Movie API

Swagger Doc:
https://wn34wdzpxpuvvpsb5e5xkirycu0jgtqs.lambda-url.us-east-1.on.aws/swagger/index.html

Front End Application:
https://main.d1h137i74nxuca.amplifyapp.com/movies

Application Design

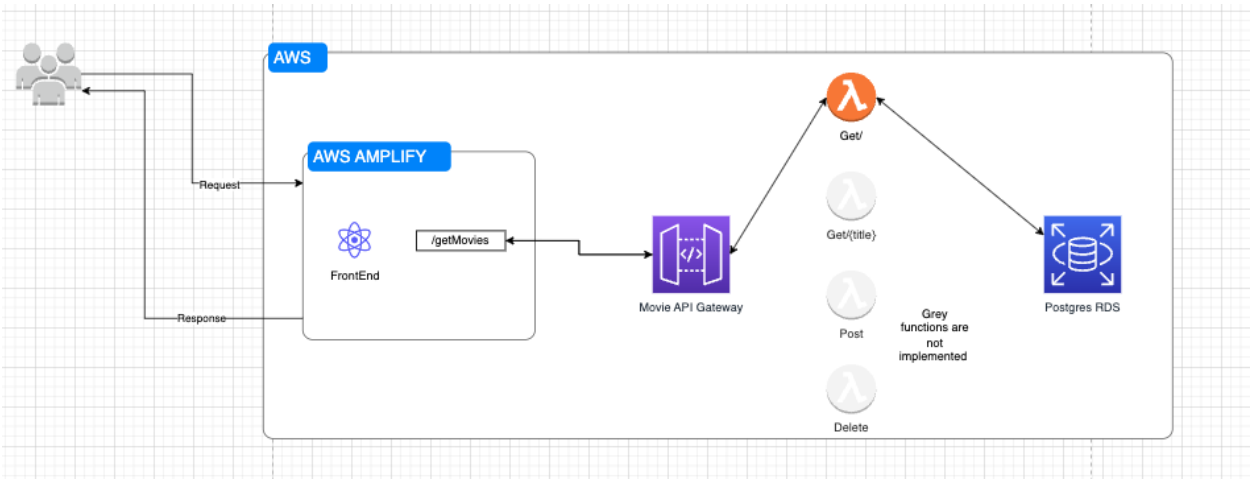
The solution for “MovieApi” consists of a front end application developed in ReactJs using the MUI component library and a backend api service developed using Asp.net core.

Cloud Infrastructure

I adopted a serverless strategy for the cloud infrastructure implementation. Additionally, I've outlined an alternative approach that I would consider as well. The front end application is deployed and hosted on AWS Amplify stack , the back end api has been deployed to AWS Lambda and I have used a Postgres rds instance as my datastore.

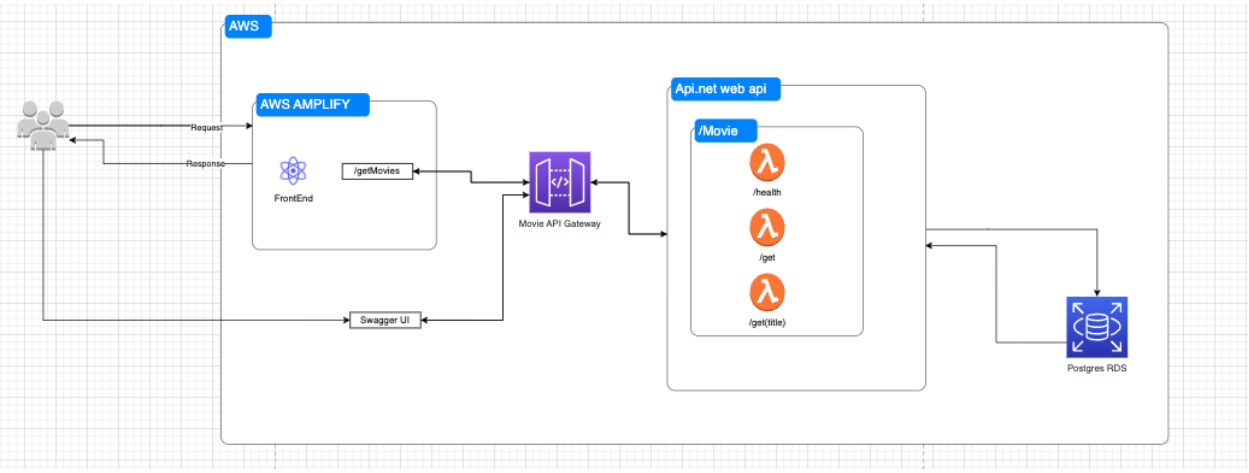
For the backend I have 3 approaches that could be used and I have documented them below

1. Single Lambda Function for each piece of functionality that is access via AWS API GATEWAY



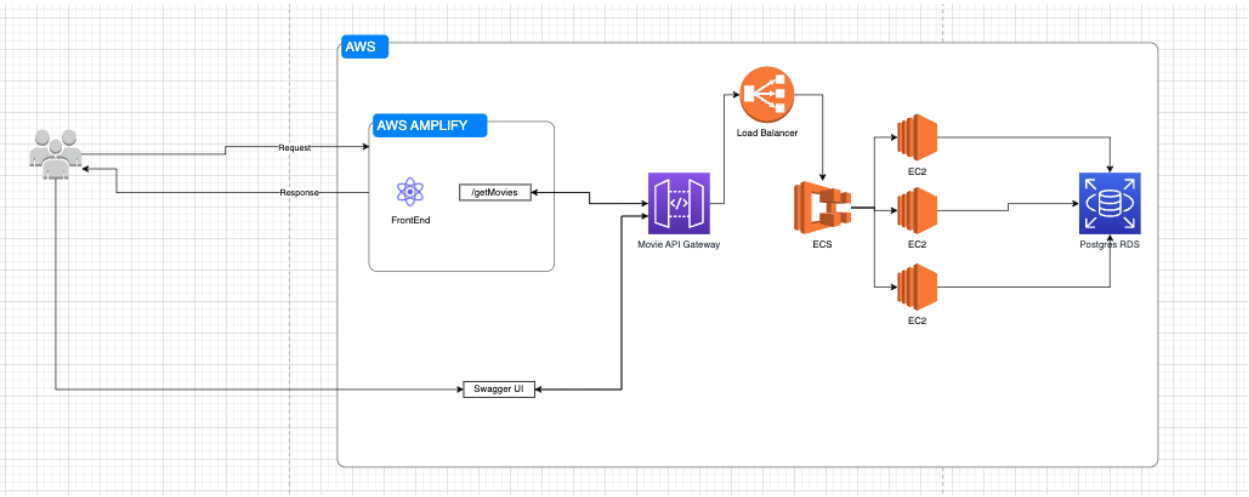
   a. In this approach a lambda function is created for each piece of functionality needed and is routed via the api gateway

2. Asp.net Core Web Api deployed on AWS Lambda that is access via AWS API GATEWAY



   a. In this approach a ASP.NET core web api is deployed through lambda via the Amazon.lambda.AspNetCoreServer NuGetPackage

3. Api.net Core Web Api deployed using Aws ECS and access via the AWS API GATEWAY



   a. In this approach the ASP.NET core web api is deployed via ECS

Among the three approaches available, I've opted for the second one as it aligns most effectively with the requirements and offers a completely managed serverless architecture.

Pros of this approach :

- Event-Driven Scaling
- Cost Efficiency
- No Server Management
- High Availability
- Quick Development and Deployment

- Auto-Scaling and Load Balancing
- Stateless Execution
- Reduced Operational Overhead
- Global Reach

Cons of this approach :

- Limited Execution Duration
- Cold Starts
- Stateless Nature
- Resource Limitations

How would I improve the solution, code base and what could have been done better

1. Select a runtime and programming language that excel in efficiently fetching data from a database / optimize the current .net solution using provisioned concurrency , lightweight ORMs like dapper and connection pooling for the db connection.
2. Include unit and integration tests via the use of test containers for the asp.net web api backend .
3. Organize the database optimally, establishing meaningful relationships between tables .
4. Create full CRUD functionality .
5. Increase logging for additional observability .