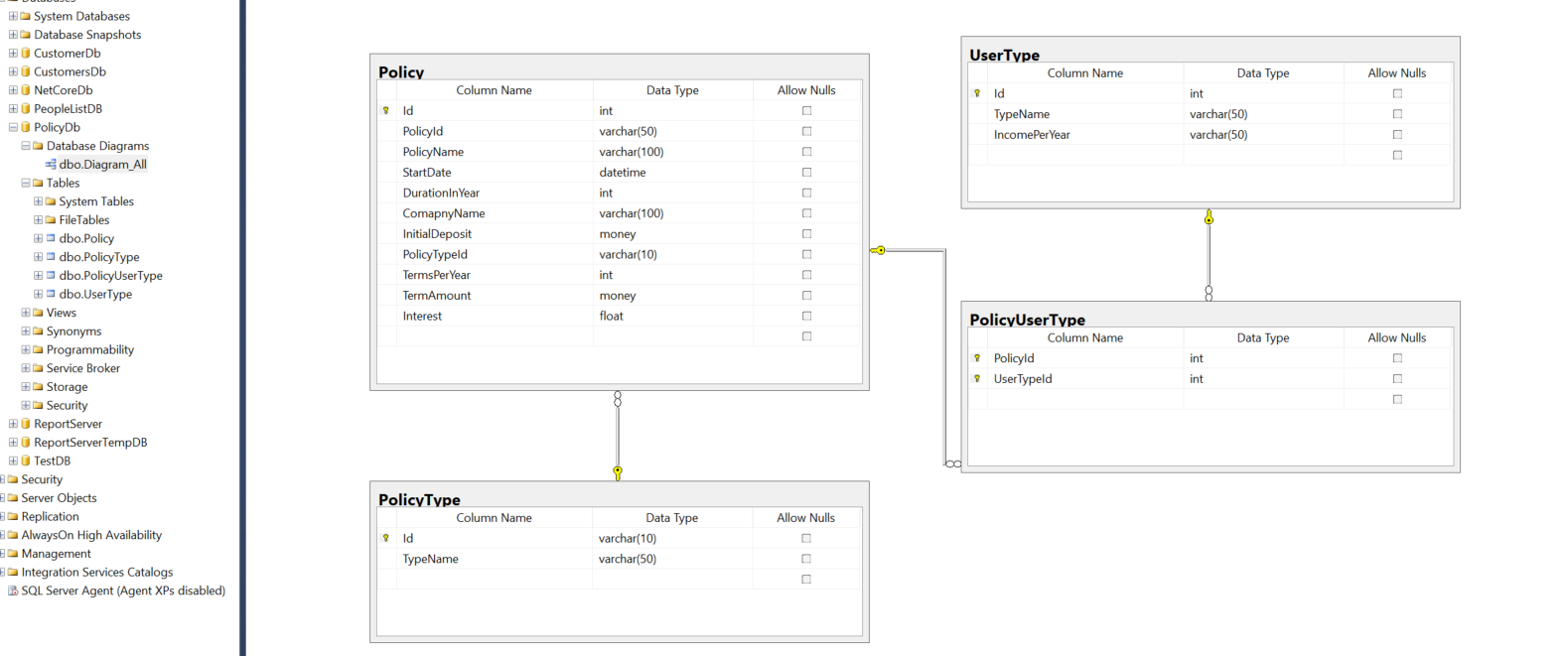


* 1. Lambda function may throw exception, like timeout exception, service side exception. It is better to handle those exception in Step Function instead of inside the Lambda function.

For example, we can define below Step Machine to handle those Exceptions for the Customer Registration Lambda function, in real implementation, shall test those exceptions and error handles properly:

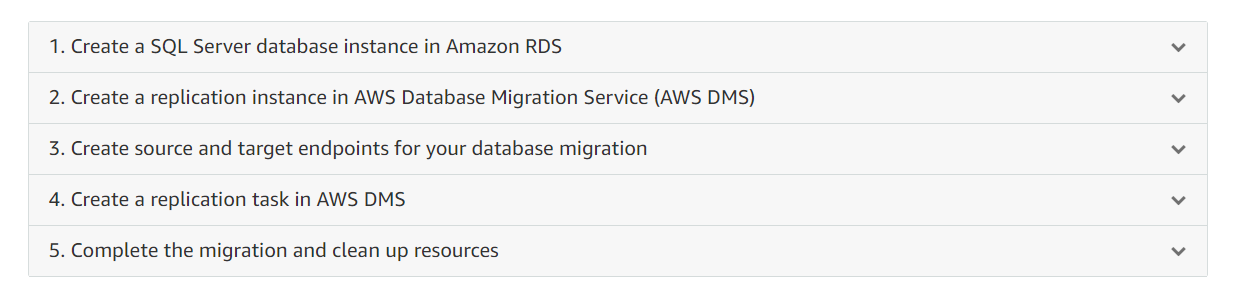


1. Policy API - AWS Elastic Beanstalk + AWS RDS + Amazon MQ Design
   1. Since Policy Registration Process have below complicate data relationship, so it is better keep use the Relationship database instead of using the NoSQL database (like AWS DynamoDB or AWS Document DB since they are hard to handle the data relationship).



* 1. We can use the AWS Database Migration Service to help us migrate our local database to AWS RDS (SQL Server DB), the detail process can be found in below link:

<https://aws.amazon.com/getting-started/hands-on/move-to-managed/migrate-sql-server-to-amazon-rds/>



* 1. Since the on premise Policy API micro service uses Rabbit MQ to asynchronies process the Policy search. And Amazon MQ can support RabbitMQ so we can find a related easy to update our code to use the Amazon MQ service.
* First we shall create a new Amazon MQ broker with correct configurations
* And then we can update below end point to AWS MQ Endpoint in the project Startup.cs code

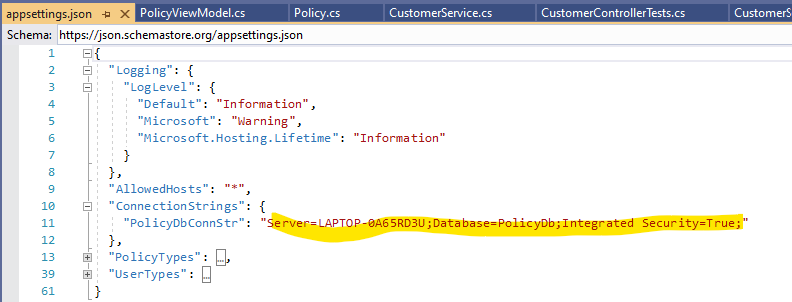




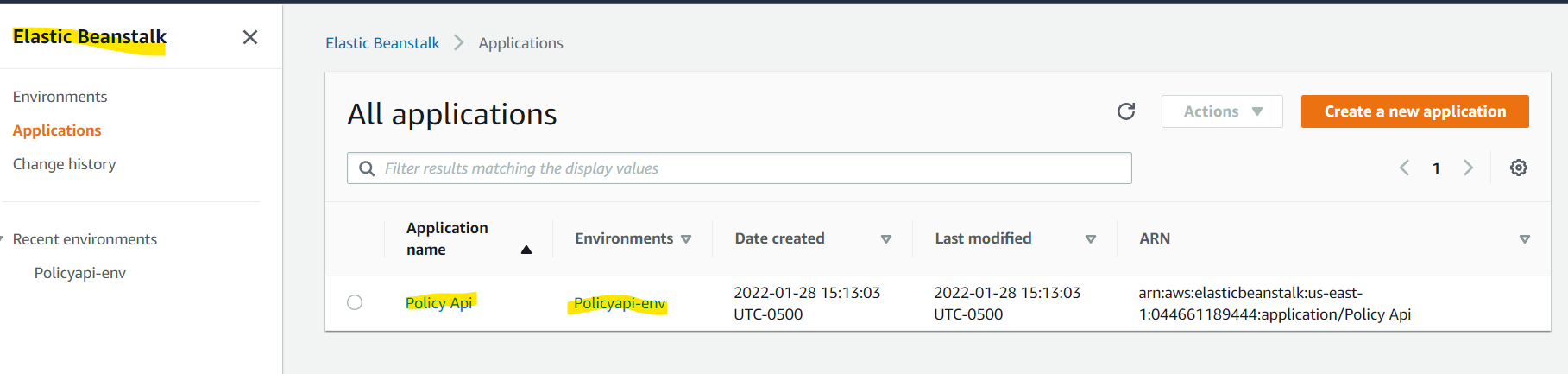
* 1. Another approach is using AWS SQS to replace the Rabbit MQ, and then we need to replace the MassTransit code by using AWS SDK to send and receive the SQS message, and also need to implement a new Asp.Net code background Process by IHostedService to act as the built-in MassTransit Hosted Service, this approach needs more code changes in current project.

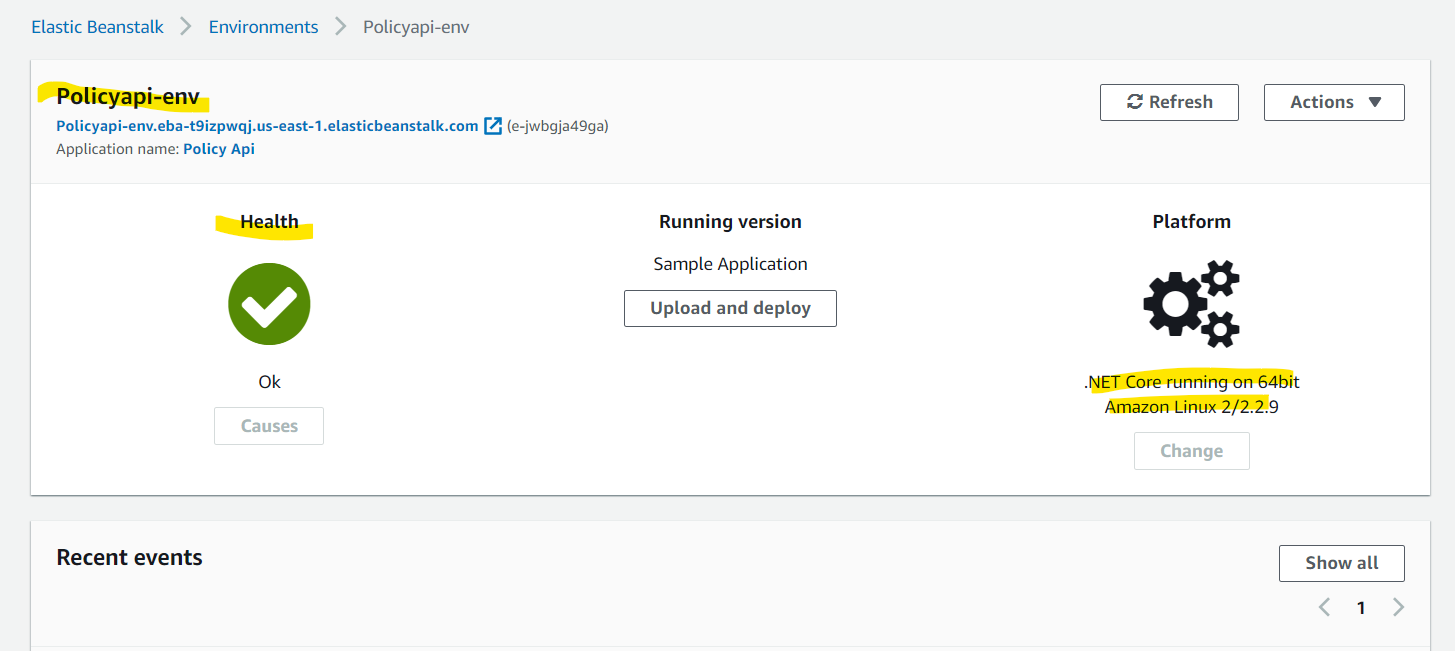
Basic steps include:

* Create a new AWS SQS queue to send and receive message for Policy Search
* Modify the Policy API by creating a background service (implementing the IHostedServie interface), and then inside the background service we can use AWS SDK to send, receive and process message and search results from AWS SQS as current Rabbit MQ + MassTransit implementation.
  1. Current Policy Api Project simply use below App settings to store the sensitive configuration data, like the DB Connection string, we can modify to use AWS Secrets Manager to store those Sensitive configuration data and use AWS SDK to retrieve those secret data inside the app.

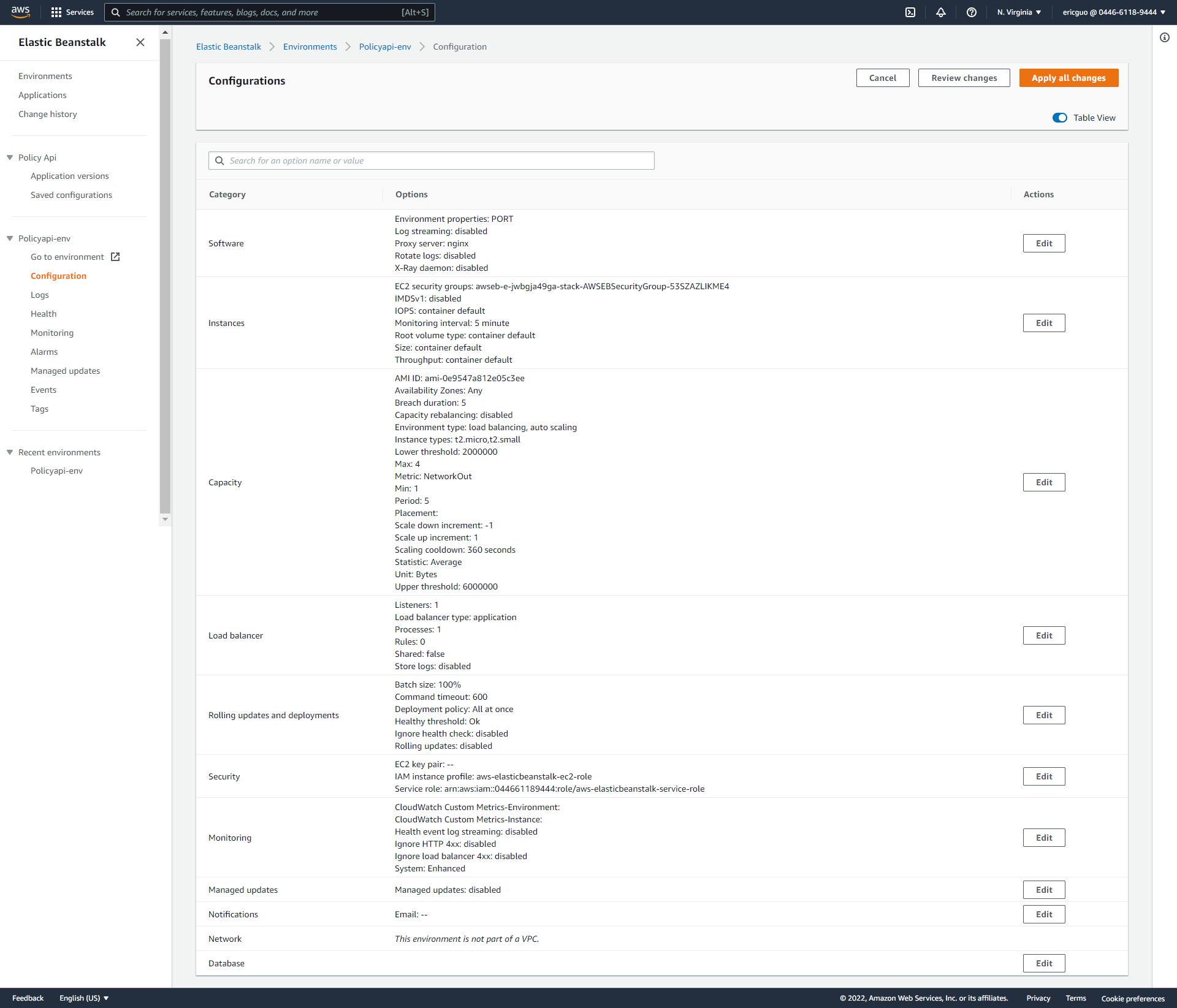


* 1. After above modifications are done, we can deploy the Policy Api Micro service to AWS Elastic Beanstalk. For example, we can create below new Policy Api Application and a new environment Policyapi-env for the application:

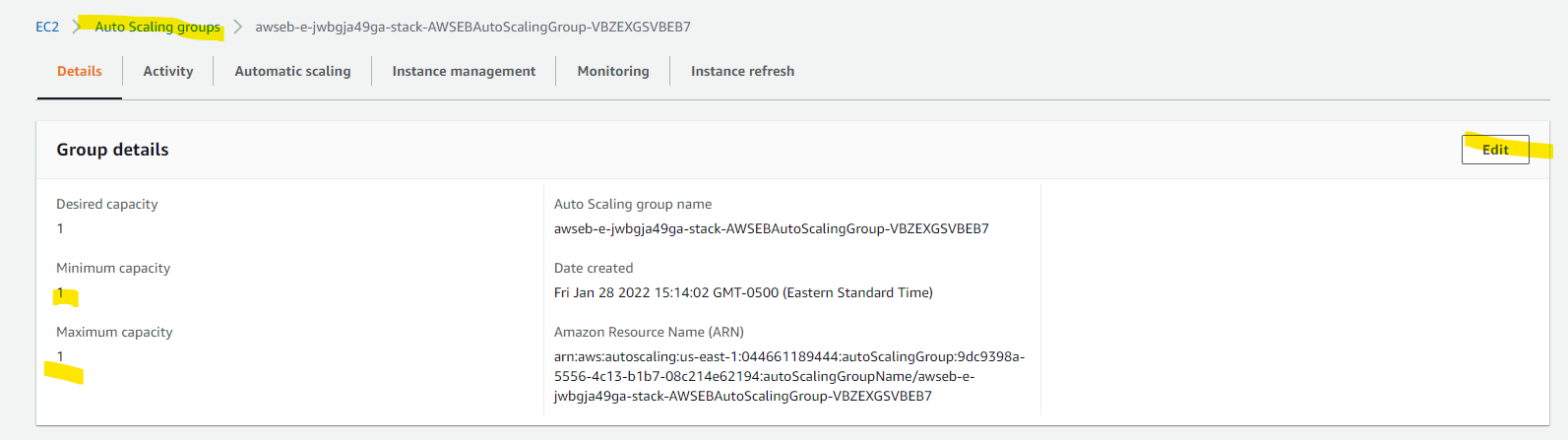




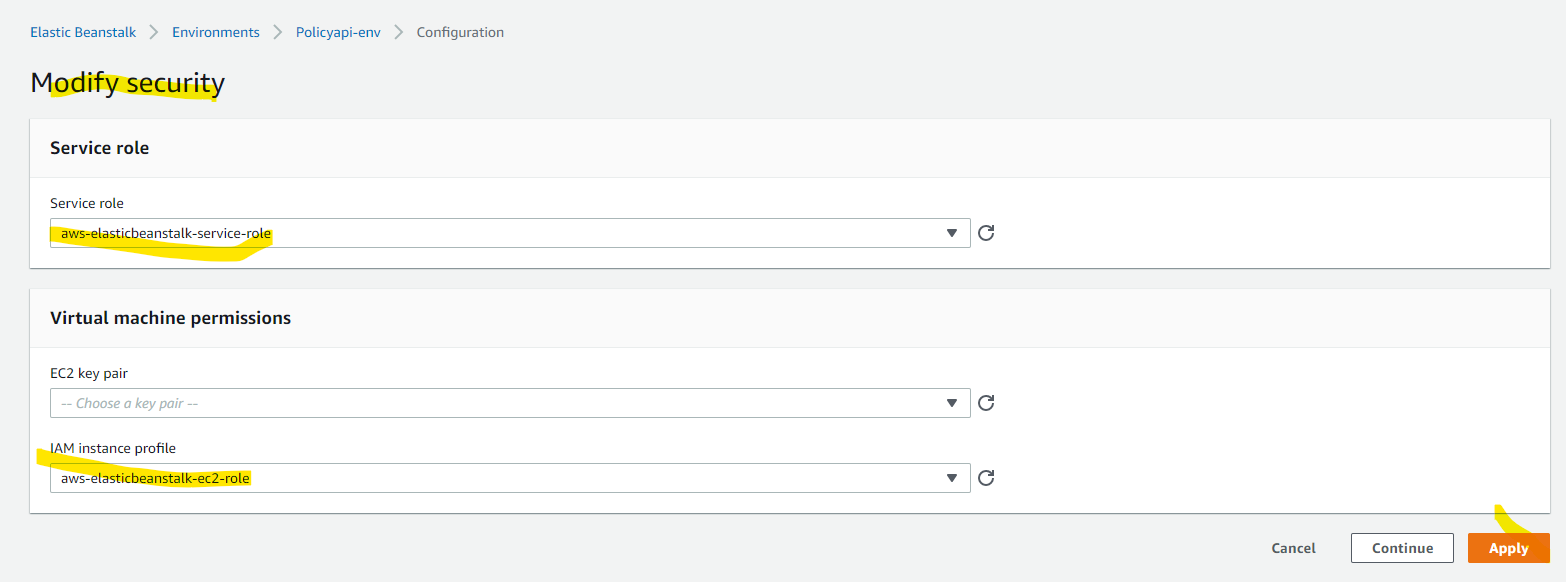
* 1. Above Application Environment automatically generated a list of resources when the new beanstalk environment is created:

For the Policy Api to run the properly and securely, may need to consider update below resources:

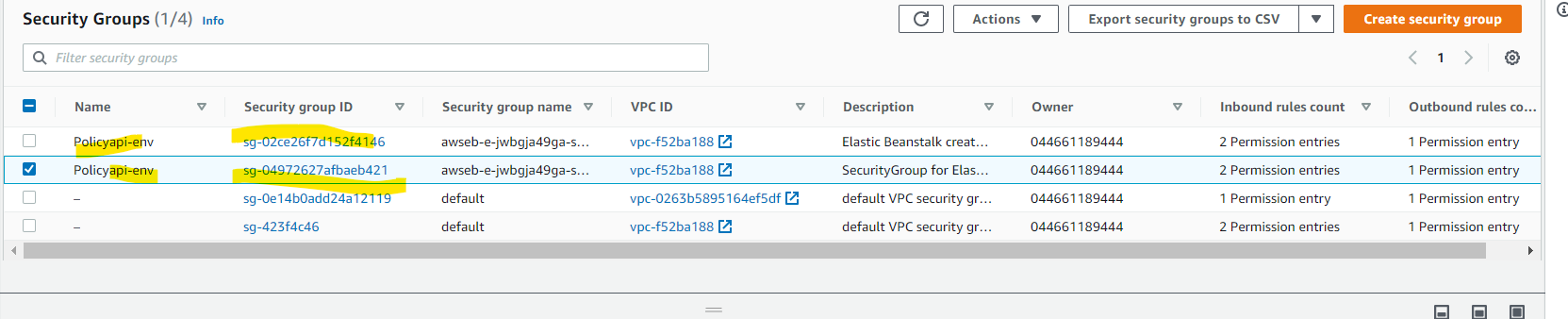
* Set the proper Min Capacity and Max Capacity values in Auto Scaling Group so that when any instances failed, the auto scaling group can automatically recover a new instance for the failed server.

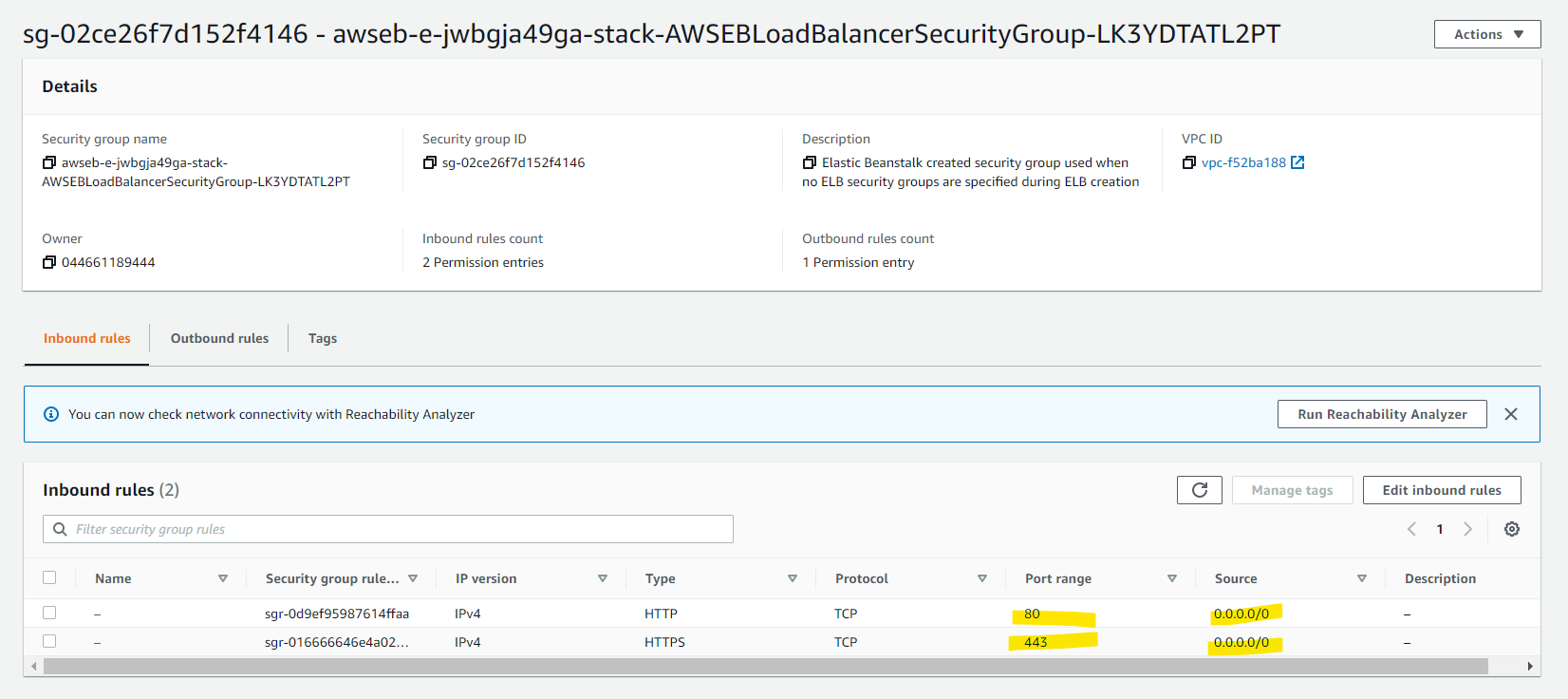


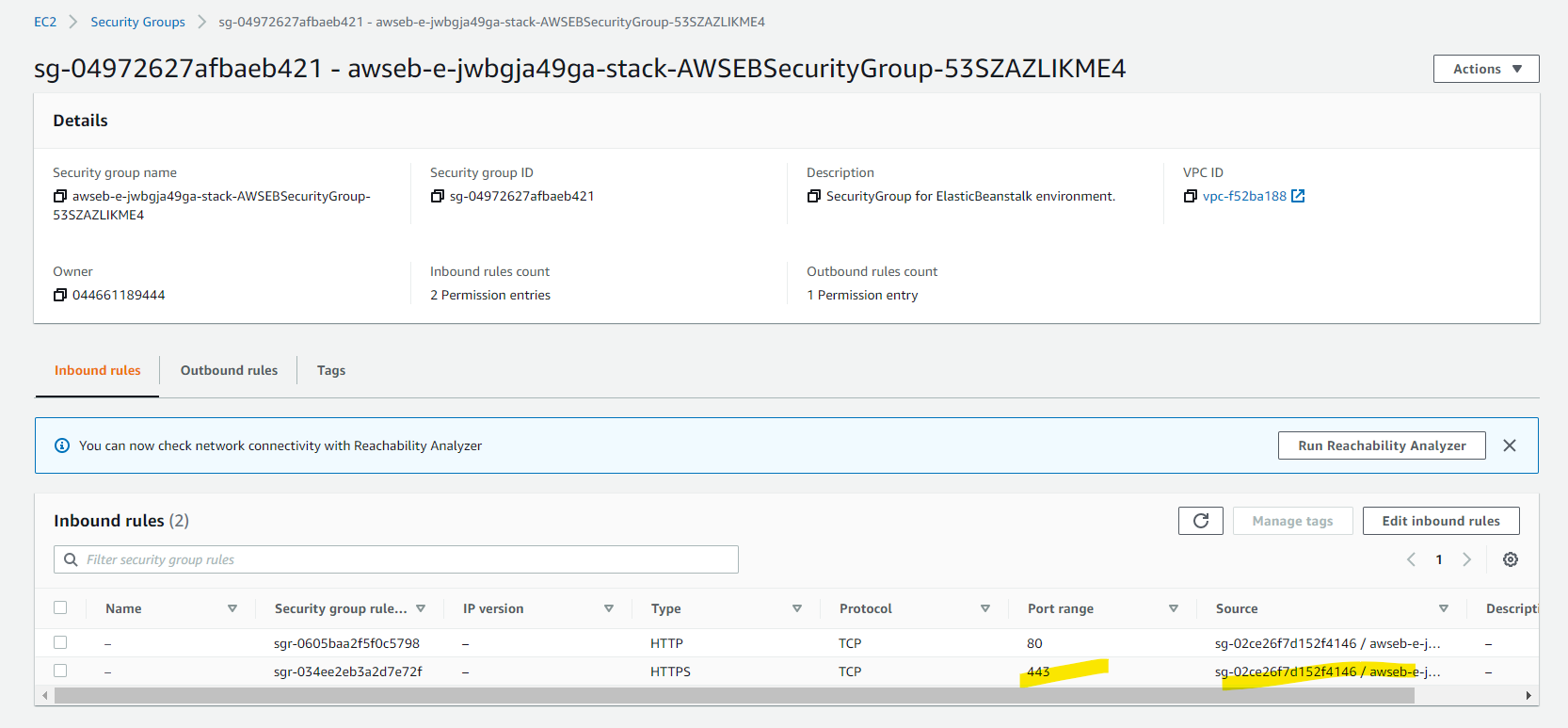
* Since the Policy API need to access other services, like AWS Secretes Manager service etc. We may need to create a new Role to replace the built in Elastic Beanstalk Service role. The new role shall have additional permission to access AWS Secrets Manager and other necessary AWS services, like Amazon MQ etc.



* Also we need to update the auto generated Security Groups, for example add the new rule for the SSL feature in Load Balancer (also need to install the SSL certificate in Load Balancer etc., otherwise the SSL setting will not work):

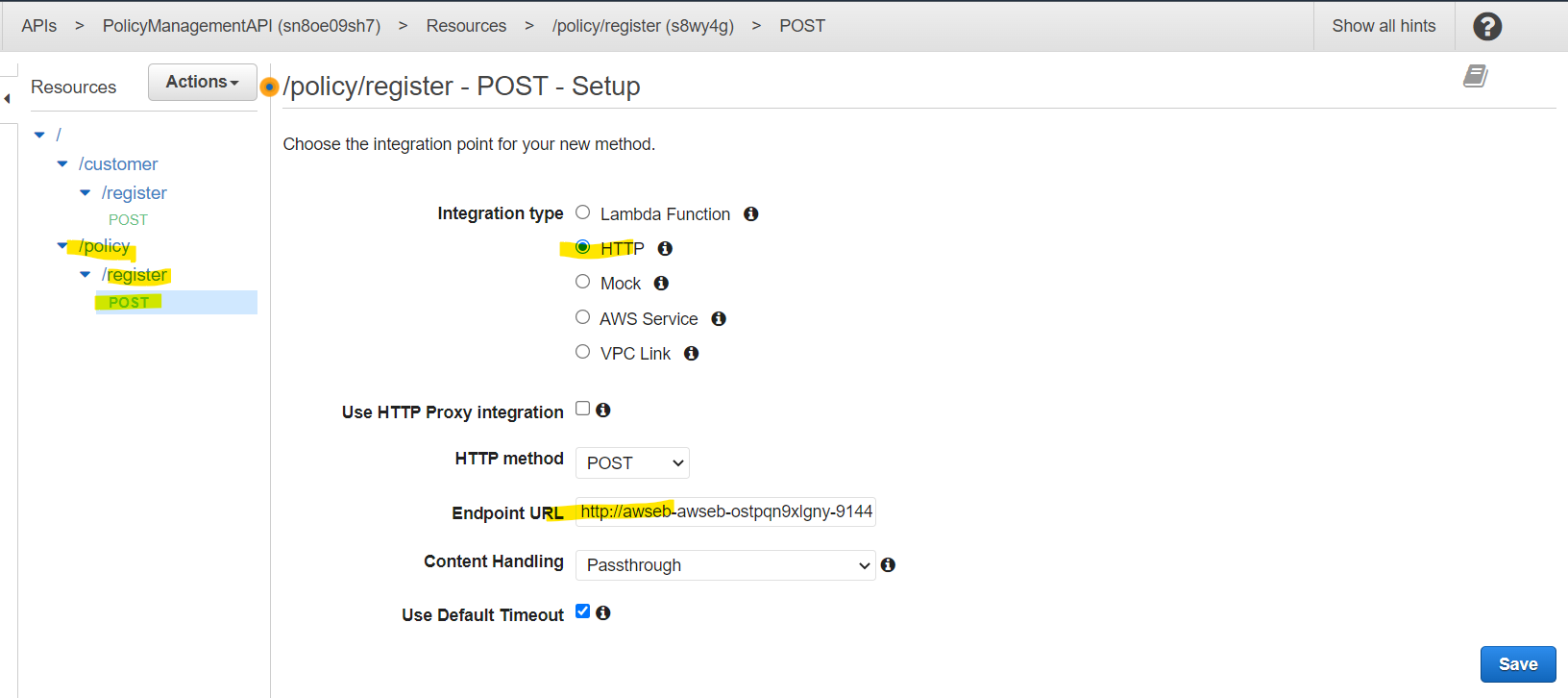






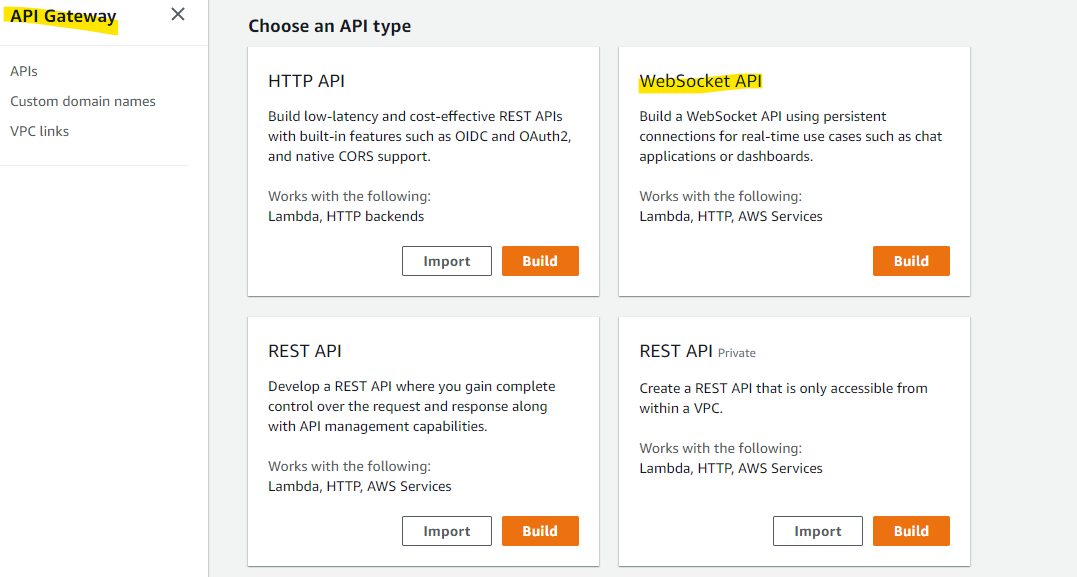
* Also can create the new Policy Register post method in below. The AWS API Gateway will replace the Ocelot gateway in the on premise implementation.

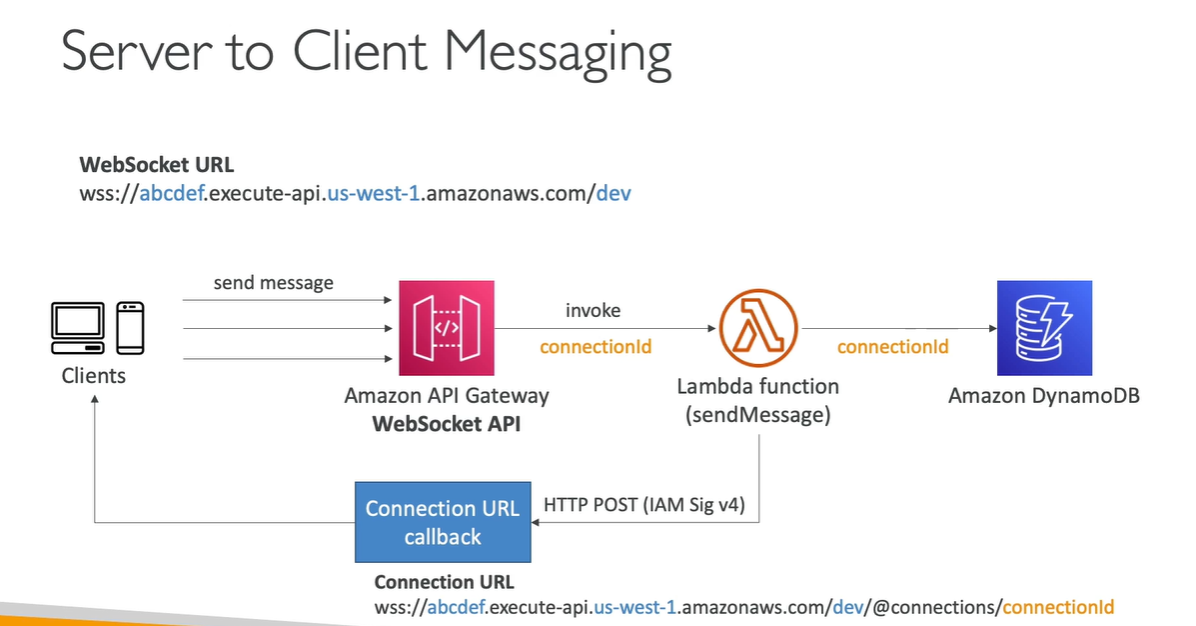
For example, create a new Resource and GET/POST method and Gateway endpoint will point to the Elastic Beanstalk’s Load balancer endpoint:



1. Policy Management API - AWS Elastic Beanstalk + AWS DocumentDB + AWS MQ Design
   1. The Policy Management API will work as a glue API between front end app (Angular app) and the backend Service Api (Ploicy API). The Policy Management API also implements the WebSocket feature to build the persistent connection between Angular client app and the server side app.
   2. Current Policy Management API uses SignalR to implement the persistent communication between Client and Server. We may modify the project to remove the SignalR feature and use AWS Websocket type API Gateway to implement the persistent communication as below diagram.

But this modification needs more changes in the current implementation, so in this design, we use another approach descripted in below item 4.3:





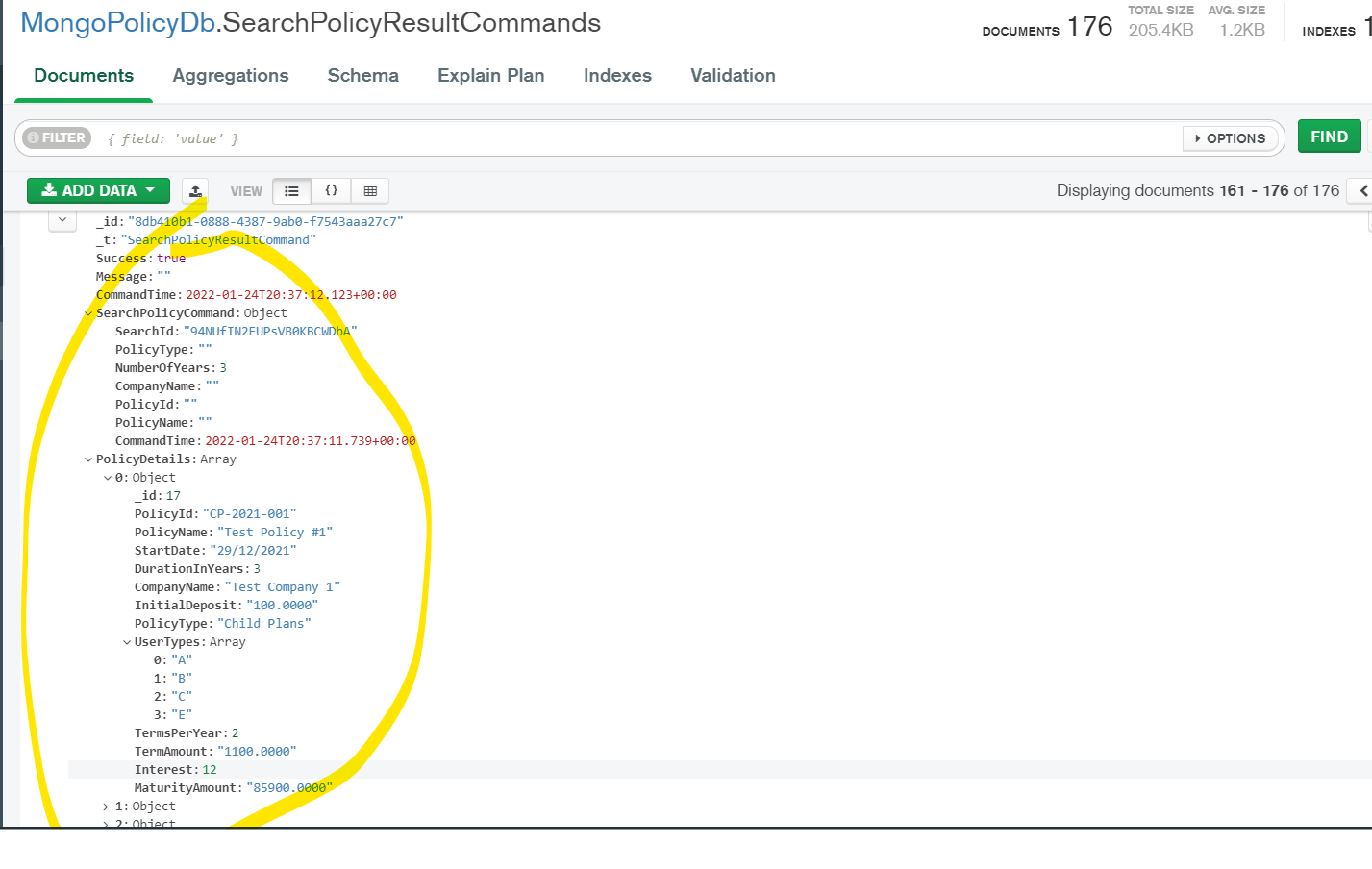
* 1. Another approach is keeping the SignalR feature, instead of deploying to AWS WebSocket API Gateway. We will deploy the modified project to AWS Elastic Beanstalk.

In this approach we still need to modify the code to use Amazon MQ as Policy API’s approach so that both Policy API and Policy Management API can work together in the AWS Platform by Amazon MQ (as they do in on premise by Rabbit MQ).

* 1. Since Policy Management API use MongoDB to store the Policy Search results for the future Audition/analysis etc. The Policy MongoDB is NOSQL document DB and has below complicated data structure.

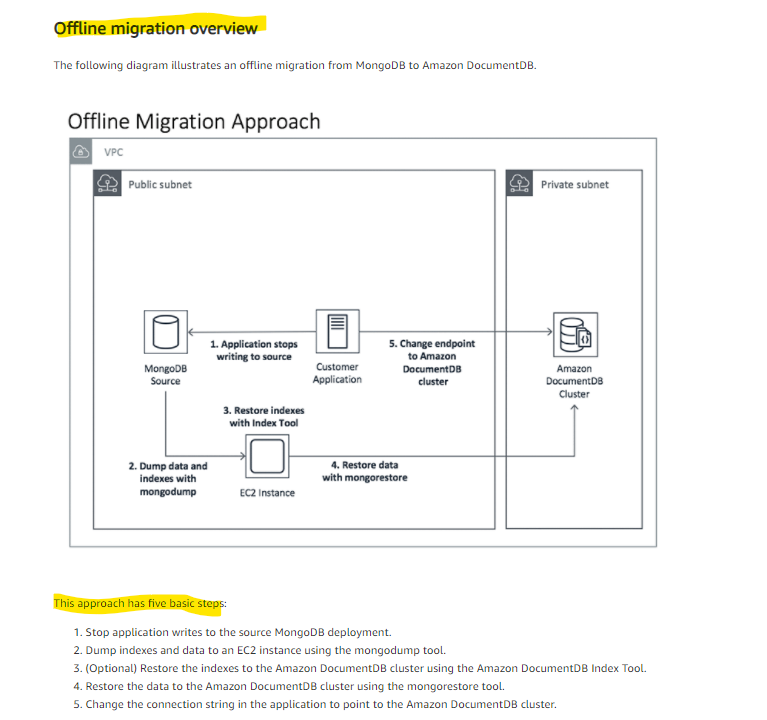
In order to have minimum code change, it is better no to use the AWS DynamoDB since it is key-value format NOSQL DB (which requires more code changes in current project to use DynamoDB).

In this design, we will use Amazon Document DB since it is compatible with MongoDB to reduce the code changes in current project.



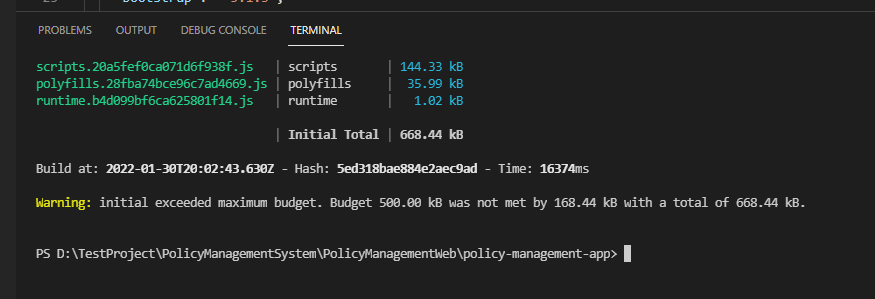
* 1. We can use below suggested Offline migration method using in the development phase’s DB migration from MongoDB to Amazon DocumentDB (with MongoDB compatibility):

<https://aws.amazon.com/blogs/database/migrate-from-mongodb-to-amazon-documentdb-using-the-offline-method/>

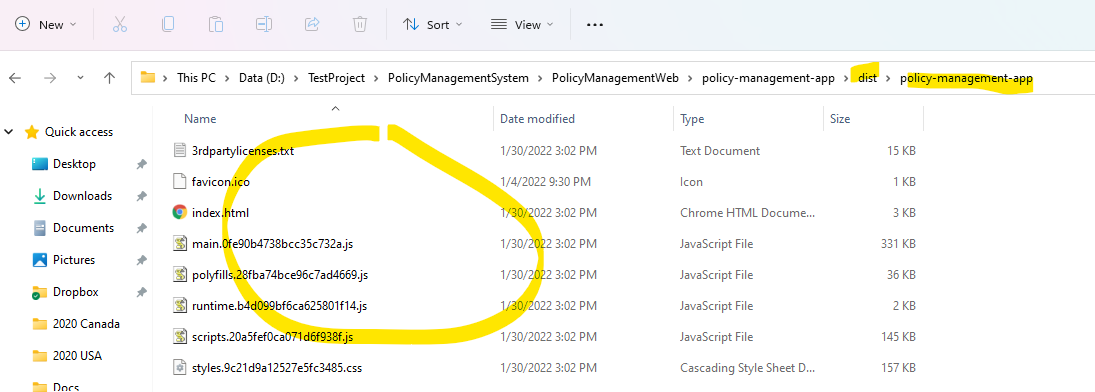


* 1. After above code and DB migration are done, we can deploy the Policy Management API project to AWS Elastic Beanstalk service as Policy API project does. Please refer to above Policy API Elastic Beanstalk deployment process for details.

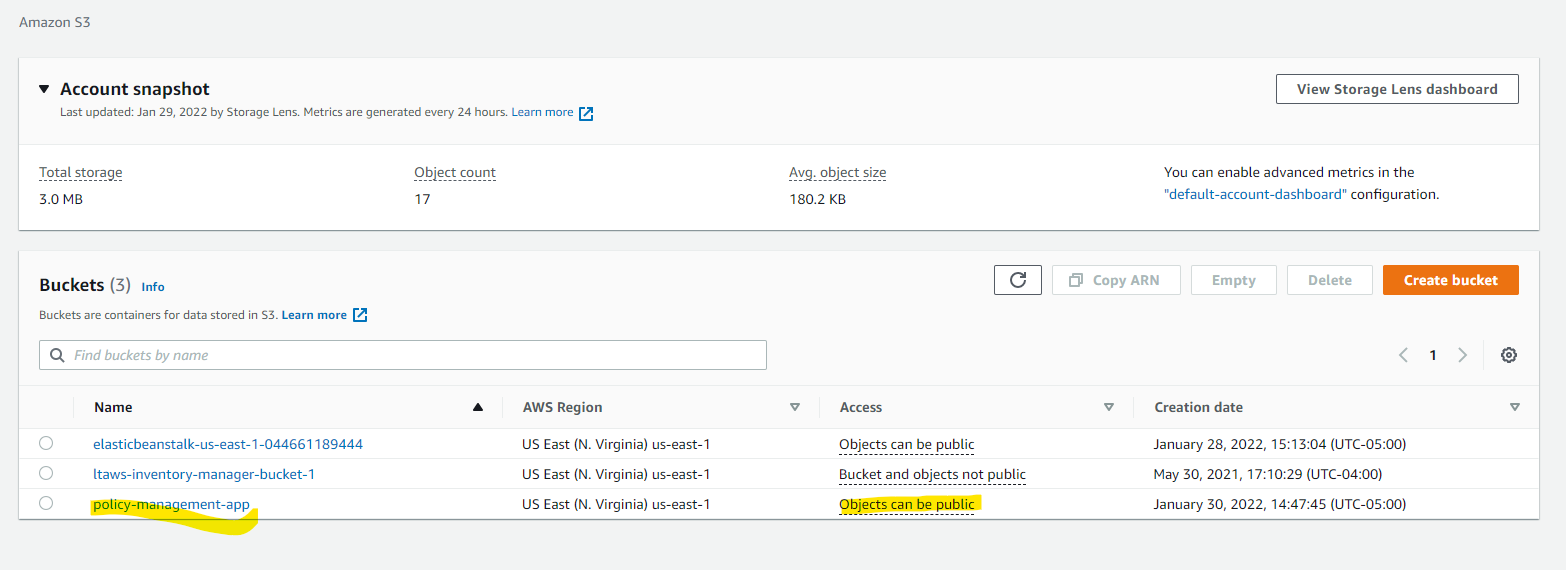
1. Front end Angular App - AWS S3 + Static Web Site Design
   1. The front end web app is created by Angular technology. It can be deployed as static web site in AWS S3 service.
   2. First we can run “ng build” command in Visual Studio Code to build the Angular app as below:



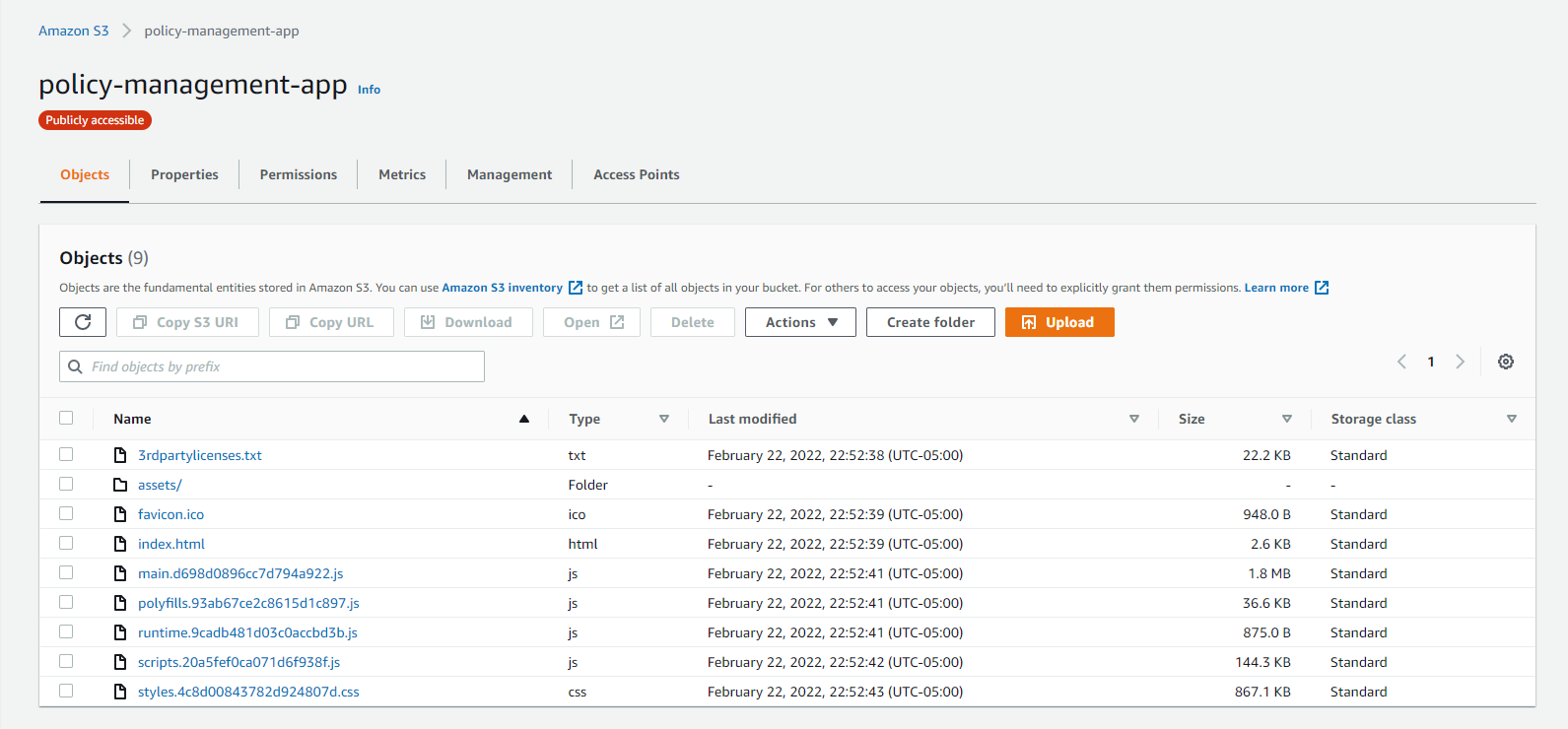
The build command will build the app into below folder:



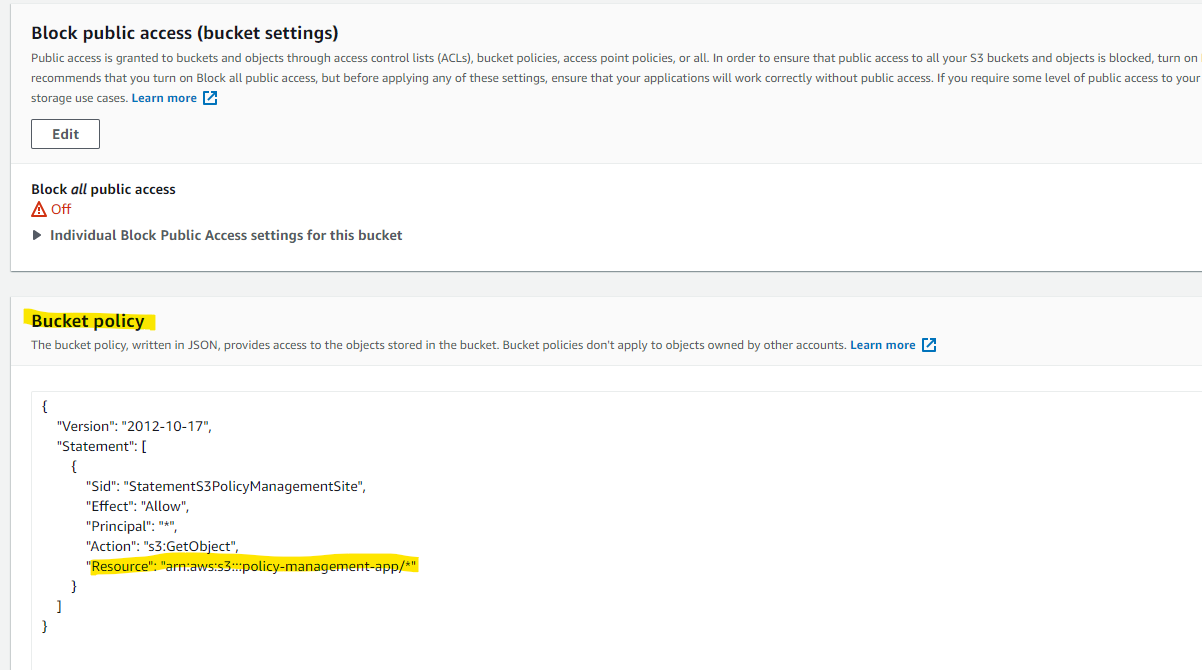
* 1. And then we can create a new S3 storage to store the static site contents, the bucket shall be public accessible:



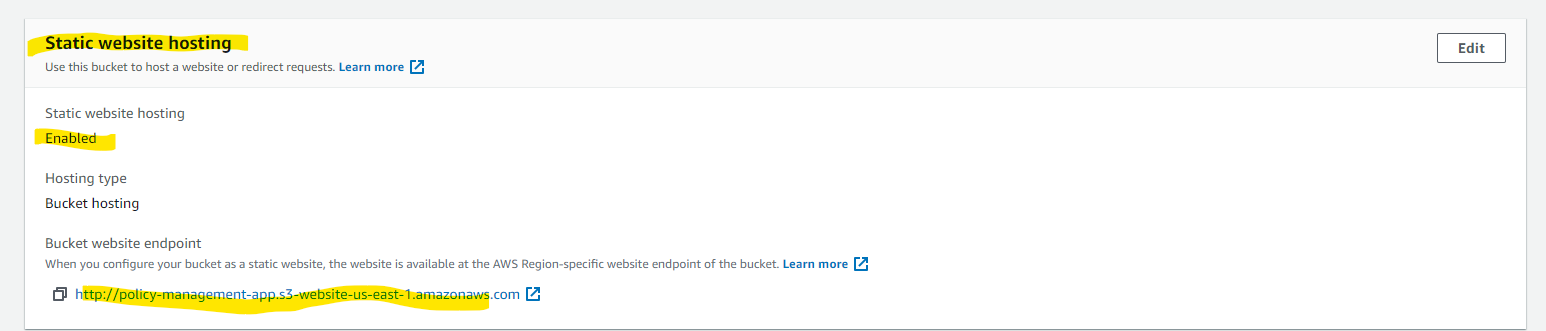
* 1. Upload the built angular app to the bucket:



* 1. Update bucket policy so that anonymous users can access the bucket contents:



* 1. Update the bucket as static site:



* 1. For local development testing, it may throw CORS error as below:

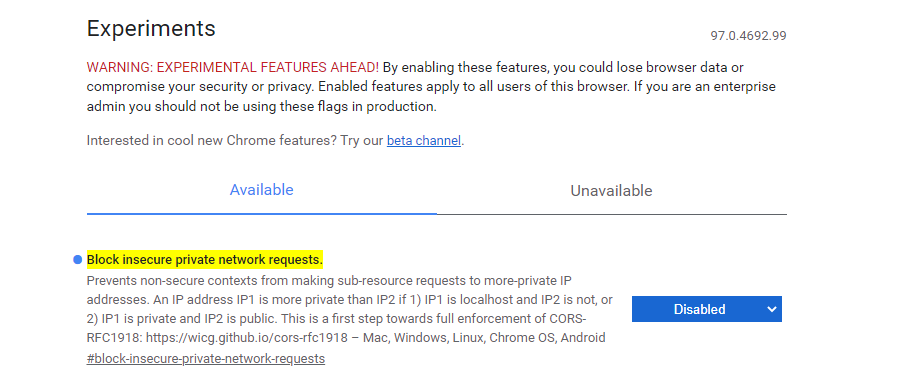
The request client is not a secure context and the resource is in more-private address space `local`.

We can solve the issue in Chrome by below steps:

1: go to chrome://flags/#block-insecure-private-network-requests

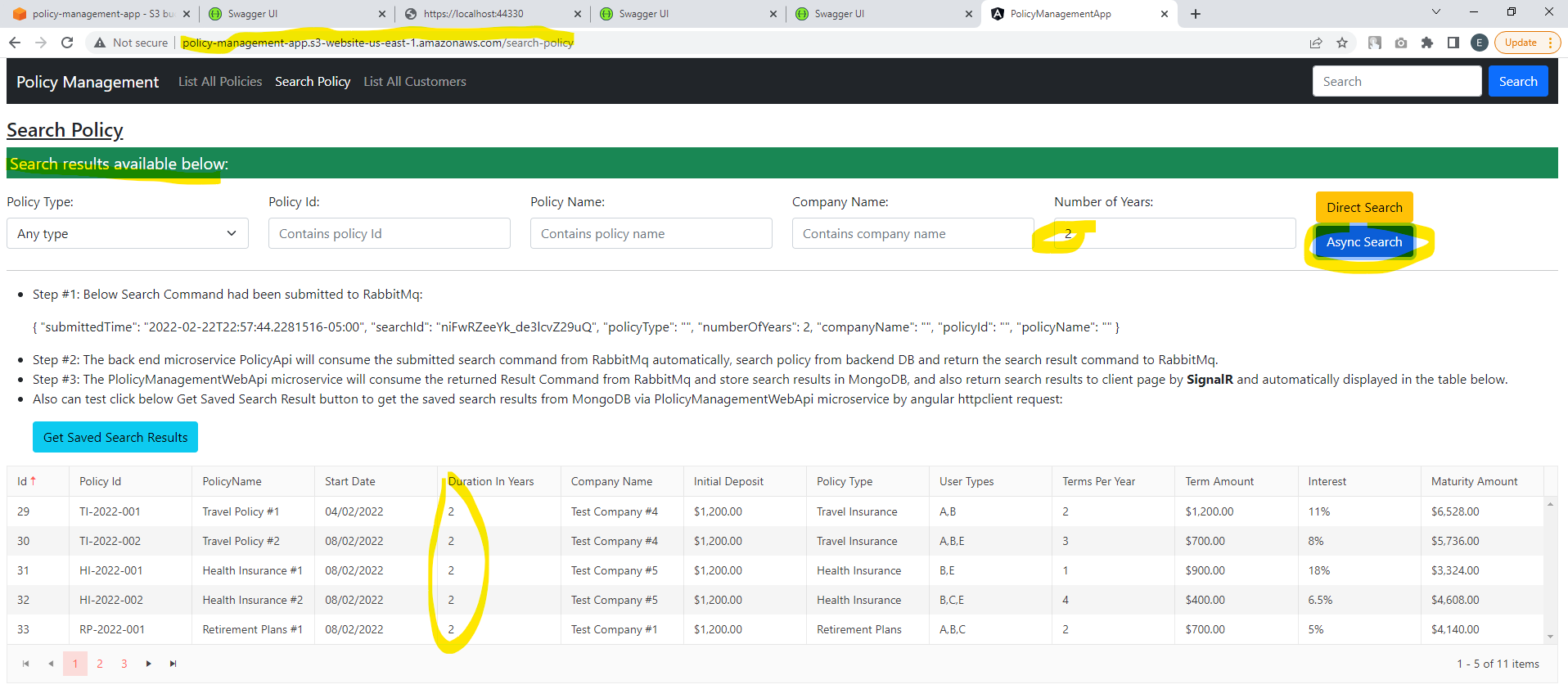
2: set Block insecure private network requests to Disabled

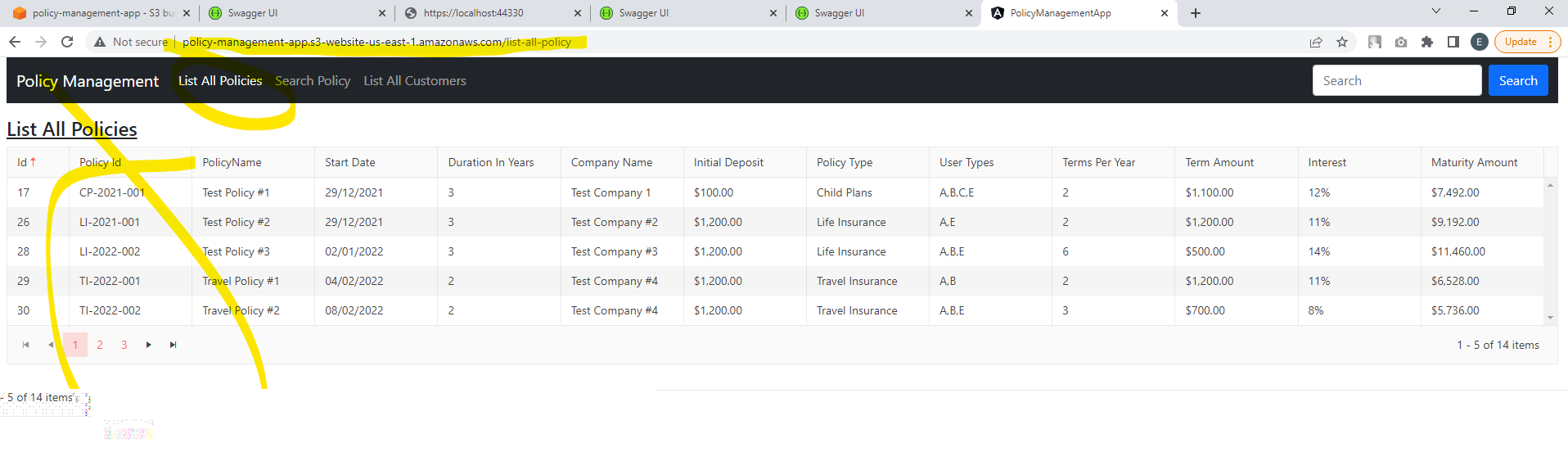
Note: this just works fine when you're in your own computer or your dev environment



* 1. Now access the Angular app site we can see the AWS Static site and find it can work properly with local Micro services:

<http://policy-management-app.s3-website-us-east-1.amazonaws.com>



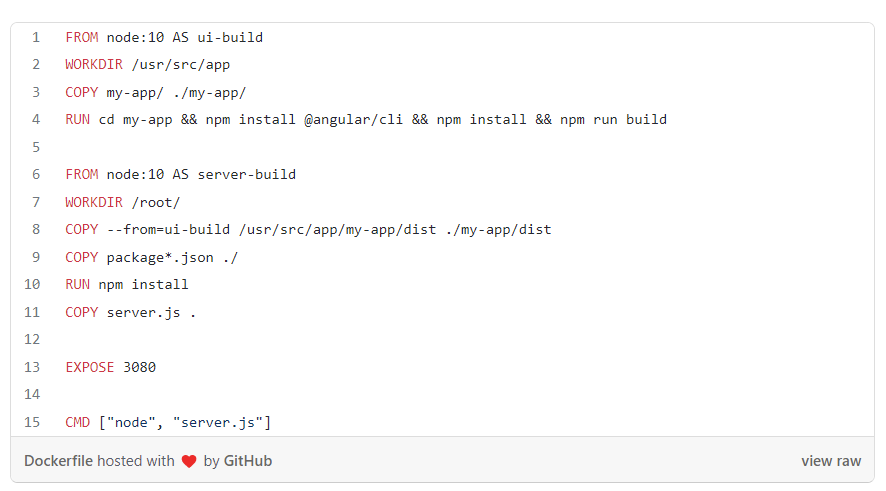


Notes: The AWS S3 static site only works with local development machine since it still access the local host services. After all services migrating to AWS in implementation phase, and updating Angular app settings, it shall work properly with those services in AWS Cloud Platform.

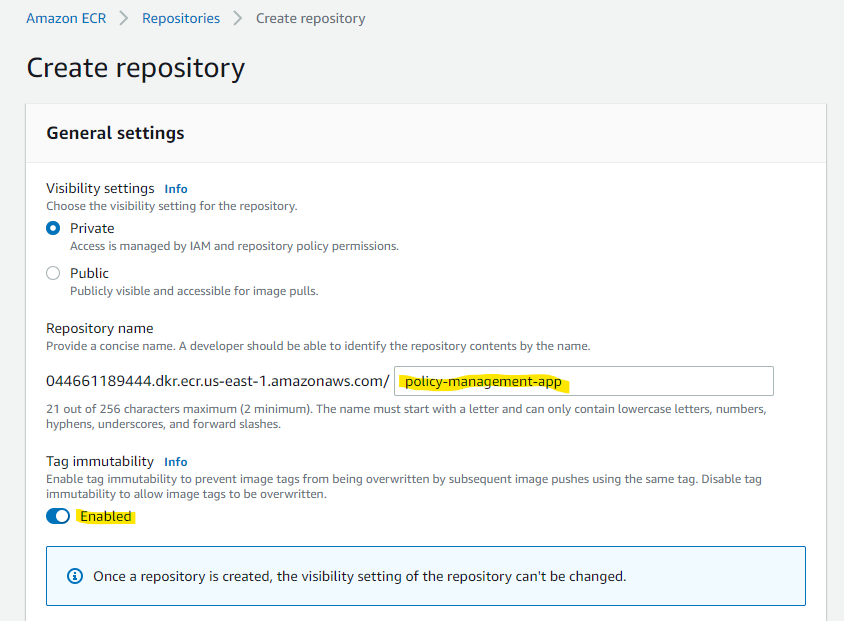
* 1. Also we can deploy the Angular app to AWS ECS with below steps:
     1. Dcokerize the angular app with Dockerfile, for example:

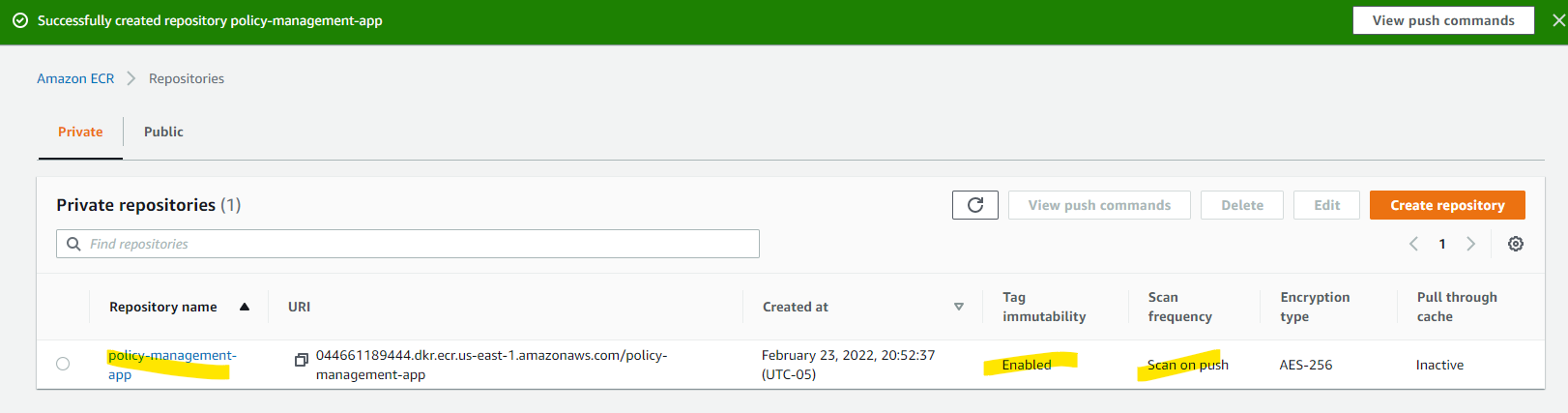
// Build Docker image with below Docker file

docker build –t policy-management-app .

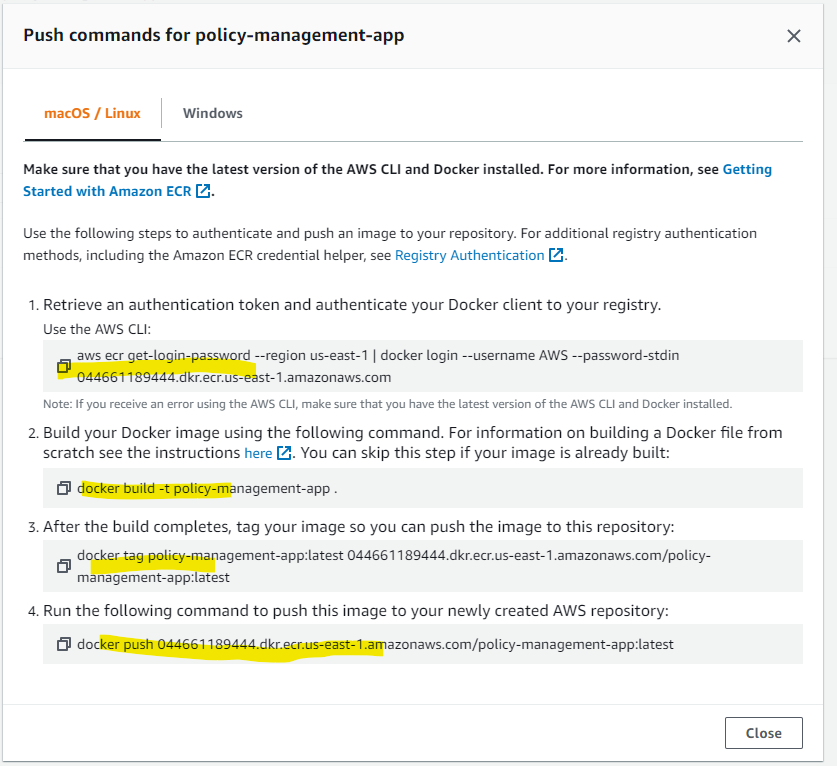


* + 1. Push the Docker image file to AWS ECR (Docker Hub), for example we can create policy-management-app repository as below, make sure enable tag immutability to prevent image tags from being overwritten by subsequent image pushes using the same tag:



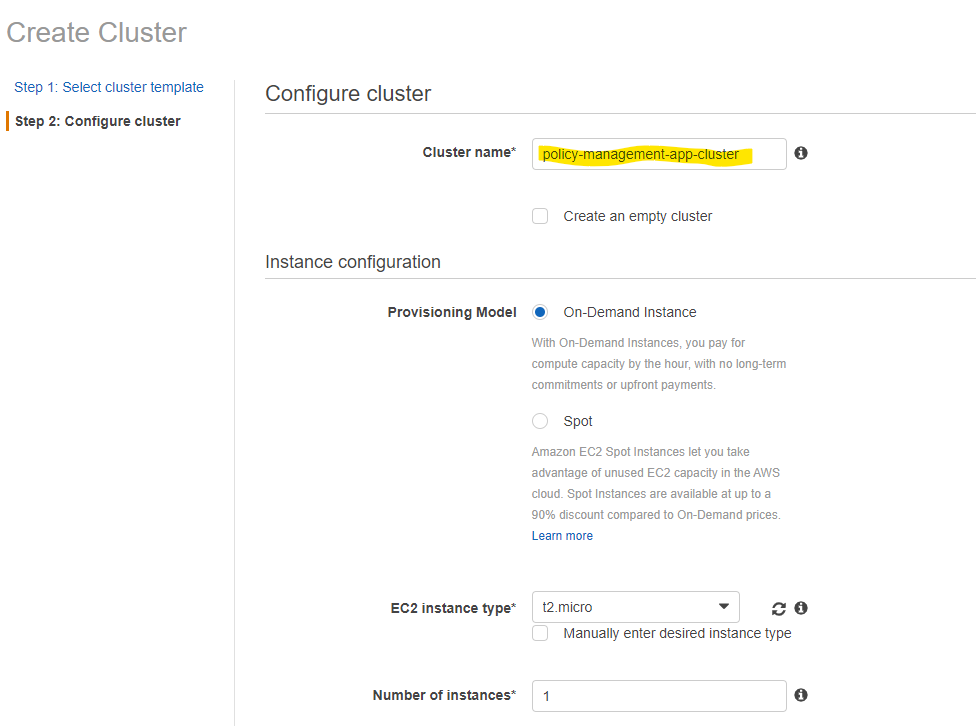


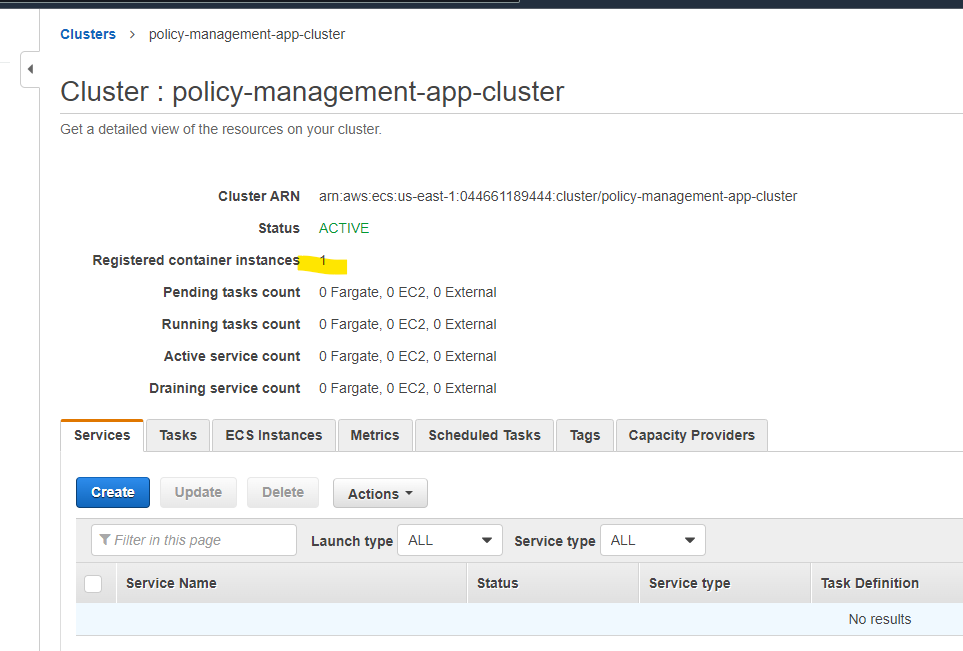
* + 1. Tagging local Docker image and pushing to AWS ECR. For example, can follow below commands to push the image to ECR:



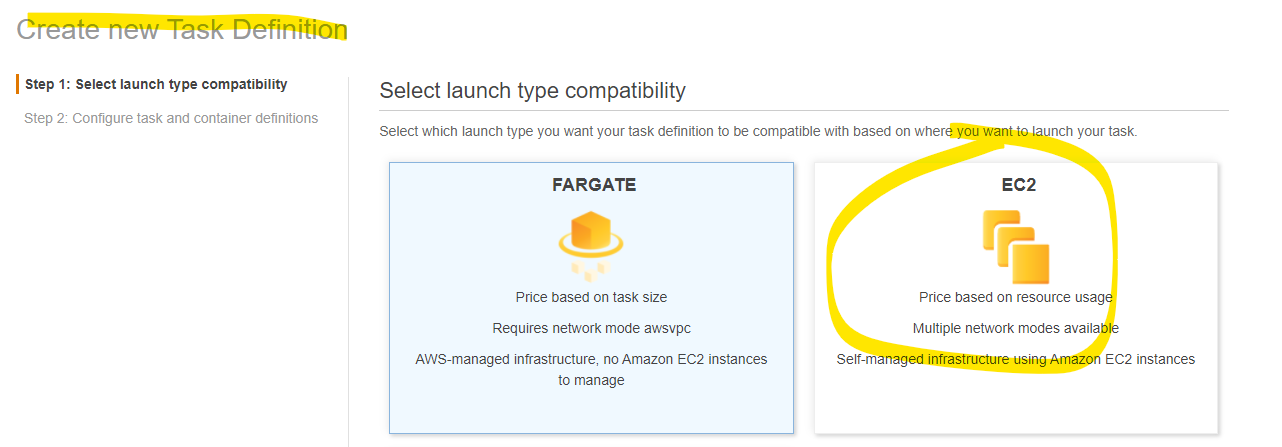
* + 1. And then need to create ECS Cluster, for example:



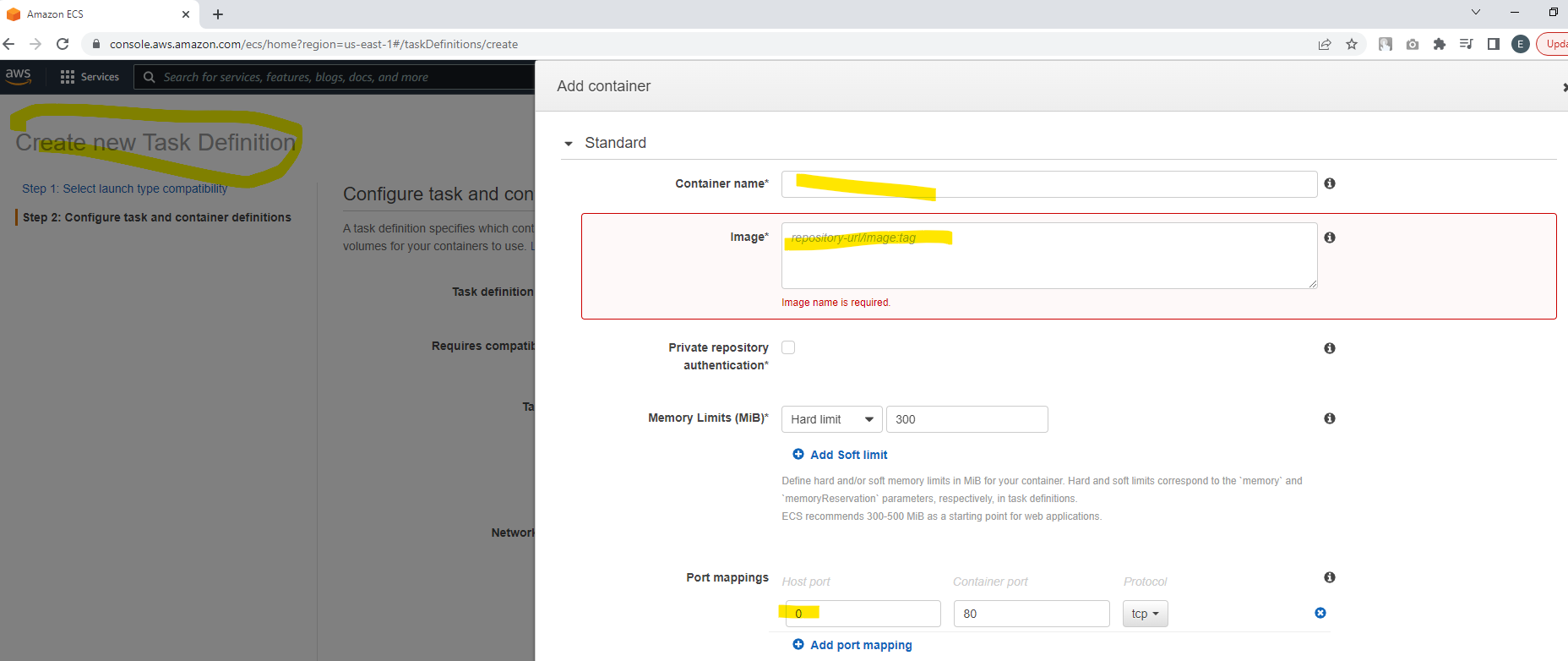




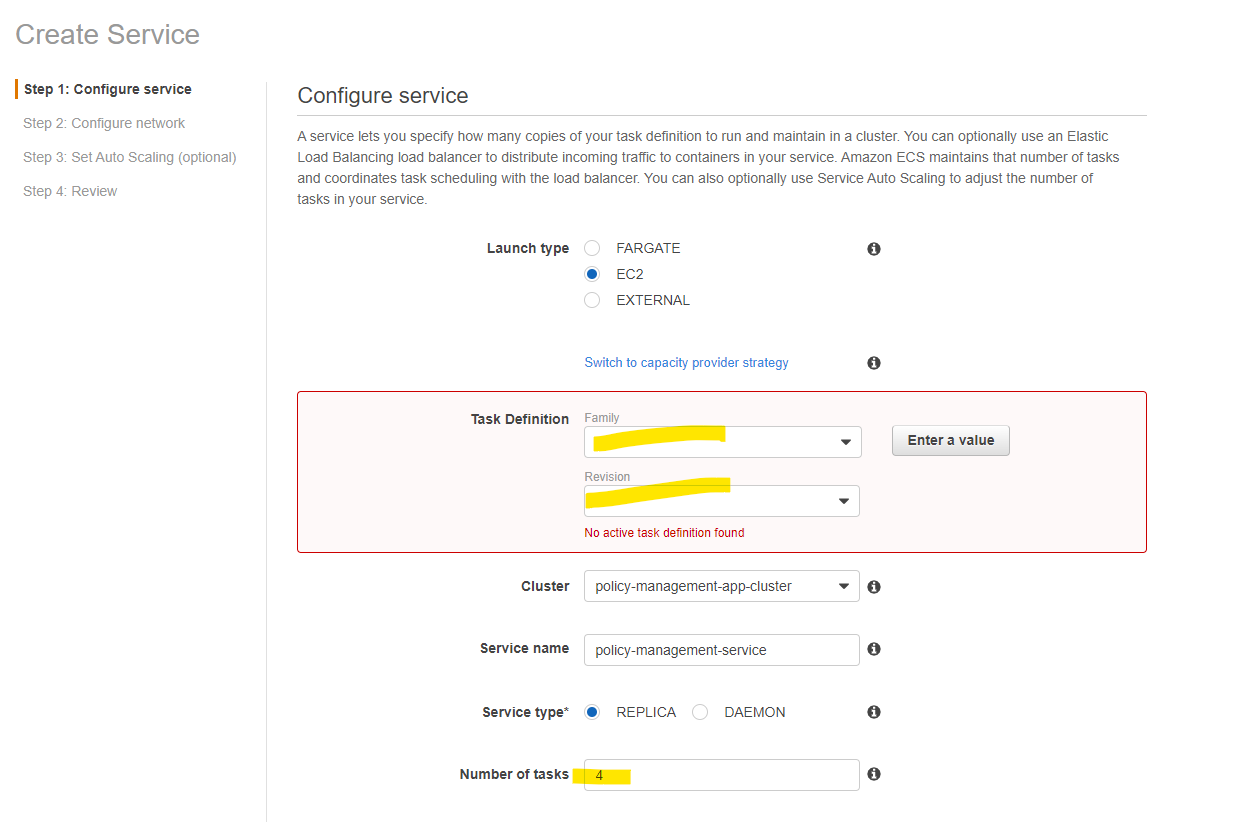
* + 1. Need to create Task that can load the policy-management-app image from ECR, for example:



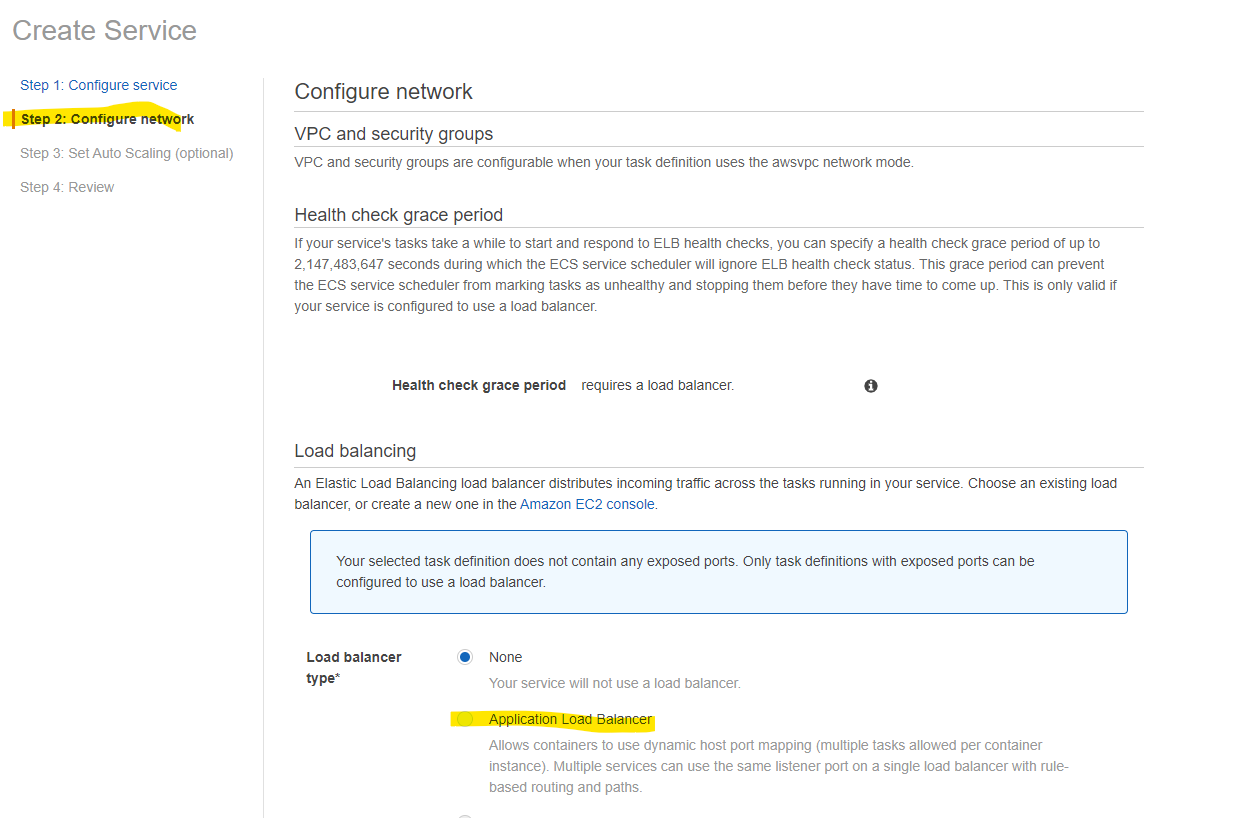
In Add container popup window, shall put the policy-management-app image from ECR, also set the Port mapping to be 0 so that we can run multiple tasks in one instance.



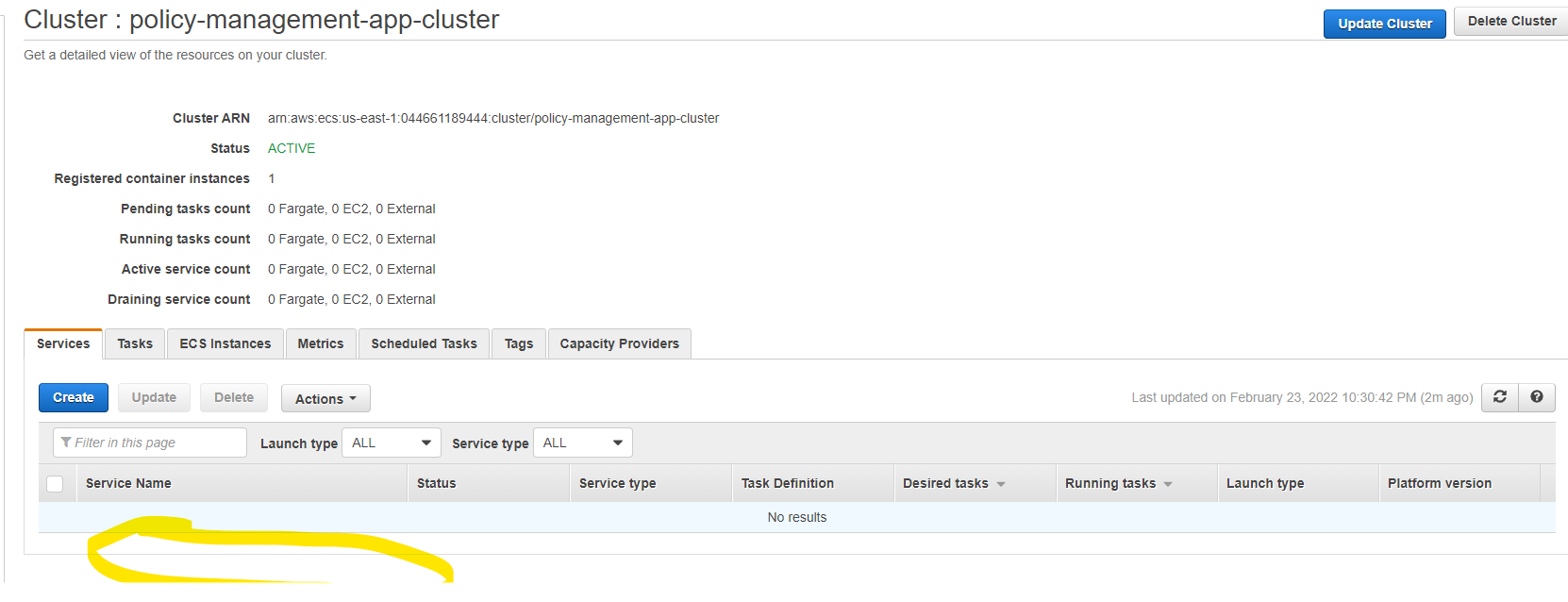
* + 1. After task create, we need to create the ECS service to load the task to ECS Cluster, for example:



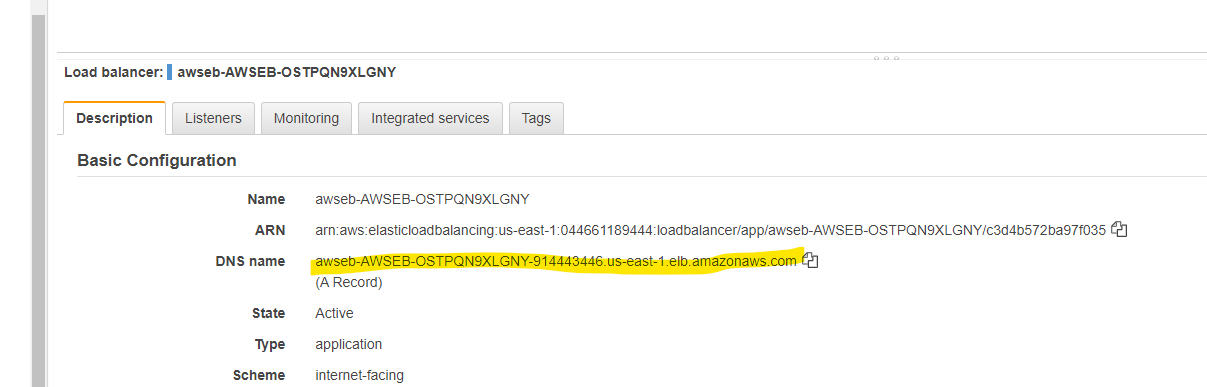
In the second step, make sure select the Application Load Balancer so that it can route the traffic to dynamical port of multiple tasks. Also make sure the ECS Cluster’s instance security group can open all ports from the security group of the Application Load Balancer.



* + 1. After ECS Service success deployed, then we can check service status as below, for example:



* + 1. And then we can test the deployment of Policy Management App from the Application Load Balancer:



1. Continue to Implementation phase…