SENG3011 T1 2021

Design Details

Team: Kool Kats

Mentor: Mirette Saleh

Team members:

Lachlan Stewart

z5207906

Pericles Telemachou

z5214738

Stephen Comino

z3470429

Joanne Chen

z5207491

Weiting Han

z5210581

1. Describe how you intend to develop the API module and provide the ability to run it in Web service mode

Overview of How We'll Develop the API Module

Our plan for developing the module will consist of the following series of steps. Since we're a group of 5 we'll be able to implement some of this functionality concurrently by assigning tasks to different group members but some functionality will have to be completed sequentially:

1. Plan and document the API requests that our system will support in Stoplight.
2. Create an AWS EC2 instance running Ubuntu and implement a Python server.
3. Setup an AWS Database and connect with that database on the EC2 instance using the Python SDK.
4. Create Lambda functions for scraping and loading data from WHO. Call these Lambda functions via the Python SDK on the EC2 instance.
    a. Schedule the web-scraping lambda functions to occur periodically (exact period is still undecided) and cache the results from WHO (in case WHO becomes unavailable).
5. All of the API request handling will be implemented on the EC2 instance which will be running a Python Flask server to respond to the API calls.

Web Service Mode and API Module Architecture Overview

Our API module architecture will consist of several key components including frontend, backend and database functionality that will enable it to run in web service mode for users. We decided to go "all in" on using Amazon's AWS services for our application and have modelled our architecture accordingly:

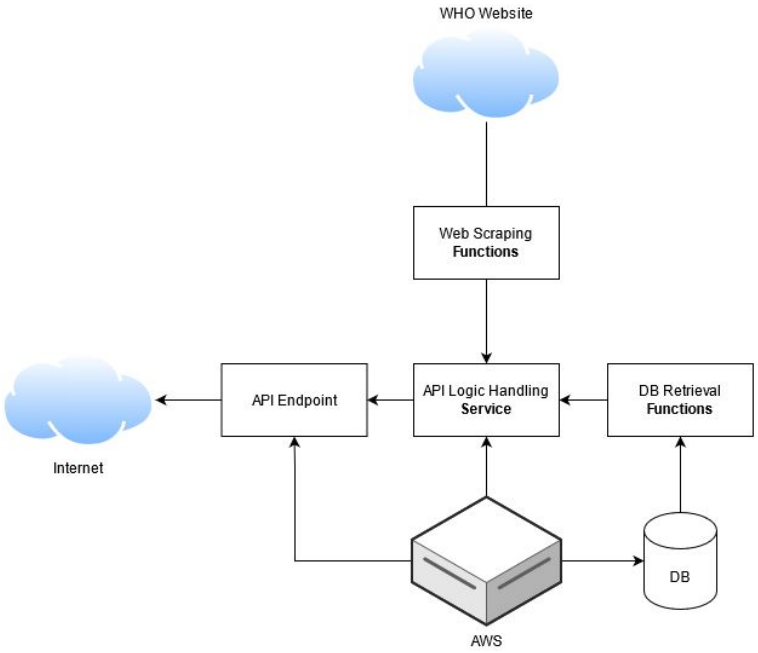
- Stoplight for designing the REST API and documenting it
- An Amazon EC2 instance (running Ubuntu)
- A Python Flask server running on the EC2 instance to handle incoming API requests
- Lambda functions that trigger when an API request is made handling:
    - Web scraping through a python library (Selenium or BeautifulSoup)
    - Amazon DB interaction with functions for submission and retrieval from the DB
    - API servicing logic using lambda functions
- Amazon's Cloud database solution for storing any web-scraped data and/or data POSTed by users

We also considered an alternative architecture which didn't use as many of the AWS services:

- Stoplight for designing the REST API and documenting it
- A python flask server, publicly hosted on AWS to handle incoming API requests
- A python backend system that handles:
    - Web scraping through a python library (Selenium or BeautifulSoup)
    - DB interaction with functions for submission and retrieval from the DB
    - API servicing logic using python functions
- A MongoDB database to store any web-scraped data and/or data POSTed by users

However, the team ultimately voted in favour of the AWS centric architecture as we felt it would make our API module's design more cohesive and easier to manage by using AWS services as much as possible.

At a high level our application would be setup in a similar way to the below diagram:



Proposed application architecture diagram

Link: https://app.diagrams.net/#G1Y1KMzLPMlLzZrrGqsApp2yWlAqc6mjBT

2. **Discuss your current thinking about how parameters can be passed to your module and how results are collected. Show an example of a possible interaction. (e.g.- sample HTTP calls with URL and parameters)**

How Parameters Can be Passed to Our Module

At the moment we're planning to outline the valid "key-value pair" parameters for each request in our API documentation. These pairs can be appended to the end of the URL using a `?` and then separated by using the `&` symbol in the standard format for API requests. That is:

`https://examplesite.com/api`**`?num=1&string=hello`**

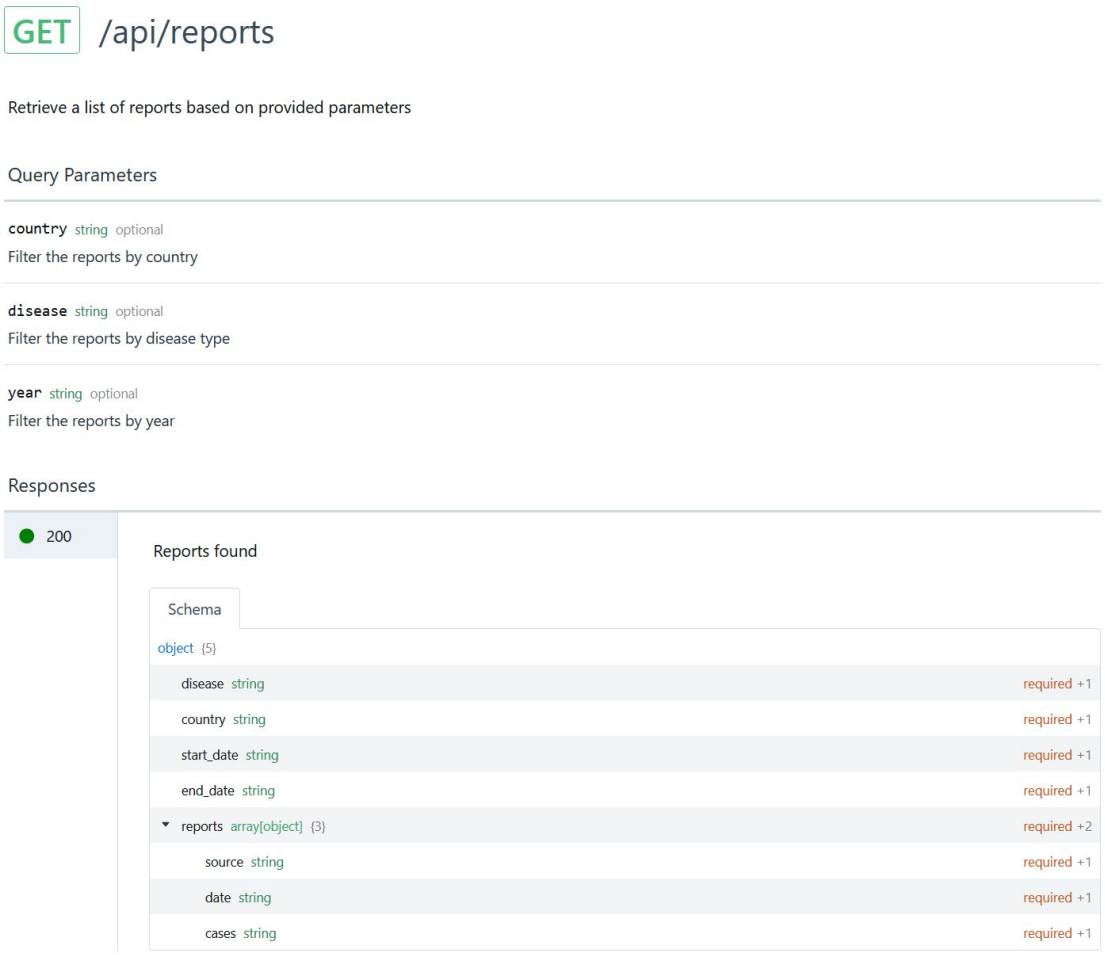
This can be done by formatting the URL like the above example or the user can use a tool like Postman to input the data using a graphical interface.

How Results are Collected

Given the current scope of the assignment we'll just be returning a JSON object. We think the nicest way of formatting that data for our users is to return it in the JSON format (for more standard data formatting and easier integration into existing programs). We'll outline how returned data will be structured for each API request in our API documentation.

Interaction Example

A client looks at the api documentation to find the request they need to make to GET reports from the database.



They then make the GET request to the API to collect data about Coronavirus cases in Sydney for the year 2019:

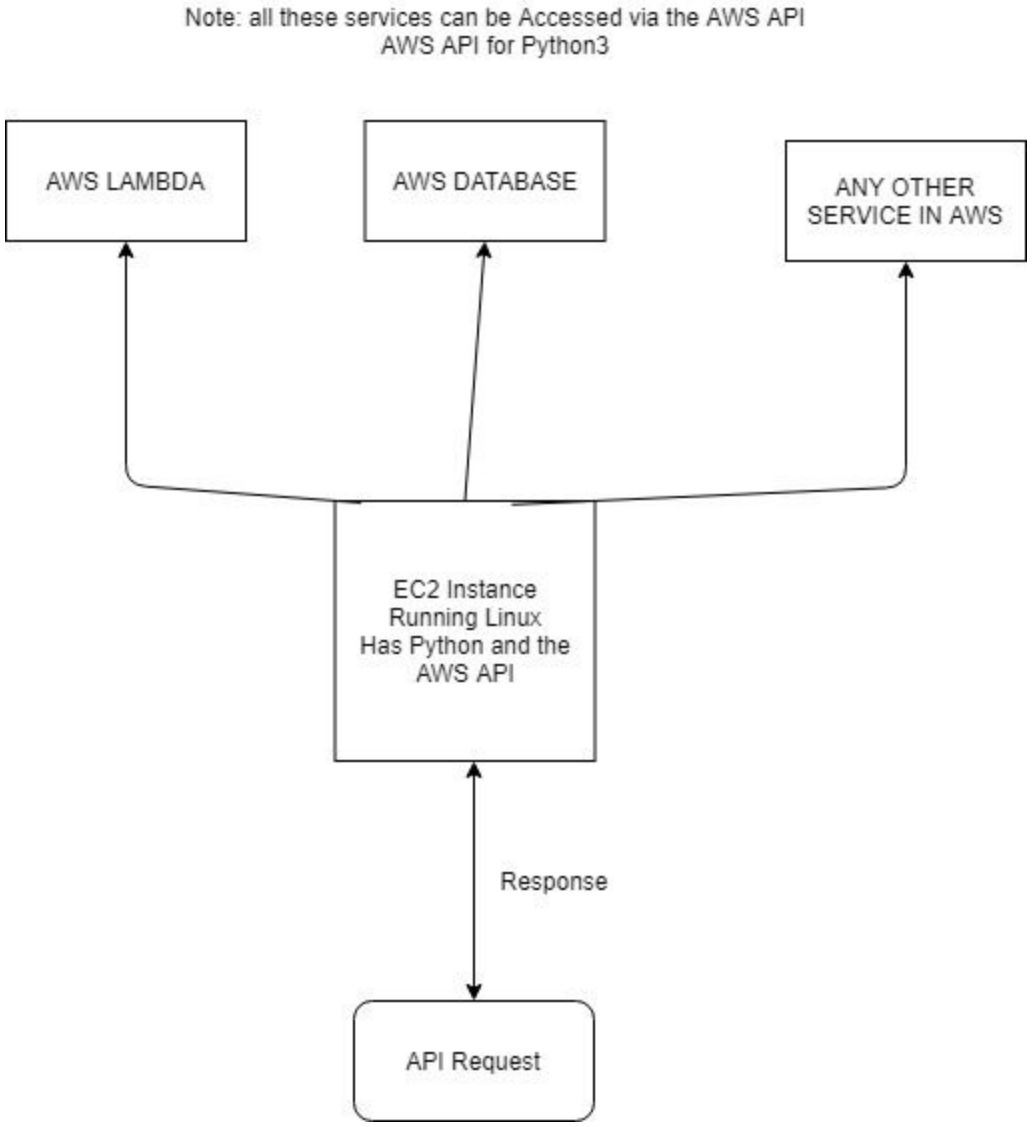
https://example.com/api/reports**?disease=COVID&country=Sydney&year=2019**

The API processes the three parameters outlined in the request, runs the logic on the server that determines whether to get the data from the database or the source (WHO) directly, fetches the data for the user, formats it as a JSON object and sends that object back in the body of the response to the user:

```
{
      “disease”: “Coronavirus (COVID-19)”,
      “country”: “Sydney”,
      “start_date”: “01-01-2019”,
      “end_date”:  “31-12-2019”
      “reports”: [{
            {
                  “url”: “example.com/reports/1”
                  “date”: “15-03-2019”,
                  “cases”: “52”,
            }
      }]
}
```

The user can then parse the response on their end and integrate the data into their project.

3. **Present and justify implementation language, development and deployment environment (e.g. Linux, Windows) and specific libraries that you plan to use.**



Operating System

We plan to host our server on a Linux Operating System (Ubuntu).

We find Linux to be the easiest server OS to maintain due to the availability of open source software compared to Windows. There is also a wide variety of documentation and tutorials for hosting a server on a linux machine.

Deployment Environment

We plan to use AWS (Amazon Web Services) to host our Server. We will use an EC2 Instance to host our Ubuntu API server.

By using AWS EC2 instance we can create a server that can be accessed from anywhere without hosting it on our local machine. This takes away all the issues of hosting the API locally.

We can access the hosted server via a dynamic IP address that AWS gives us. We can also apply for this address to become static, so we have an address that does not change. This way people can use our address as an entry point for accessing our API and not worry about it changing.

Development Environment

We can develop our API either on the AWS instance or locally and port the code over. We will have to make sure that the libraries that we are using are cross platform compatible if we want to develop on a Windows machine. We will utilise the AWS Python3 SDK (BOTO3) to communicate to different AWS services.

For example BOTO3 can be used to create/destroy/run lambda instances and access the database.

We will most likely use the IDE vscode or another widely available text editor.

AWS Services:

AWS services provide a database and lambda functions which we can access via an API.

AWS Lambda Functions

AWS lambda functions allow us to run code without creating a server to host the code.

- By using lambda functions we can scale our product very well. We will have the capability to handle a high volume of requests and reply quickly.

AWS Database

By hosting our database with the AWS Database service we are able to reduce the time needed and the complexity involved in hosting our own sql database.

AWS provides a simple, easy to use API that lets us query and interact with the database.

These services can help simplify our code and organise it in a structure that can be understood better. By using AWS services we can make our product more easily scalable and manageable.

Programming Language:

We have chosen Python3 for the development of our server because it is simple to use and there are extensive libraries and modules available for it that are well documented.

We will write the API in Python3 using the Flask module and use the Python3 SDK for AWS (BOTO3) to create and maintain Lambda instances and connect with the AWS database.

Python3 FLASK for API server:

https://pypi.org/project/Flask/

Justification:

Flask is well documented and straightforward to use. Most of our team has experience using this module.

Python3 SDK for AWS : BOTO3

https://boto3.amazonaws.com/v1/documentation/api/latest/index.htm

Justification:

The AWS API gives us the power to create/destroy instances of AWS components via a server. With this module we will be able to control every other AWS instance

Web Scraping:

We will scrape the data off of the WHO website using either the python3 module Selenium or Beautiful Soup.

Python3 Selenium:

https://pypi.org/project/selenium/

Python3 BeautifulSoup 4

https://pypi.org/project/beautifulsoup4/

Justification:

Both Selenium and BeautifulSoup seem to be the most popular options for web scraping using Python. We plan to try each one before we commit to using one for the rest of the project.

Overall

As well as these specific libraries we plan to use a wide variety of modules found in the Python3 Standard Library. These modules cover everything, from list manipulation to complex data structures and algorithms.

Frontend Plan

Programming Language

Node js is the language we are going to use to create the frontend server. We will also use the ReactJS framework.

We chose ReactJS as the frontend framework because it is popular, has lots of online support and documentation, and because it is the framework that we are most familiar with.

We chose Bootstrap css because it is well documented and easy to use. There is lots of example code for us to use online.