

SENG3011
Design Details Report
Team: git push --force

Members:

Austin Vuong, Elliott Yu, Cuthbert Zhong, Alvin Cheng, Jeffrey Wu

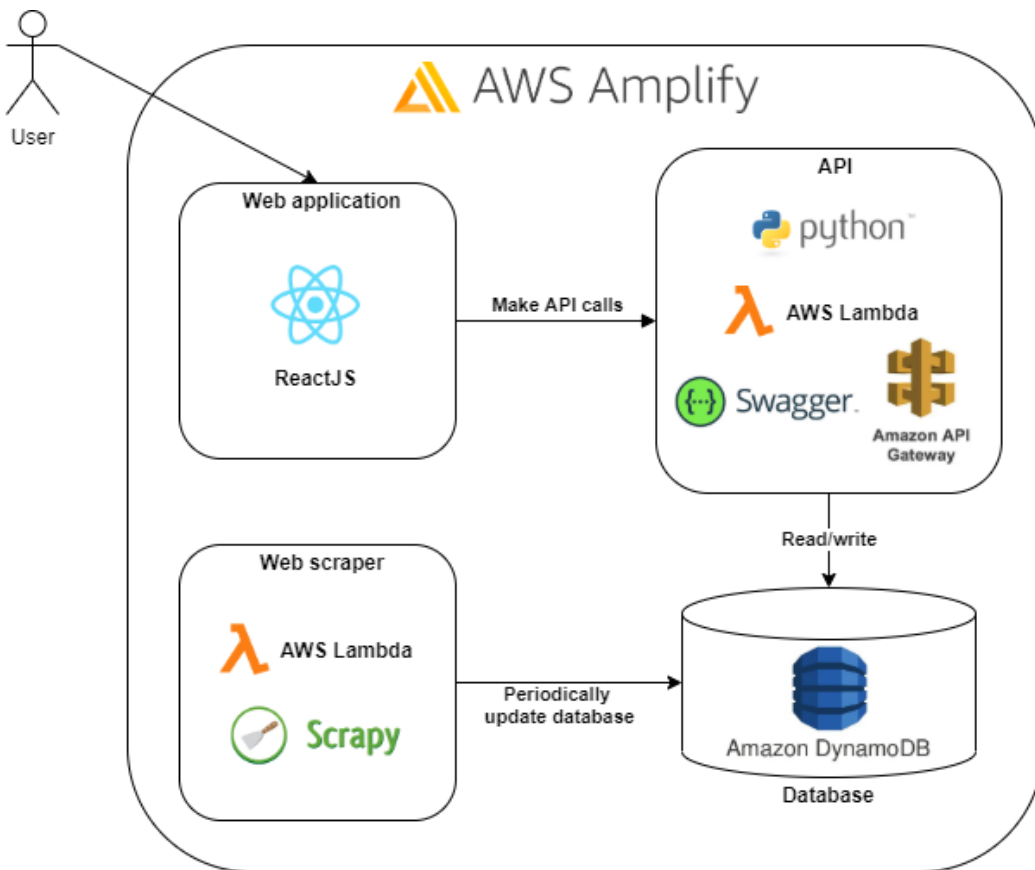
Overview

The purpose of this document is to discuss the details of the design and implementation of the team's Epidemic Outbreaks API, using the Global Incident Map's Outbreak site as the team's data source (<http://outbreaks.globalincidentmap.com/>). The site consists of entries categorised by disease/syndrome which divulge information about specific articles/reports of cases within these outbreaks around the world. The team's aim is to make this information accessible through a publicly available API which will be accessible in Web service mode as well as programmatically (eg. Web applications).

This document will discuss the following topics:

- The languages and environments that the team plan to utilise in the development of the API
- The design and approach of the API Endpoints and parameters
- An example of the intended interaction and operation of the API

Architecture Diagram



The above diagram summarises the overall software architecture of the project. In later sections, the team goes into further depth on justifying these architectural design choices as well as their benefits to the project and how the team would specifically utilise them to achieve the outcomes of the project.

As an overview, the software architecture described above is intended to be mostly reliant on our API which is backed by a DynamoDB database. This will be the main source of any data that is being queried from our data source. As such, this DynamoDB database will be periodically (weekly) updated with new data entries from our data source by the team's Web Scraper which is developed with the Scrapy Python library and run as an AWS Lambda function. These data entries will be processed and loaded into the team's DynamoDB database for future use by the API.

The API itself will be developed with use of Python as the programming language and hosted through Amazon's API Gateway and AWS Lambda. Documentation of the API will be done through Swagger.

Looking forward to Phase 2 of the project, it has been proposed that ReactJS will be the main JavaScript framework that will be of use in the development of the team's web application.

Languages and Environments

Design Paradigms

Server vs. Serverless

In recent years, there has been an increasing amount of serverless offerings such as Google Cloud Functions and AWS Lambda, which have provided alternatives to more traditional architectures where servers and infrastructure were maintained by the development team. Whilst offering many improvements from a traditional server setup, the pros and cons of each approach must be analysed before coming to a decision.

Server		Serverless	
Pros	Cons	Pros	Cons
<ul style="list-style-type: none">• Unlimited access• Complete control over project and tech stack• No function timeouts	<ul style="list-style-type: none">• High setup cost• Required to manually provision and manage infrastructure• Not easily scalable	<ul style="list-style-type: none">• Cost effective• Fast deployment and development• No need to manage or provision infrastructure• Easily scalable	<ul style="list-style-type: none">• May be locked into a particular technology or language• Not created for long running processes

Due to the limited timeframes and deliverables set in our project, we have determined that a serverless architecture would be most appropriate for this task. By eliminating the overhead of server provisioning and management, a serverless design would enable our team to focus on the development of core features and greatly speed up development and deployment times. Given the time-critical nature of our project, this would be of great benefit to our team.

NoSQL vs Relational Databases

Like serverless architectures, NoSQL databases are a more recent development aimed to reduce the complexity and overhead relating to the design and maintenance of relational databases and schemas.

Relational Databases		NoSQL Databases	
Pros	Cons	Pros	Cons
<ul style="list-style-type: none">• Supports ACID operations• Clearly structured schemas	<ul style="list-style-type: none">• Not easily scalable• Data models are not as flexible	<ul style="list-style-type: none">• Flexible data models• Faster database development	<ul style="list-style-type: none">• Does not support ACID operations• Potential for duplicate data

For much the same reasons as those for choosing serverless databases, we will be working with a NoSQL database. By not dealing with a structured relational database and schemas, development will again be faster and more adaptable to changing project requirements.

Languages and Frameworks

Scrapy

Web scraping itself presents many challenges regarding keyword extraction for summarising reports. Considering the nature of the data source, we need to extract the information from each entry from the data source as well as their corresponding linked articles. Collecting large amounts of data in this fashion will require a deep understanding of web-scraping frameworks. We have discovered 3 major frameworks that will aid the team in collecting this data. (*Social Network for Programmers and Developers*, 2021)

BeautifulSoup	Scrapy	Selenium
<ul style="list-style-type: none">• User Friendly• Easy to learn and master• Requires dependencies• Inefficient	<ul style="list-style-type: none">• Efficient• Extensibility• Portability• Not user friendly	<ul style="list-style-type: none">• Versatile• Works well with JavaScript• Developed for automated web testing• Inefficient

Among these frameworks, our team decided to use Scrapy to scrape the data from our data source and to summarise the reports. Our decision was based on the considerations below:

- Portability
 - No dependencies are required.
 - Will work well in a serverless architecture.
- Efficient
 - Written in Twisted, popular event driven networking framework for python
 - Very fast framework compared to BeautifulSoup and Selenium.
 - Asynchronous capability – no wait time for response when handling multiple requests
- Extensibility
 - Unlike BeautifulSoup it can be extended with more functionality than just string matching
 - Extended objects from Scrapy such as Linkextractor and CrawlSpider provide the ability to obtain a link object from the response object from scraping. For the data source we are using, it is required to collect data both from each entry of data as well as the referenced article in each entry. Thus these extensions will be beneficial for our particular data source due to its nested nature (*Architecture overview — Scrapy 2.4.1 documentation*, 2008).

Python

Python is an easily extensible and widely supported programming language. It is designed for Rapid Application Deployment and as a glue language to connect existing components together. (*What is Python? An Executive Summary, 2021*). Python's high level syntax also emphasizes program readability and reduces the cost of program maintenance.

Additionally, Python is the language on which Scrapy is built upon and is supported by a variety of software and infrastructure providers such as AWS, Microsoft Azure and Google Cloud. As such, the compatibility of Python with existing components as well as its wide support across a wide range of software and infrastructure services makes it an obvious choice for a programming language.

Deployment Environments

Amazon Web Services

	AWS	Google Cloud	Azure
Serverless Functions	<ul style="list-style-type: none">• Lambda	<ul style="list-style-type: none">• Cloud Functions	<ul style="list-style-type: none">• Functions
Serverless Functions Limits (Per Month)	<ul style="list-style-type: none">• 1M invocations• 400,000 GB-seconds	<ul style="list-style-type: none">• 2M Invocations• 400,000 GB-seconds	<ul style="list-style-type: none">• 1M invocations• 400,000 GB-seconds
Languages Supported	<ul style="list-style-type: none">• Node• Java• Python	<ul style="list-style-type: none">• Node• Java• Python	<ul style="list-style-type: none">• Node• Java• Python
NoSQL Databases	<ul style="list-style-type: none">• DynamoDB	<ul style="list-style-type: none">• Firestore	<ul style="list-style-type: none">• Cosmos DB
Database Limits	<ul style="list-style-type: none">• 25GB• 20 RCU's• 20 WCU's	<ul style="list-style-type: none">• 1 GB• 20K Writes (Daily)• 50K Reads• 20K Deletes	<ul style="list-style-type: none">• 5 GB• 400 Request Units per sec
Accelerated Deployment	<ul style="list-style-type: none">• Amplify	<ul style="list-style-type: none">• Firestore	<ul style="list-style-type: none">• None

As previously mentioned, a serverless architecture and NoSQL database for storing data are preferred.

This leaves us with 3 main service providers: Microsoft Azure, AWS and Google Cloud.

- Both Google Cloud and AWS provide abstraction layers to help developers create and manage backend services more quickly and easily.

For reasons explained below, our team has chosen AWS as the preferred service to host and deploy our API onto the web.

AWS Lambda

As mentioned previously, our programming language and web scraper are written in Python. As such, it is essential that the serverless compute service that we choose is compatible with our technology stack as well as efficient for development.

- Both Google and Amazon provide software to accelerate backend development.

- However, Google Firebase only offers serverless functions in JavaScript/TypeScript leaving AWS Lambda as the only service that offers Python support with substantial backend acceleration capabilities through AWS Amplify.
- Through AWS API Gateway, we can configure REST endpoints that trigger Lambda Functions, thus exposing our API to the web.
 - Lambda Functions can invoke other AWS services such as databases, allowing users to fetch data from a REST API endpoint.
- Lambda functions can also be run at scheduled times, which allows Scrapy to periodically fetch data without the need for a dedicated always online server.

Amplify

Amplify is a set of tools and services by Amazon built on top of existing AWS services to help speed up and streamline application development.

- By providing a layer of abstraction, AWS Amplify will help us to fast track our development by reducing the complexity and technical knowledge required to use AWS services.
- Amplify supports many popular web frameworks including JavaScript, React, Angular, Vue, Python, and mobile platforms including Android, iOS, React Native, Ionic, Flutter.
 - This gives us the freedom to choose from a variety of frameworks for our web application.

DynamoDB

As mentioned previously, a NoSQL database is preferred for the storage of data generated from the web scraper. Due to AWS Lambda being the choice for a serverless compute service, DynamoDB is a natural choice for a NoSQL database. As an offering by AWS with tight integration with Amazon's other offerings such as Amplify and Lambda Functions, DynamoDB will greatly improve the speed of development.

Continuous Integration and Delivery (CI/CD):

Continuous Integration is a “set of practices that drive development teams to implement small changes and check in code to version control repositories frequently” (Sacolick, 2020).

Continuous Delivery is the “automation of the delivery of applications to selected infrastructure environments” (Sacolick, 2020).

The team has decided to include CI/CD tools as part of the overall Deployment Environment in order to introduce greater efficiency in testing and deployment of work changes. This was a decision made in consideration of the fast-paced nature of the project, where automation of testing and delivery will be beneficial in narrowing the lead time from design and development to delivery.

Our team decided to utilise AWS Amplify's built-in CI/CD pipeline tools for automated deployment of our application and API. Our decision was made based on the considerations below:

- AWS Amplify
 - A set of tools and services that we already planned on leveraging for other parts of our project (API, Web App Hosting, e.t.c)
 - With this, we can deeply integrate the functionality of the CI/CD pipeline with the rest of the technology stack.
- AWS CodePipeline
 - Despite its flexibility to integrate various tools within the CI/CD process, AWS CodePipeline brings more complexity than is required within our use case in context of the project.
- Jenkins
 - An open source tool that is easily customisable with plugins (Jenkins, 2020).
 - However, Jenkins is an external tool that is separate from the AWS family of products. This may cause integration issues later in the development process which would otherwise be non-existent with a more integrated tool (ie. AWS Amplify).

Development Environments

A few key components of our team's development environment are the AWS Amplify Console for its Admin UI to manage our API and web app, Swagger for API documentation, and Postman or Insomnia for API testing.

Our team has decided to use various AWS services in the development of our project such as DynamoDB and Lambda, so AWS Amplify naturally fits in with the AWS stack. It provides a set of tools and services, such as the Admin UI and CLI, that integrate well with other AWS services by providing a layer of abstraction that removes the need to deeply upskill on each individual service as well as providing a clearer integration process. As such, this will definitely help fast track our development given the tight time frame. Amplify also supports numerous popular web frameworks such as React, Angular, Python, and mobile platforms like Android, iOS and React Native. This opens up the options we have for stage 2 of the project when we develop a web application, and also enables us to reuse the same AWS stack with minimal modifications to the existing work done.

API testing tools are an invaluable component when developing an API by providing a place to make these HTTP calls and observe the response. Beyond that, these tools can also provide functionality like writing test scripts to ensure the API is working as expected, and managing and organising different requests and environments. Given that these tools are to be used individually, no team-wide decision needs to be made about which one to use, Postman and Insomnia are two great options that both provide the aforementioned functionality and are well documented. However, it is worth noting that Insomnia only provides team collaboration with their paid Teams plan, while Postman allows for 25 shared requests on the free plan.

SwaggerHub will be used to document our API and host on a public URL, providing users with a UI for calling the API. SwaggerHub was chosen because of its simplicity to achieve the outcomes of documentation in the context of the project. As we have chosen to use AWS Amplify for implementation which creates a single resource in Amazon API Gateway (Sans, 2020) under the hood, this provides support for exporting the API as OpenAPI 2/3 which can then be exported into SwaggerHub to be hosted. The popularity of SwaggerHub also makes it so that there is plenty of documentation and guides available compared to other platforms like Stoplight which has much less. However, the free plan for SwaggerHub only permits 1 user. Despite this, our team has a comprehensive management plan to delegate specific roles to each team member. As such, the limited user issue with SwaggerHub should not create any issue.

API/Parameters Discussion

In preparation for the implementation of the API, an analysis of the data source is required to determine the parameters that can be made available for the API to be utilised effectively. This was achieved by observing the structure of the data provided by the data source and understanding how this data can be subsequently scraped and organised within our database.

Data Source Analysis

Each entry provided within the “Outbreaks Global Incident Map” data source is grouped within the respective tables that relate to the disease they are reporting on.

The entries themselves provide details including:

- date/time
- country
- city
- description

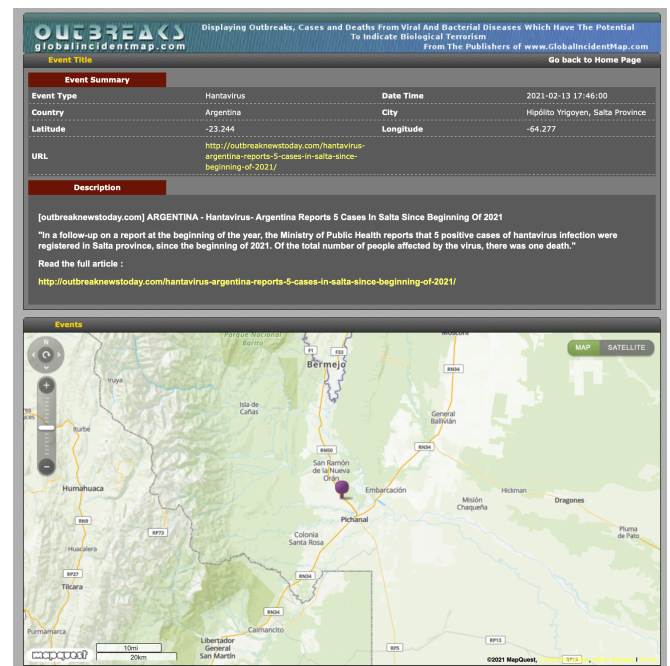


DATE/TIME	DETAIL	COUNTRY	CITY	
2021-02-13 17:46:00	Detail	Argentina	Hipólito Yrigoyen, Salta Province	ARGENTINA - Hantavirus- Argentina Reports 5 Cases In Salta Since Beginning Of 2021
2021-02-13 17:45:00	Detail	Argentina	General Mosconi, Salta Province	ARGENTINA - Hantavirus- Argentina Reports 5 Cases In Salta Since Beginning Of 2021
2021-02-13 17:42:00	Detail	Argentina	Colonia Santa Rosa, Salta Province	ARGENTINA - Hantavirus- Argentina Reports 5 Cases In Salta Since Beginning Of 2021
2021-02-13 16:49:00	Detail	Argentina	Orán, Salta Province	ARGENTINA - Hantavirus- Argentina Reports 5 Cases In Salta Since Beginning Of 2021

Source: <http://outbreaks.globalincidentmap.com>

However, each entry is paired with a “details” link that opens a new page that provides the above information that is shown in the table as well as further information including coordinates, a url to the related article and an extended description of the entry.

From this, we can determine that the relevant data can be sourced from both the entry on the “Outbreaks Global Incident Map” data source as well as the article that is referenced in each entry. This will form the data required by the specification of which will be stored within our database and served to the user by the API when queried. The specific sources and form of this data will be discussed in a later section.



Event Summary

Event Type	Hantavirus	Date Time	2021-02-13 17:46:00
Country	Argentina	City	Hipólito Yrigoyen, Salta Province
Latitude	-23.244	Longitude	-64.277
URL	http://outbreaknewstoday.com/hantavirus-argentina-reports-5-cases-in-salta-since-beginning-of-2021/		

Description

[outbreaknewstoday.com] ARGENTINA - Hantavirus- Argentina Reports 5 Cases In Salta Since Beginning Of 2021

"In a follow-up on a report at the beginning of the year, the Ministry of Public Health reports that 5 positive cases of hantavirus infection were registered in Salta province, since the beginning of 2021. Of the total number of people affected by the virus, there was one death."

Read the full article : <http://outbreaknewstoday.com/hantavirus-argentina-reports-5-cases-in-salta-since-beginning-of-2021/>

Events

Map showing the location of the outbreak in Salta Province, Argentina.

Source: <http://outbreaks.globalincidentmap.com/eventdetail.php?ID=37805>

API/Parameter Design

From analysis of the data source, we can determine the relevant parameters that will be required to utilise our API. The parameters are discussed below.

The three required parameters as outlined in project specification:

- period of interest/time (start_date, end_date) (*path*)
 - This will consist of start and end date conforming to the format required:
 - (yyyy-MM-ddTHH:mm:ss)
 - The API will expect these dates to be passed in as strings. Therefore, if the API is programmatically called, any date objects that may be used to represent this must be stringified.
 - If certain components of the date are not provided by the user, this should be handled in the front-end for error checking before being passed to the API
 - Any errors that are passed into the API will result in the API returning a 400 Bad Request response
- key_terms (*path*)
 - Key terms provided to the API will be accepted as a comma-separated string of characters.
 - An empty input for this parameter will not result in a failed response. All relevant results of the search based on other parameters will be returned instead.
 - For this parameter, we can expect to determine relevant results by identifying whether these key terms appear in the description of the relevant reports/entries.
 - By using indexing, we can provide a more efficient search of key terms to return relevant data.
 - To narrow our search, the time period and location filters based on the given parameters of the call can be applied first then the key terms search.
- location (*path*)
 - Locations provided to the API will be subject to processing of relevant locations that may be of interest to expand the search through use of geo-locations (as suggested in the project specification).
 - As stated this will narrow the search of the relevant reports to be returned to the endpoint.

Additional parameters:

- disease (*path*)
 - A specific disease can form a parameter where the user calling the API intends to only receive reports relating to a single disease.
 - This will allow for the search to be narrowed significantly.
- syndrome (*path*)
 - A specific syndrome can form a parameter where the user calling the API intends to only receive reports relating to a single syndrome.
 - This will allow for the search to be narrowed significantly.
- user & password (*body*)
 - An account will be required to access the API.

- With user and password parameters, the API will be able to generate a token for the user to use when interacting with the API (eg. passed in as header on each request).
- token (*header*)
 - Each request will require an accompanying token to ensure that access has been granted for this particular user to use the API.
 - This is in place to ensure that the API will be used by users who have been granted access to avoid hitting rate limits and/or malicious attacks.
 - This will take the form of JSON web tokens (JWT).

API Endpoints

The potential endpoints that will form our API will be discussed below. As stated, each endpoint will require a Token for Authorization except the login and register endpoints. (Eg. Token = eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....). Also, each endpoint (except the authentication ones) will respond to successful queries with a JSON object conforming to the format required in the project specification. Unsuccessful queries will be handled accordingly with their respective HTTP response codes.

N.B The design of the API and its endpoints were inspired by a sample Swagger doc (<https://petstore.swagger.io/#/>)

Reports Search Endpoint

This endpoint will consist of multiple versions where the user is able to search based on the filters that they choose to provide, which can involve one or multiple parameters

- *GET reports/?period={period}&location={location}&terms={terms}*
 - eg. period = "2020-10-01T08:45:10-2020-11-01T19:37:12"
 - eg. location = "Sydney"
 - eg. terms = "Coronavirus,Flu"
 - (UPDATE THIS!)
- *GET reports/?period={period}*
 - For this endpoint, the response will consist of results filtered only by the provided date range.
 - eg. period = "2020-10-01T08:45:10-2020-11-01T19:37:12"
- *GET reports/?location={location}*
 - For this endpoint, the response will consist of results filtered only by the provided location.
 - eg. location = "Sydney"
- *GET reports/?terms={terms}*
 - For this endpoint, the response will consist of results filtered only by the provided terms.
 - eg. terms = "Coronavirus,Flu"

Disease-Based Search Endpoint

This endpoint will be based on one parameter where a specific disease is queried for, returning all the reports for that particular disease

- *GET reports/disease?disease={disease}*
 - Only the specified disease will be used as a filter to form the response
 - eg. disease = "Coronavirus"

Syndrome-Based Search Endpoint

This endpoint will be based on one parameter where a specific syndrome is queried for, returning all the reports for that particular syndrome

- *GET reports/syndrome?syndrome={syndrome}*
 - Only the specified syndrome will be used as a filter to form the response
 - eg. syndrome = "Meningitis"

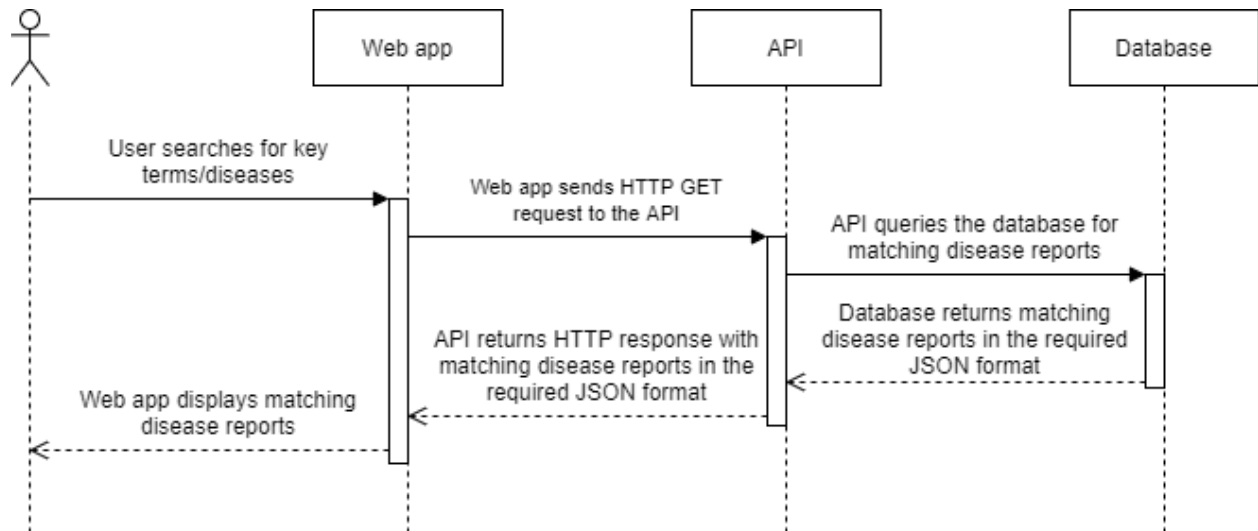
Authentication Endpoints

These endpoints will allow users to register/login to use the API. Each of these endpoints will return a response containing a token which will enable them to use the other endpoints of the API. This will be implemented in order to defend against malicious attacks.

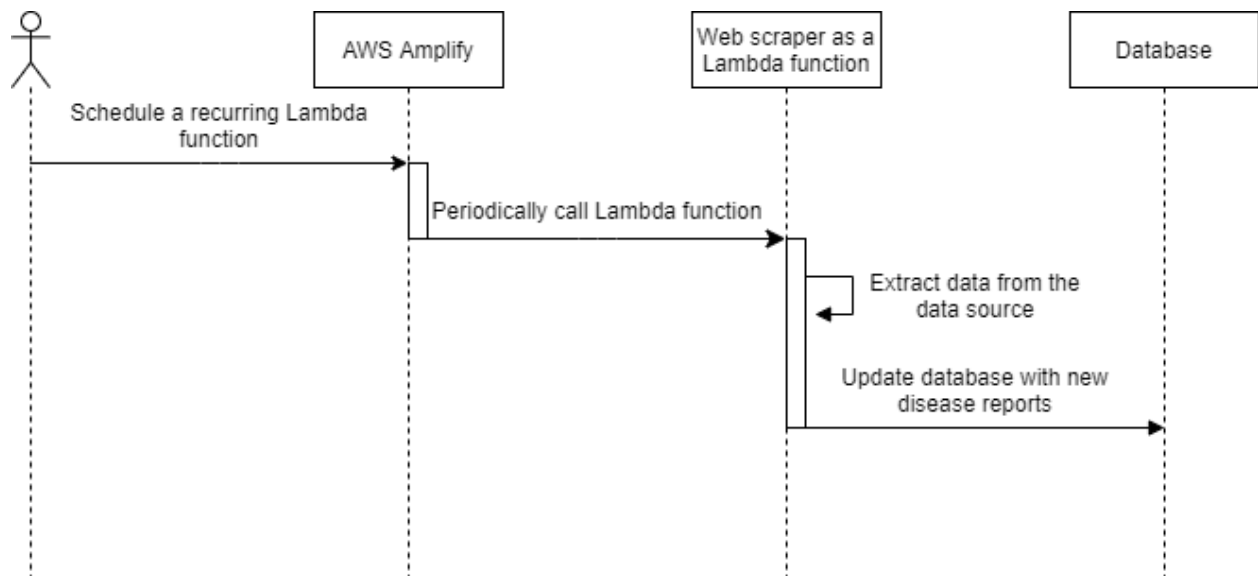
- *POST auth/login*
 - body: user: <username:string>, password: <password:string>
 - eg. response - Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....
- *POST auth/register*
 - body: user: <username:string>, password: <password:string>
 - eg. response - Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....

API Interaction Design

The API will be deployed on AWS Amplify such that it will run in Web service mode with the user interacting with the Swagger docs to trigger API requests. These API requests will be fulfilled by the API itself and will subsequently form responses that will be consumed by the user. These responses will be formed by queries to an AWS DynamoDB database. Relevant data will then be processed and returned by the API to the user.



In order to extract data from the intended data source, a web scraper is required. This web scraper will also be deployed on AWS Amplify as an AWS Lambda function which will be periodically triggered to update the database with the latest data from the data source.



API Implementation Details

API Final Architecture Details

Web Scraper

AWS Lambda

As mentioned previously, our programming language and web scraper are written in Python. As such, it is essential that the serverless compute service that we choose is compatible with our technology stack as well as efficient for development.

- Both Google and Amazon provide software to accelerate backend development.
 - However, Google Firebase only offers serverless functions in JavaScript/TypeScript leaving AWS Lambda as the only service that offers Python support with substantial backend acceleration capabilities through AWS Amplify.
- AWS Lambda functions can also be run at scheduled times, which allows Scrapy to periodically fetch data without the need for a dedicated always online server.
- AWS Lambda also integrates with DynamoDB and other AWS services, allowing us to connect our scraper to other parts of the software architecture easily.

Thus, AWS Lambda was chosen as the serverless compute platform for our scraper to be hosted on the web.

Challenges and Shortcomings

One of the main challenges with running the Web Scraper on AWS Lambda was importing the correct Python dependencies and packages. This was particularly challenging due to the lack of documentation regarding Python functions on AWS Amplify. This challenge was addressed through the use of AWS Lambda Layers, which allows Lambda functions to import additional code such as runtimes, dependencies and Python packages.

However, by using an AWS Lambda Layer, we were unable to test our function locally on our own machine, which required us to upload and provision resources in the cloud in order to test and develop our Lambda function. This was a time consuming and laborious process and most definitely a shortcoming of our development.

Scrapy

As previously mentioned, Scrapy was chosen to be the main tool to scrape our data source for the team's API implementation. It is a tool specially created for downloading and saving data from the web which makes it more suitable for the project than other frameworks like BeautifulSoup or Selenium. It also outputs the result in JSON format such that it can be efficiently accessed and processed in conjunction with our NoSQL based database (DynamoDB).

Challenges and Shortcomings

Due to the nature of our data source, it was challenging to collect every piece of data from the articles linked from our data source. The linked articles are from more than 50 different websites, meaning that individual spiders would need to be created to crawl each type of website with a different set of rules. The team has decided that this is unfeasible with the given timeframe for the project. However, it was possible to collect detailed summaries of those articles via scraping each data entry, not the linked article. The linked articles were still crawled down for collecting some key terms with basic spiders.

Database

Amazon DynamoDB

Amazon DynamoDB was chosen for the team's API implementation due to its NoSQL structure. As the intended storage and output of data for the API is of JSON format, the team felt the database should also match this format. Therefore, utilising a document-based database (NoSQL) that mirrors this JSON format, will allow for efficient searching and access of relevant data to be served to the API. This would avoid unnecessary pre-processing of data which may over-complicate or introduce delays in the API response (eg. Object-relational mapping and table joins).

Challenges and Shortcomings

Through use of Amazon DynamoDB paired with AWS Amplify, configuration was exclusively done with AWS Amplify's CLI. Although this ended up proving no delays in configuration, this initially was a challenge due to the lack of visual interfaces to configure the database model, unlike Google's Firestore. This was a challenge in ensuring the database model design was communicated properly amongst team members without having a visual idea of the model being accessible through AWS Amplify's console. However, this was resolved through agreeing upon the model conceptually and reflecting this idea throughout the implementation of the other components of the API, including putting to the database from the scraper in AWS Lambda and getting from the database through AWS API Gateway/AWS Lambda for the API itself.

API

The API implementation involved utilising several AWS services with AWS Amplify providing a layer of abstraction and both a UI and CLI for setup. These services include API Gateway for API management, AWS Lambda as the backend for the API, DynamoDB as the database, and Python as our language of choice.

AWS Lambda

Lambda was chosen as our serverless solution instead of running a server, on an EC2 instance for example. There was no need for a server to be run when no calls were being made, removing the need for resources being consumed when no calls were being made to the API, as well as the overhead in server provisioning and management.

Another selling point of Lambda was its UI allowing developers to edit the function's source code and deploying it directly, and also with a testing page to immediately test any changes. This would immensely speed up the development process by removing the need to constantly push local changes to AWS through the CLI.

AWS API Gateway

Following that, API Gateway integrates seamlessly with Amplify, using Lambda functions for its backend while also providing an excellent UI for easy management and testing of the API. It is worth noting that with this choice in infrastructure, the initial setup of the API was as simple as using the 'amplify add api' command in the Amplify CLI then following a few prompts. This was crucial in allowing our team to focus on development rather than setup.

Challenges and shortcomings

The abstraction provided by AWS Amplify in the setup of the API provided an efficient start to the project. However, after the setup, the team still had to learn each service (Lambda, API Gateway and DynamoDB) to some extent to be able to implement the API. To begin with, access permissions across the different services were confusing to set up with another AWS service called Identity and Access Management (IAM) involved. Luckily, after the first setup with the help of documentation and guides, further Lambda functions and API endpoints followed the same procedure and our team documented this knowledge to share amongst each other, eliminating the need for everyone to learn the process.

Another challenge the team came across in the development of the API was debugging, especially with so many different parts and services involved. For example, errors could be from syntax in the Python code, access permissions between API Gateway, Lambda and DynamoDB, incorrect use of Boto3 methods (AWS's Python SDK) when querying the database, incorrect method setup in API Gateway or connecting to the wrong Lambda function, and much more. This challenge was especially prominent due to our lack of experience with AWS services as a team, but we were able to overcome this by learning how to navigate the UI/CLI for each

service. For example, Lambda functions can be tested locally using the 'amplify mock function <functionName>' command, or on the AWS console online, both of which show output and logs. API Gateway also has similar testing functionality on the online AWS console which shows the response to the user, as well as from its associated Lambda function with all errors included. Moreover, in our weekly meetings as a team, we investigated errors and learnt how to use these services together to both share the knowledge and ensure everyone was on the same page.

Logging

Our team's chosen method of logging details about the API consists of two parts: a JSON snippet that is returned along with the API response to the user, and in AWS CloudWatch. The snippet returned in the response body describes our team's name, time the API was accessed, and the API's data source. This information is tailored towards the end user and provides them with more context. On the other hand, more technical logs that are helpful to us developers are stored in AWS CloudWatch, a monitoring and observability service. The choice to incorporate this service was driven by the fact that AWS Lambda (the backend of our API) supported automatic logging of request details to CloudWatch, while also allowing the use of Python's 'logging' library to easily enrich these logs. In particular, we log the request event (incl. Query parameters, endpoint called, and referrer such as SwaggerHub) and response object (incl. Status code and body) sent to and from the API, useful for debugging. Moreover, Lambda automatically provides the function duration and memory used, useful for insight into resource utilisation.

Example success response log

▶	2021-03-18T23:34:34.744+11:00	START RequestId: d642367a-62f2-43af-b224-b3f121172777 Version: \$LATEST	
▶	2021-03-19T01:27:23.419+11:00	START RequestId: b22a489d-8498-4460-884d-683d97b74c18 Version: \$LATEST	
▶	2021-03-19T01:27:23.420+11:00	received event:	
▼	2021-03-19T01:27:23.420+11:00	<pre>{'resource': '/reports', 'path': '/reports', 'httpMethod': 'GET', 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate, br', 'CloudFront-Forwarded-Proto': 'https', 'CloudFront-Is-Desktop-Viewer': 'true', 'CloudFront-Is-Mobile-Viewer': 'false', 'CloudFront-Is-SmartTV-Viewer': 'false', 'CloudFront-Is-Tablet-Viewer': 'false', 'CloudFront-Viewer-Country': 'AU', 'Host': '6u977749j2.execute-api.ap-southeast-2.amazonaws.com', 'Postman-Token': 'ea57f00e-bd2c-4003-8172-05c4a305f329', 'User-Agent': 'PostmanRuntime/7.26.10', 'Via': '1.1 1b8d25a254f659ff0989f0e7bb579086.cloudfront.net (CloudFront)', 'X-Amz-Cf-Id': 'S6-Qt1x4YLjXw5as0f4PbcACvkhn8TZ_e14Fzck8N5nRzqhSUCcKpA==', 'X-Amzn-Trace-Id': 'Root=1-6053634a-67185e796065a2f5605b86a8', 'X-Forwarded-For': '58.173.101.182, 130.176.156.125', 'X-Forwarded-Port': '443', 'X-Forwarded-Proto': 'https'}, 'multiValueHeaders': {'Accept': ['*/*'], 'Accept-Encoding': ['gzip, deflate, br'], 'CloudFront-Forwarded-Proto': ['https'], 'CloudFront-Is-Desktop-Viewer': ['true'], 'CloudFront-Is-Mobile-Viewer': ['false'], 'CloudFront-Is-SmartTV-Viewer': ['false'], 'CloudFront-Is-Tablet-Viewer': ['false'], 'CloudFront-Viewer-Country': ['AU'], 'Host': ['6u977749j2.execute-api.ap-southeast-2.amazonaws.com'], 'Postman-Token': ['ea57f00e-bd2c-4003-8172-05c4a305f329'], 'User-Agent': ['PostmanRuntime/7.26.10'], 'Via': ['1.1 1b8d25a254f659ff0989f0e7bb579086.cloudfront.net (CloudFront)', 'X-Amz-Cf-Id': ['S6-Qt1x4YLjXw5as0f4PbcACvkhn8TZ_e14Fzck8N5nRzqhSUCcKpA=='], 'X-Amzn-Trace-Id': ['Root=1-6053634a-67185e796065a2f5605b86a8'], 'X-Forwarded-For': ['58.173.101.182, 130.176.156.125'], 'X-Forwarded-Port': ['443'], 'X-Forwarded-Proto': ['https']}, 'queryStringParameters': {'terms': 'zombie'}, 'multiValueQueryStringParameters': {'terms': ['zombie']}, 'pathParameters': None, 'stageVariables': None, 'requestContext': {'resourceId': 'mpabaz', 'resourcePath': '/reports', 'httpMethod': 'GET', 'extendedRequestId': 'cYxztGbvSwMF90Q=', 'requestTime': '18/Mar/2021:14:27:22 +0000', 'path': '/staging/reports', 'accountId': '675954207968', 'protocol': 'HTTP/1.1', 'stage': 'staging', 'domainPrefix': '6u977749j2', 'requestTimeEpoch': 1616077642826, 'requestId': '67e2e4a9-5849-4c86-bb8e-f8d41b7bbd8a', 'identity': {'cognitoIdentityPoolId': None, 'accountId': None, 'cognitoIdentityId': None, 'caller': None, 'sourceIp': '58.173.101.182', 'principalOrgId': None, 'accessKey': None, 'cognitoAuthenticationType': None, 'cognitoAuthenticationProvider': None, 'userArn': None, 'userAgent': 'PostmanRuntime/7.26.10', 'user': None, 'domainName': '6u977749j2.execute-api.ap-southeast-2.amazonaws.com', 'apiId': '6u977749j2'}, 'body': None, 'isBase64Encoded': False}</pre>	Copy
▶	2021-03-19T01:27:23.621+11:00	[INFO] 2021-03-18T14:27:23.620Z b22a489d-8498-4460-884d-683d97b74c18 Response JSON:	
▼	2021-03-19T01:27:23.621+11:00	<pre>[INFO] 2021-03-18T14:27:23.621Z b22a489d-8498-4460-884d-683d97b74c18 { "statusCode": 404, "headers": { "Content-Type": "application/json" }, "body": "{\"message\": \"No results found\"}" }</pre>	Copy
▶	2021-03-19T01:27:23.640+11:00	END RequestId: b22a489d-8498-4460-884d-683d97b74c18	
▼	2021-03-19T01:27:23.640+11:00	REPORT RequestId: b22a489d-8498-4460-884d-683d97b74c18 Duration: 220.55 ms Billed Duration: 221 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 336.82 ms	Copy
		<pre>Copy - The dengue mosquito has been found in Tennant Creek triggering a large response\\\" to eliminate any spread of the insects and the diseases they carry.\\\"\\\", \"reports\": [{\"diseases\": [\"dengue\"], \"syndromes\": [], \"locations\": [{\"country\": \"Australia\", \"location\": \"Tennant Creek NT\"}], \"event_date\": \"2021-03-09 13:07:00\"}], \"headline\": \"Dengue mosquito found in Tennant Creek\", \"date_of_publication\": \"2021-03-09 13:07:00\", \"url\": \"https://www.katherinetimes.com.au/story/7159298/dengue-mosquito-found-in-tennant-creek/\", {\"logs\": {\"team\": \"git push --force\", \"accessed-time\": \"2021-03-18 12:34:34\", \"data-source\": \"Global incident map\"}}}]</pre>	
▶	2021-03-18T23:34:34.982+11:00	END RequestId: d642367a-62f2-43af-b224-b3f121172777	
▼	2021-03-18T23:34:34.982+11:00	REPORT RequestId: d642367a-62f2-43af-b224-b3f121172777 Duration: 238.11 ms Billed Duration: 239 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 308.56 ms	Copy

Example 404 no results found log

▶	2021-03-19T01:27:23.419+11:00	START RequestId: b22a489d-8498-4460-884d-683d97b74c18 Version: \$LATEST	
▶	2021-03-19T01:27:23.420+11:00	received event:	
▼	2021-03-19T01:27:23.420+11:00	<pre>{ 'resource': '/reports', 'path': '/reports', 'httpMethod': 'GET', 'headers': { 'Accept': '/*/*', 'A... { 'resource': '/reports', 'path': '/reports', 'httpMethod': 'GET', 'headers': { 'Accept': '/*/*', 'Accept-Encoding': 'gzip, deflate, br', 'CloudFront-Forwarded-Proto': 'https', 'CloudFront-Is-Desktop-Viewer': 'true', 'CloudFront-Is-Mobile- Viewer': 'false', 'CloudFront-Is-SmartTV-Viewer': 'false', 'CloudFront-Is-Tablet-Viewer': 'false', 'CloudFront-Viewer- Country': 'AU', 'Host': '6u977749j2.execute-api.ap-southeast-2.amazonaws.com', 'Postman-Token': 'ea57f00e-bd2c-4003-8172-05c4a305f329', 'User-Agent': 'PostmanRuntime/7.26.10', 'Via': '1.1 1b8d25a254f659ff0989f0e7bb579086.cloudfront.net (CloudFront)', 'X-Amz-Cf-Id': 'S6- Qt1x4YLjXw5as0f4PbcACvkhnBTZ_el4Fzck8N5nRzqhSUCcKpA==', 'X-Amzn-Trace-Id': 'Root=1-6053634a-67185e796065a2f5605b86a8', 'X-Forwarded-For': '58.173.101.182, 130.176.156.125', 'X-Forwarded-Port': '443', 'X-Forwarded-Proto': 'https', 'multiValueHeaders': { 'Accept': ['/*/*'], 'Accept-Encoding': ['gzip, deflate, br'], 'CloudFront-Forwarded-Proto': ['https'], 'CloudFront-Is-Desktop-Viewer': ['true'], 'CloudFront- Is-Mobile-Viewer': ['false'], 'CloudFront-Is-SmartTV-Viewer': ['false'], 'CloudFront-Is-Tablet-Viewer': ['false'], 'CloudFront-Viewer- Country': ['AU'], 'Host': ['6u977749j2.execute-api.ap-southeast-2.amazonaws.com'], 'Postman-Token': ['ea57f00e-bd2c-4003-8172- 05c4a305f329'], 'User-Agent': ['PostmanRuntime/7.26.10'], 'Via': ['1.1 1b8d25a254f659ff0989f0e7bb579086.cloudfront.net (CloudFront)'], 'X-Amz-Cf-Id': ['S6-Qt1x4YLjXw5as0f4PbcACvkhnBTZ_el4Fzck8N5nRzqhSUCcKpA=='], 'X-Amzn-Trace-Id': ['Root=1-6053634a- 67185e796065a2f5605b86a8'], 'X-Forwarded-For': ['58.173.101.182, 130.176.156.125'], 'X-Forwarded-Port': ['443'], 'X-Forwarded-Proto': ['https']}, 'queryStringParameters': { 'terms': 'zombie'}, 'multiValueQueryStringParameters': { 'terms': ['zombie']}, 'pathParameters': None, 'stageVariables': None, 'requestContext': { 'resourceId': 'mpabaz', 'resourcePath': '/reports', 'httpMethod': 'GET', 'extendedRequestId': 'cYxztGbvSwMF90Q=', 'requestTime': '18/Mar/2021:14:27:22 +0000', 'path': '/staging/reports', 'accountId': '675954207968', 'protocol': 'HTTP/1.1', 'stage': 'staging', 'domainPrefix': '6u977749j2', 'requestTimeEpoch': 1616077642826, 'requestId': '67e2e4a9-5849-4c86-bb8e-f8d41b7bbd8a', 'identity': { 'cognitoIdentityPoolId': None, 'accountId': None, 'cognitoIdentityId': None, 'caller': None, 'sourceIp': '58.173.101.182', 'principalOrgId': None, 'accessKey': None, 'cognitoAuthenticationType': None, 'cognitoAuthenticationProvider': None, 'userArn': None, 'userAgent': 'PostmanRuntime/7.26.10', 'user': None}, 'domainName': '6u977749j2.execute-api.ap-southeast-2.amazonaws.com', 'apiId': '6u977749j2'}, 'body': None, 'isBase64Encoded': False}</pre>	Copy
▶	2021-03-19T01:27:23.621+11:00	[INFO] 2021-03-18T14:27:23.620Z b22a489d-8498-4460-884d-683d97b74c18 Response JSON:	
▼	2021-03-19T01:27:23.621+11:00	<pre>[INFO] 2021-03-18T14:27:23.621Z b22a489d-8498-4460-884d-683d97b74c18 {"statusCode": 404, "headers... { "statusCode": 404, "headers": { "Content-Type": "application/json" }, "body": "{\"message\": \"No results found\"}" }</pre>	Copy
▶	2021-03-19T01:27:23.640+11:00	END RequestId: b22a489d-8498-4460-884d-683d97b74c18	
▼	2021-03-19T01:27:23.640+11:00	REPORT RequestId: b22a489d-8498-4460-884d-683d97b74c18 Duration: 220.55 ms Billed Duration: 221 m... REPORT RequestId: b22a489d-8498-4460-884d-683d97b74c18 Duration: 220.55 ms Billed Duration: 221 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 336.82 ms	Copy

Example 400 bad request log

▶	2021-03-19T01:30:30.117+11:00	START RequestId: b5efd2d7-46d6-4725-b6b0-aae7f37d5c97 Version: \$LATEST
▶	2021-03-19T01:30:30.121+11:00	received event:
▼	2021-03-19T01:30:30.121+11:00	<pre>{ 'resource': '/reports', 'path': '/reports', 'httpMethod': 'GET', 'headers': { 'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate, br', 'CloudFront-Forwarded-Proto': 'https', 'CloudFront-Is-Desktop-Viewer': 'true', 'CloudFront-Is-Mobile-Viewer': 'false', 'CloudFront-Is-SmartTV-Viewer': 'false', 'CloudFront-Is-Tablet-Viewer': 'false', 'CloudFront-Viewer-Country': 'AU', 'Host': '6u977749j2.execute-api.ap-southeast-2.amazonaws.com', 'Postman-Token': '079215a0-272d-462d-8ab6-42b06320716f', 'User-Agent': 'PostmanRuntime/7.26.10', 'Via': '1.1 1b8d25a254f659ff0989f0e7bb579086.cloudfront.net (CloudFront)', 'X-Amz-Cf-Id': '6QToZfrIVQ9RJGHXV9Ug0SXoMVLqMs7La4prU6fjFisvUUiiTb41gw==', 'X-Amzn-Trace-Id': 'Root=1-60536406-51ff9c4c22bb282557c661f0', 'X-Forwarded-For': '58.173.101.182, 130.176.156.185', 'X-Forwarded-Port': '443', 'X-Forwarded-Proto': 'https', 'multiValueHeaders': { 'Accept': ['*/*'], 'Accept-Encoding': ['gzip, deflate, br'], 'CloudFront-Forwarded-Proto': ['https'], 'CloudFront-Is-Desktop-Viewer': ['true'], 'CloudFront-Is-Mobile-Viewer': ['false'], 'CloudFront-Is-SmartTV-Viewer': ['false'], 'CloudFront-Is-Tablet-Viewer': ['false'], 'CloudFront-Viewer-Country': ['AU'], 'Host': ['6u977749j2.execute-api.ap-southeast-2.amazonaws.com'], 'Postman-Token': ['079215a0-272d-462d-8ab6-42b06320716f'], 'User-Agent': ['PostmanRuntime/7.26.10'], 'Via': ['1.1 1b8d25a254f659ff0989f0e7bb579086.cloudfront.net (CloudFront)'], 'X-Amz-Cf-Id': ['6QToZfrIVQ9RJGHXV9Ug0SXoMVLqMs7La4prU6fjFisvUUiiTb41gw=='], 'X-Amzn-Trace-Id': ['Root=1-60536406-51ff9c4c22bb282557c661f0'], 'X-Forwarded-For': ['58.173.101.182, 130.176.156.185'], 'X-Forwarded-Port': ['443'], 'X-Forwarded-Proto': ['https'] }, 'queryStringParameters': None, 'multiValueQueryStringParameters': None, 'pathParameters': None, 'stageVariables': None, 'requestContext': { 'resourceId': 'mpabaz', 'resourcePath': '/reports', 'httpMethod': 'GET', 'extendedRequestId': 'cYyQ9HliSWMFlog=', 'requestTime': '18/Mar/2021:14:30:30 +0000', 'path': '/staging/reports', 'accountId': '675954207968', 'protocol': 'HTTP/1.1', 'stage': 'staging', 'domainPrefix': '6u977749j2', 'requestTimeEpoch': 1616077830085, 'requestId': '8ad76598-e21c-4136-98d7-aca31f6e7e2c', 'identity': { 'cognitoIdentityPoolId': None, 'accountId': None, 'cognitoIdentityId': None, 'caller': None, 'sourceIp': '58.173.101.182', 'principalOrgId': None, 'accessKey': None, 'cognitoAuthenticationType': None, 'cognitoAuthenticationProvider': None, 'userArn': None, 'userAgent': 'PostmanRuntime/7.26.10', 'user': None, 'domainName': '6u977749j2.execute-api.ap-southeast-2.amazonaws.com', 'apiId': '6u977749j2' }, 'body': None, 'isBase64Encoded': False } } }</pre>
▶	2021-03-19T01:30:30.121+11:00	[INFO] 2021-03-18T14:30:30.121Z b5efd2d7-46d6-4725-b6b0-aae7f37d5c97 Response JSON:
▼	2021-03-19T01:30:30.122+11:00	<pre>[INFO] 2021-03-18T14:30:30.121Z b5efd2d7-46d6-4725-b6b0-aae7f37d5c97 { "statusCode": 400, "headers": { "Content-Type": "application/json" }, "body": "{\"message\": \"Please provide a period of interest, key terms, or a location\"}" }</pre>
▶	2021-03-19T01:30:30.122+11:00	END RequestId: b5efd2d7-46d6-4725-b6b0-aae7f37d5c97
▼	2021-03-19T01:30:30.122+11:00	REPORT RequestId: b5efd2d7-46d6-4725-b6b0-aae7f37d5c97 Duration: 1.98 ms Billed Duration: 2 ms... REPORT RequestId: b5efd2d7-46d6-4725-b6b0-aae7f37d5c97 Duration: 1.98 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 75 MB

Bibliography

- *Architecture overview — Scrapy 2.4.1 documentation* (2008), <<https://docs.scrapy.org/en/latest/topics/architecture.html>>.
- Jenkins 2020, Jenkins, accessed 4 March 2021, <<https://www.jenkins.io>>.
- Sacolick, I 2020, *What is CI/CD? Continuous integration and continuous delivery explained*, InfoWorld, accessed 4 March 2021, <<https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html>>.
- *Social Network for Programmers and Developers* (2021), <<https://morioh.com/p/eabed717fa86>>.
- Swagger n.d., *Swagger Petstore*, Swagger, accessed 1 March 2021, <<https://petstore.swagger.io/#/>>.
- Amazon 2021, accessed 5 March 2021, <<https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-export-api.html>>.
- Sans 2020, accessed 5 March 2021, <<https://gerard-sans.medium.com/create-a-rest-api-integrated-with-amazon-dynamodb-using-aws-amplify-and-vue-5be746e43c22>>.
- SmartBear Software 2021, accessed 5 March 2021, <<https://swagger.io/blog/api-development/introducing-the-amazon-api-gateway-integration/>> .
- Retz 2021, accessed 4 March 2021, <<https://rapidapi.com/blog/insomnia-vs-postman-vs-paw/>>.
- *What is Python? Executive Summary* (2021), <<https://www.python.org/doc/essays/blurb/>>.