**SENG3011 Final Report**

**Y2**

Table of Contents

[Introduction 3](#_Toc70259328)

[Scope 3](#_Toc70259329)

[Use Cases 4](#_Toc70259330)

[Requirements 4](#_Toc70259331)

[Functional requirements 4](#_Toc70259332)

[Performance Requirements 4](#_Toc70259333)

[Interface requirements 4](#_Toc70259334)

[System Design 5](#_Toc70259335)

[Conceptual Design 5](#_Toc70259336)

[System Architecture 6](#_Toc70259337)

[Detailed Design 6](#_Toc70259338)

[External Apis 13](#_Toc70259339)

[Achievements 13](#_Toc70259340)

[Team Organisation 15](#_Toc70259341)

[Workload acknowledgment 15](#_Toc70259342)

[Project Review 15](#_Toc70259343)

# Introduction

## Scope

An introduction of the document and why this is here, the scope of the document. This document template is the starting point of your submission. Add or delete any sections and heading as required. However, there are specifically some headings that must be included for the actual submission. (as per indicated)

Herein this version, the document has been additionally formatted to be presentable to client. It was first exported from confluence followed by a formatting of the document such as indentation, page numbers, authorship etc.

# Use Cases

Focus on what has been achieved

# Requirements

## Functional requirements

These are the functional requirements for your subsystem in order for the solution to be considered as a “solution”. The solution must achieve these functional aims. You may include herein subheadings such as requirements for MVP or version 1 or 2 …. of the product. This list of functional requirements should be labeled and listed with numbers so it may be referred to in the subsequent sections of the document. Such as in the testing procedure.

## Performance Requirements

## Interface requirements

This section contains all the interface requirements of your subsystem. You should be able to document all the constraints that are interrelated to other subsystems and hence be able to know if your subsystem is meeting these requirements.

# System Design

## Conceptual Design

The API will be developed using GraphQL as opposed to an alternative such as REST. To expose this as a Web service, we will use AWS AppSync as a managed GraphQL service. GraphQL naturally fits the specifications requirements, given the unstructured nature of the various data sources. The data format for how things like articles and reports are specified in the specification, makes defining a GraphQL schema simple. From this we can expect to receive the following benefits from using GraphQL:

* Improved performance
  + allows you to write rich specific queries that only fetch the fields that you want. This eliminates over-fetching or under-fetching problems thereby reducing network overhead. Overall ‘chatter’ between endpoints is reduced since only one endpoint - the GraphQL server - is hit initially to resolve a query. As such multiple requests do not have to be made to multiple endpoints.
* Reduced complexity in front-end development
  + the development of the front-end is optimised as front-end code does not have to fetch data from multiple endpoints, but can rather write a rich SQL-like query that will fetch all necessary data in one request. Unlike REST, a new endpoint is not required for each different type of ‘view’ required by the front-end. Moreover, we’d like to maximise the scalability, performance and fault-tolerance of the GraphQL API and have therefore decided to combine with a micro-services architecture that uses server- less compute. Micro-services decouples services, improves scalability and performance, and in this case is reduced in complexity due to the use of a GraphQL schema.

The API back-end (e.g resolver functions, schema) will be written in Golang, using the - library for GraphQL. Golang is a minimalist, garbage collected yet performant, strongly and statically typed compiled programming language. For our purposes, we favour the performance, ease of development and strong type system which reduces compile time errors.

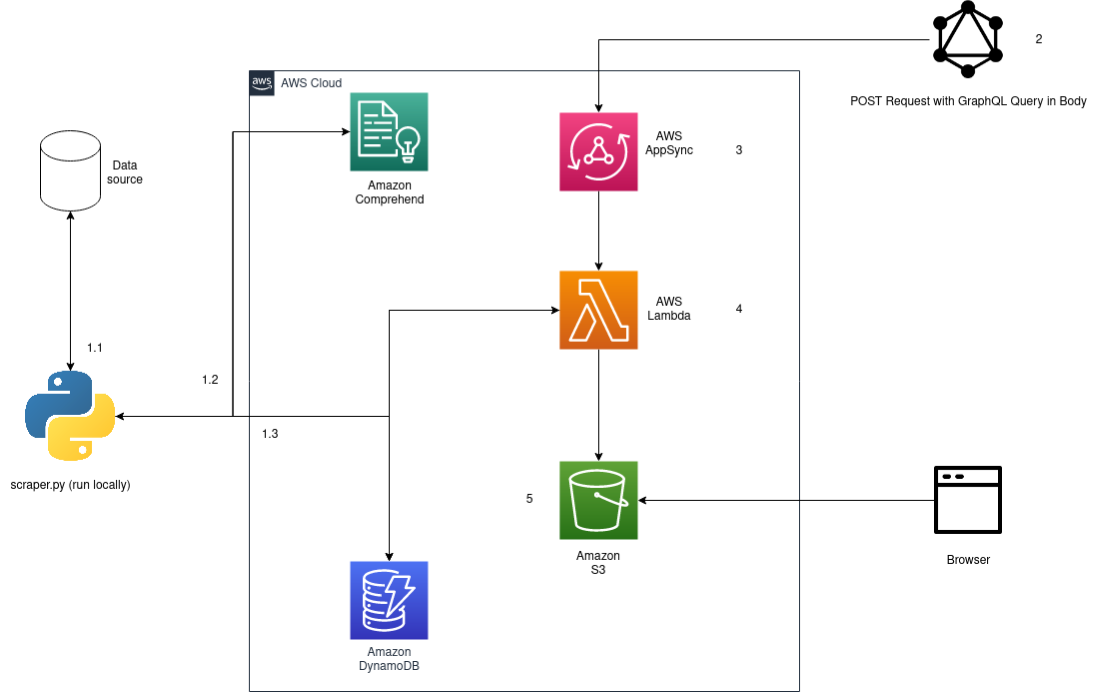
To assist the API in resolving data, we will use the fully managed proprietary NoSQL database service Amazon DynamoDB; the unstructured nature of much of the data makes it unsuitable for a mapping to a relational database.

The application will be deployed as a cloud-native with AWS, we will be using AWS Code-pipeline for our build process. From a broader DevOps point of view, we will be using Trello to manage user-stories, tasks and features in a simple scrum board for each sprint.

For our deployment environment, we have chosen to opt for server-less compute to integrate seamlessly with our micro-services architecture. Given that we are AWS-native, we will use AWS Lambda, effectively abstracting away all server/compute considerations as each service will run as a Lambda function running Golang functions.

## System Architecture

The API utilises a broad range of managed services on Amazon Web Services to build an endpoint which can be queried after the scraper program is run. The flow can be summarised as such:



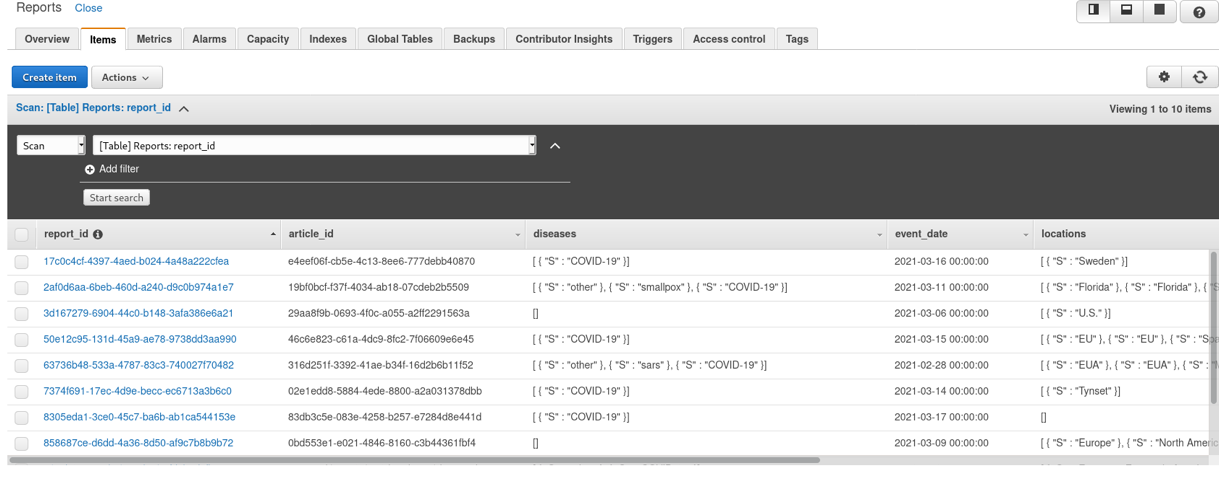
## Detailed Design

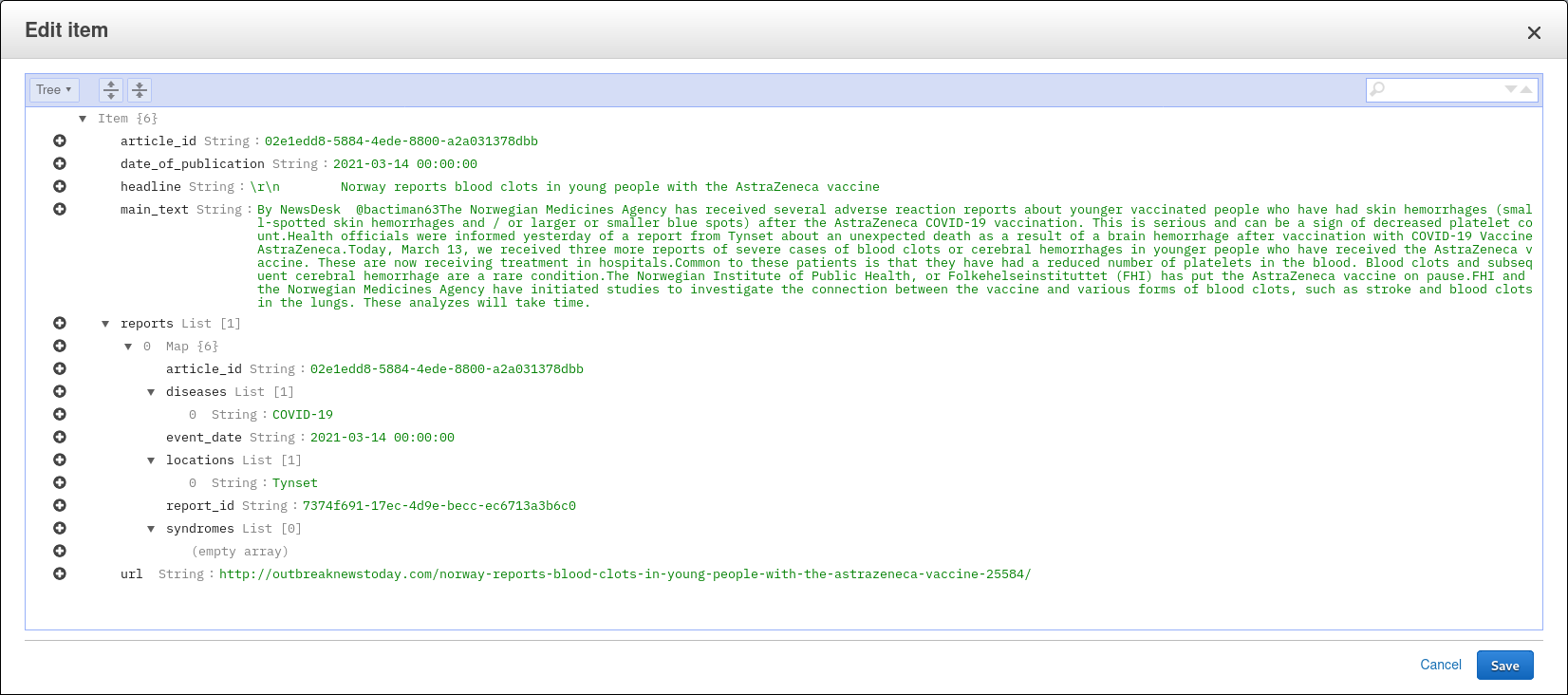
The initial flow as depicted in the System Architecture figure (above) details how the database is initialised to be queried by the API.

Run scraper.py. The scraper sits in its own AWS Lambda function with a CloudWatch Events Rate Expression such as rate (1 day) which invokes the Lambda function every day. The scraper scrapes the given data source of <http://outbreaknewstoday.com/> using the Beautiful Soup library, article objects are then generated as python dictionaries in the desired format which fields such as date\_of\_publication, headline etc. In addition to these fields we also add an article-id which is uniquely generated uuid from a uuid4() call. Reported are stored as children objects under the key reports in an article, with each report containing it’s own unique report-id as well as the article-id of it’s parent article.

The scraper then begins to intelligently generate disease reports. This involves querying AWS’s managed Natural Language Processing Service Amazon Comprehend. This an all programmatic access to AWS resources is done through the AWS python SDK boto3. The API call to the DetectEntitites endpoint send the main\_text of a given article in the request, and returns a list of classified entities sorted by type and score that the Amaxon Comprehend real-time analysis found in the article. From this fields such as location in a report can be populated.

The article object is now fully populated, complete with children report objects. This is then serialized into json with json.dumps() and then written into the Articles table in DynamoDB., with reports also being written but to a separate Reports table. Examples below:





1. Once the DynamoDB table is populated, a user can now query the API endpoint to make a request. To resolve requests to the endpoint, we have described a single GraphQL query type called getArticles(), which takes in combination of period\_of\_interest, location and key\_terms. The Api only supports date ranges and now singular / exact dates. Periods\_of\_interest is a required parameter.
2. Below is the GraphQL schema defining the various types and queries.

type Articles {

url: string!

headline: String!

main\_text: string!

reports: [Reports]!

article\_id: String!

date\_of\_publication: String!

}

type Mutation {

putArticle(url: String!, headline: String!): Articles

}

type Query {

getArticles(period\_of\_interest: String, key\_terms: String, location: String): Response!

}

type Response {

body: [Articles]

statusCode: Int!

statusMessage: String!

}

type Schema {

query: Query

mutation: Mutation

}

1. PutArticle() is defined as the placeholder since the API has no mutation / write requirements. Across the whole schema ! denotes compulsory for a field or parameter. As seen in the getArticles() definition, none of the input fields are compulsory as parameters, as this logic is handled by the bank-end. This is due to making query parameters required results in poor error messages being sent to the user upon a request where a required parameter is not given. The response type is defined for easy return statements in the Lambda function resolver with the body field being a nested type containing a list of articles. This maps very nicely to python as evidence in the lambda.py source-code where return statements are simply

return {

“body”: results,

“statusCode”: 200,

“statusMessage”: “Query successful.”

}

1. For error responses or responses with no body, body can simply be left empty or not at all returned:

return {

“statusCode”: 200,

“statusMessage”: “No articles matching the given query”

}

This seamless integration of services is extremely powerful and a main advantage of using GraphQL and particularly AWS AppSync. All data from the scraper to the database and finally to the GraphQL schema/ Lambda function are serialized in the exact same way.

All endpoints are POST, with the user inputting the GraphQL query in the body of the request. After specifying which fields it wishes to retrieve whether that is all of them.

query {

getArticles(key\_terms: “COVID-19”, location: “United States”, period\_of\_interest: “2021-01-01 17:00:xx to 2021-03-22 00:00:00”){

statusMessage

body {

article\_id

date\_of\_publication

main\_text

headline

url

reports {

article\_id

event\_date

diseases

locations

report\_id

syndromes

}

}

statusCode

}

}

{

“data”: {

“getArticles”: {

“statusMessage”: “Query successful”,

“body”: [

{

“article\_id”: “316d251f-3392-41ae-b34f-16d2b6b11f52”,

“date\_of\_publication”: “2021-02-28 00:00:00”,

“main-text”: “On Saturday, the U.S Food and Drug Administration issued an emergency use authorization (EUA) for the third vaccine for the prevention of coronavirus disease 2019 (SARS-CoV-2). the EUA allows the Janssen COVID-19 Vaccine to be distributed in the US for use in individuals 18 years of age and older. “The authorization of this vaccine expands the availability of vaccines, the best medical prevention method for COVID-19, to help us in the fight against this pandemic, which has claimed over half a million lives in the United States,” said acting FDA Commissioner Janet Woodcock, M.D. “The FDA, through our open and transparent scientific review process, has now authorized three COVID-19 vaccines with the urgency called for during this pandemic, using the agency’s rigorous standards for safety, effectiveness and manufacturing quality needed to support emergency use reporting Systems (VAERS) for Hanssen COVID-19 Vaccine: serious adverse events, cases of Multisystem Inflammatory Syndrome and cases of COVID-19 that result in hospitalization or death…..”,

“headline”: “\r\n FDA issues Emergency Use Authorization for the Janssen COVID-19 Vaccine, the 3rd to date. “,

“url”: “http://outbreaknewstoday.com/fda-issues-emergency-use-authorization-for-the-janssen-covid-19-vaccine-the-3rd-to-date-76713/“,

“reports”: [

{

“article\_id”: “316d251f-3392-41ae-b34f-16d2b6b11f52”,

“event\_date”: “2021-02-28 00:00:00”,

“diseases”: [

“other”,

“sars”,

“COVID-19”

],

“locations”: [

“EUA”,

“EUA”,

“Mexico”,

“U.S”,

“U.S”,

“U.S”,

“United States”,

“Mexico”,

“South America”,

“South America”,

“South Africa”,

“South Africa”,

],

“report\_id”; “63736b48-533a-4787-83c3-740027f70482”,

“syndromes”: [

“Acute respiratory syndrome”

]

}

]

}

],

“statusCode”: 200

}

}

}

Or simply a few:

query {

getArticles(key\_terms: “COVID-19”, location: “United States”, period\_of\_interest: “2021-01-01 17:00:xx to 2021-03-22 00:00:00”){

statusMessage

body {

date\_of\_publication

headline

url

reports {

event\_date

diseases

locations

}

}

statusCode

}

}

{

“data”: {

“getArticles”: {

“statusMessage”: “Query successful”,

“body”: [

{

“date\_of\_publication”; “2021-02-28 00:00:00”,

“headline”: “\r\n FDA issues Emergency Use Authorization for the Janssen COVID-19 Vaccine, the 3rd to date. “,

“url”: “https://outbreaknewstoday.com/fda-issues-emergency-use-authorization-for-the-janssen-covid-19-vaccine-the-3rd-to-date-76713/“,

“reports”: [

{

“event\_date”: “2021-02-28 00:00:00”,

“diseases”: [

“other”,

“sars”,

“COVID-19”

],

“locations”: [

“EUA”,

“EUA”,

“Mexico”,

“U.S”,

“U.S”,

“U.S”,

“United States”,

“Mexico”,

“South America”,

“South America”,

“South Africa”,

“South Africa”,

],

}

]

}

],

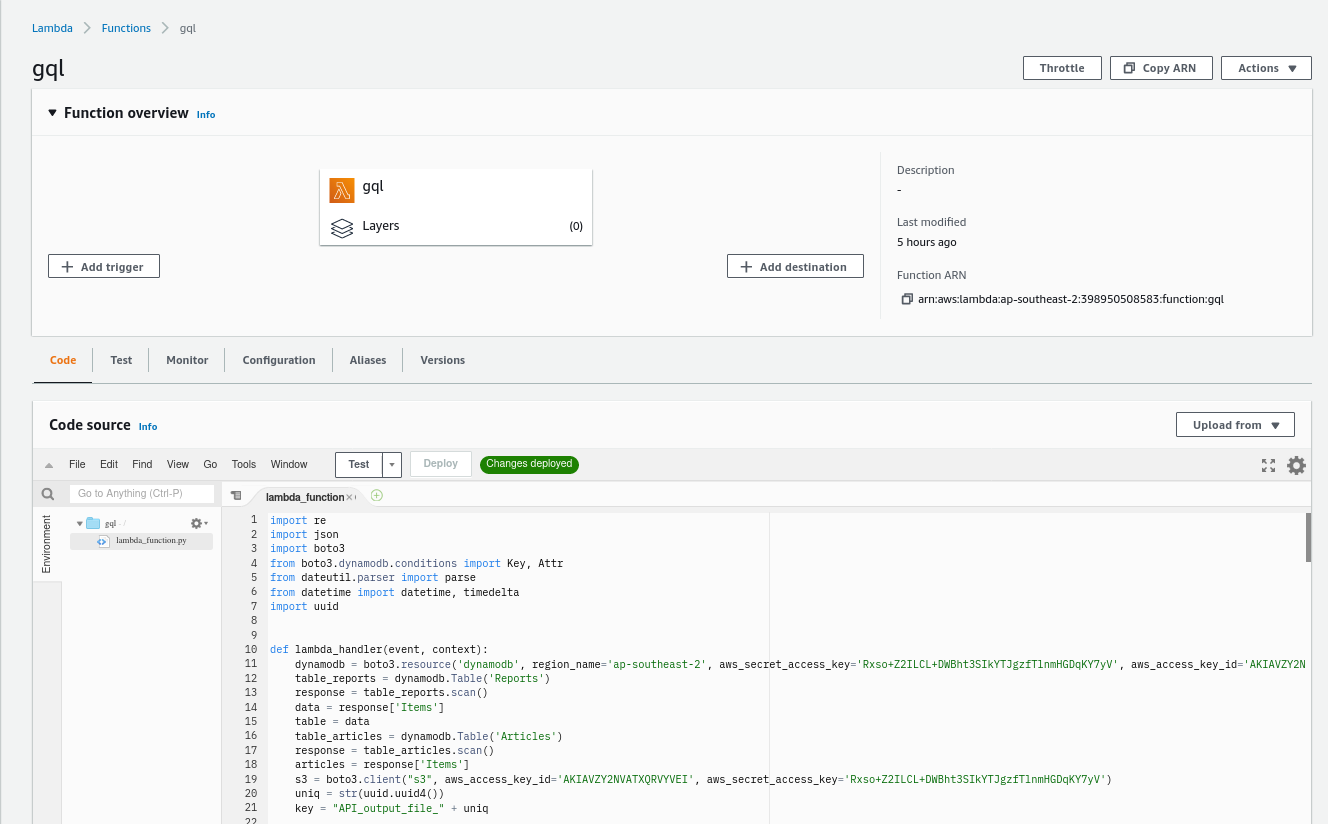
“statusCode”: 200

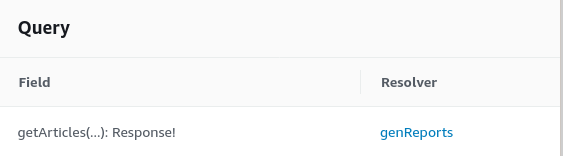
}

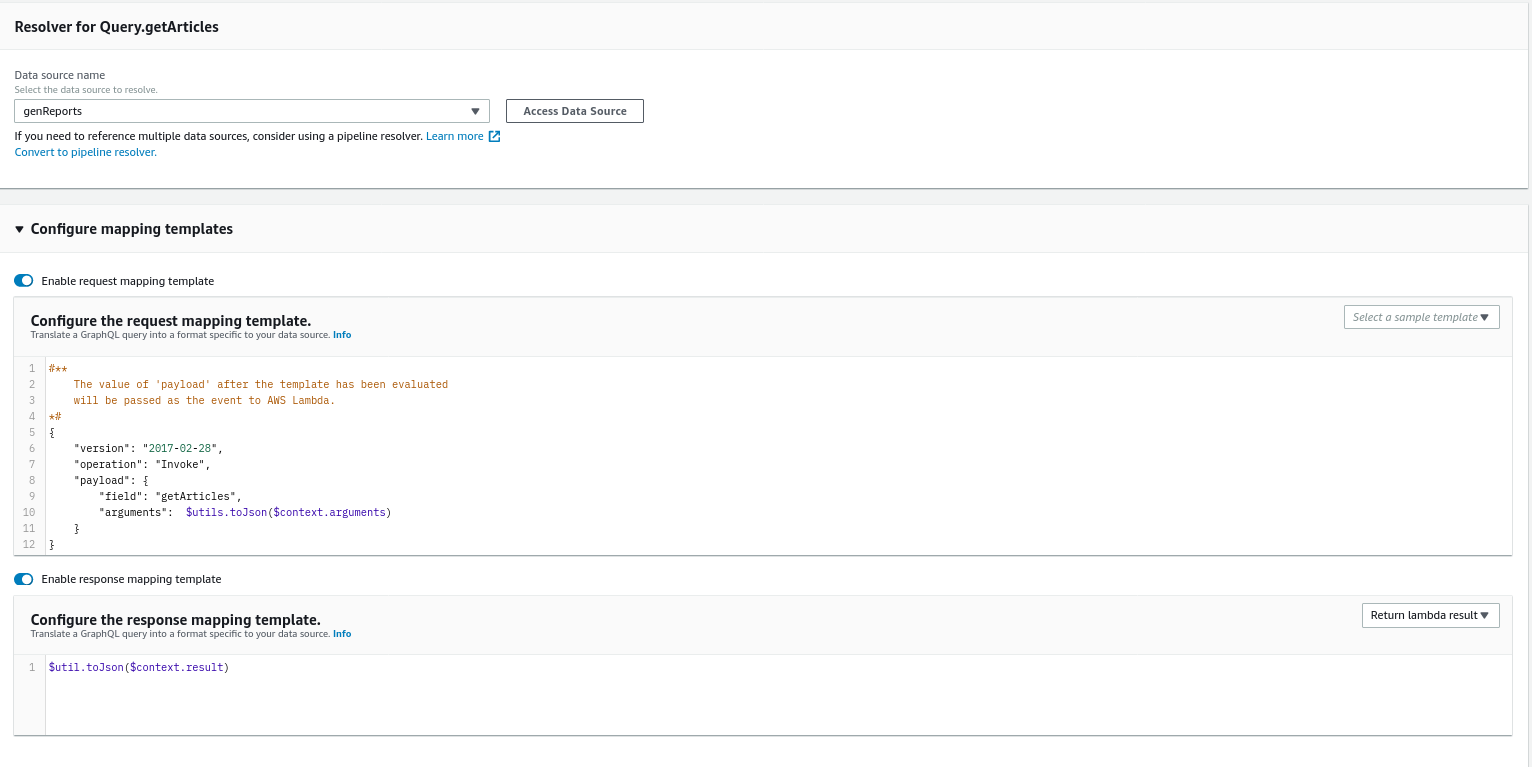
}

}

The resolver Lambda function “gql” is shown below on the console, as well as how it is “attached” to the getArticles query and configuired in VTL to return data.







The “gql” Lambda function can now resolve the request. The lambda\_handler(event, context) function is passed the query parameters in the event object as such:

{

‘field’: ‘getArticles’,

‘arguments’: {

‘period\_of\_interest’: [’2021=01=01 17:00:xx to 2021-03-22 00:00:00’],

‘key\_terms’: ‘COVID-19,sars’,

‘location’: ‘Sweden’

}

}

## External Apis

The list of external API used throughout this project include

* Google Maps Javascript API
* Auth0

## Achievements

API Achievements -

Query parameters like location can be accessed as such: event[‘arguments’][‘location’]. The code the retrieves all entries in the Reports DynamoDB table. Then it filters this table first by period\_of\_interest, then by key\_terms or location depending on whether they are provided or not. This is done through various list comprehensions, minimising the number of queries made to the DynamoDB table, given that filtering on attributes like this:

Table\_reports.scan(FilterExpression-Attr(“event\_date”).eq(date)) is performance expensive as it scans through the entire table. The results are returned in the previously specified format, with error messages for incorrectly formatted inputs (e.g bad date range) or no results for the given query.

The Api has been tested for a variety of combination of inputs to check if they work correctly.

1. Check whether the request provided contains all the fields and the format for all fields is correct. In this case the API returns a 200 response with articles as JSON objects.
2. Check where only the period\_of\_interest and key\_terms is provided in the request and the period\_of\_interest is provided in the correct format. The API returns a 200 response with the articles as JSON objects.
3. Check where only the period\_of\_interest and location is provided in the request and the period\_of\_interest is provided in the correct format. The API returns a 200 response with the articles as JSON objects
4. Check where if period\_of\_interest field is not provided in the request, the API returns a 400 response with the status message indicating the error to user
5. Check where if period\_of\_interest field is not provided in the request, the API retusn a 400 response with the status message indicating the error to user
6. Check for the case where the start date provided in the request is greater then the end date for the request, where the API returns a response with the status message indicating the error to the user
7. Check for the case where the request contains all the necessary fields in the correct format, but no articles match the request parameters. In this case the API returns a 200 response, but also the status message which indicates that the response is empty.

The cases listed above have been checked for and thoroughly verified. There use cases follow the specification given quite closely and takes care of most types of requests.

# Team Organisation

# Workload acknowledgment

|  |  |  |  |
| --- | --- | --- | --- |
| **Member name** | **SID** | **Sections responsible** | **Overall %** |
| Shantanu Kulkarni |  | Backend and API, Database, Cloud Architecture and Team Lead | 16% |
| Gaurung Rastagi |  | Frontend, Backend, Database | 16% |
| Malavika Thakore |  | Frontend, Database | 16% |
| Ben Phoebus | z5313164 | Frontend | 16% |
| Haodong Mo |  | Frontend | 16% |
| Jordan Aronson |  | Frontend, Database | 16% |

# Project Review

The team was able to develop a functional backend using AWS and GraphQL technologies, as well as a functioning frontend using the React framework. Unfortunately, both of these were only able to work in isolation. We were also able to implement some external third party APIs such as the Google Maps Javascript API and Auth0 to handle the authentication aspects of our project.