SENG3011 - Software Engineering Workshop 3

Design Details

Team megAPIxels: Rubin Roy, Lachlan Fraser, Humza Saeed, Austin Walsh, Sam Thorley

Mentor: Aditya Kishore

Table of Contents

API Development Process

API Architecture

For this project, we have developed a web API to format and return articles and disease reports from the ProMED website. We will have a Python scraper that will extract article data from the ProMED website to be stored in tables in a web-based SQL database. Accordingly, the database parser will return data in JSON form at the request of our API. Both the scraper and parser scripts will be run from an Amazon lambda function. The API itself will be RESTful and designed using Amazon API Gateway. The API will be called by external client applications as well as by our own application that will use ReactJS as our frontend framework and retrieve information from external APIs as well. This architecture is displayed diagramatically in Figure 1 below.

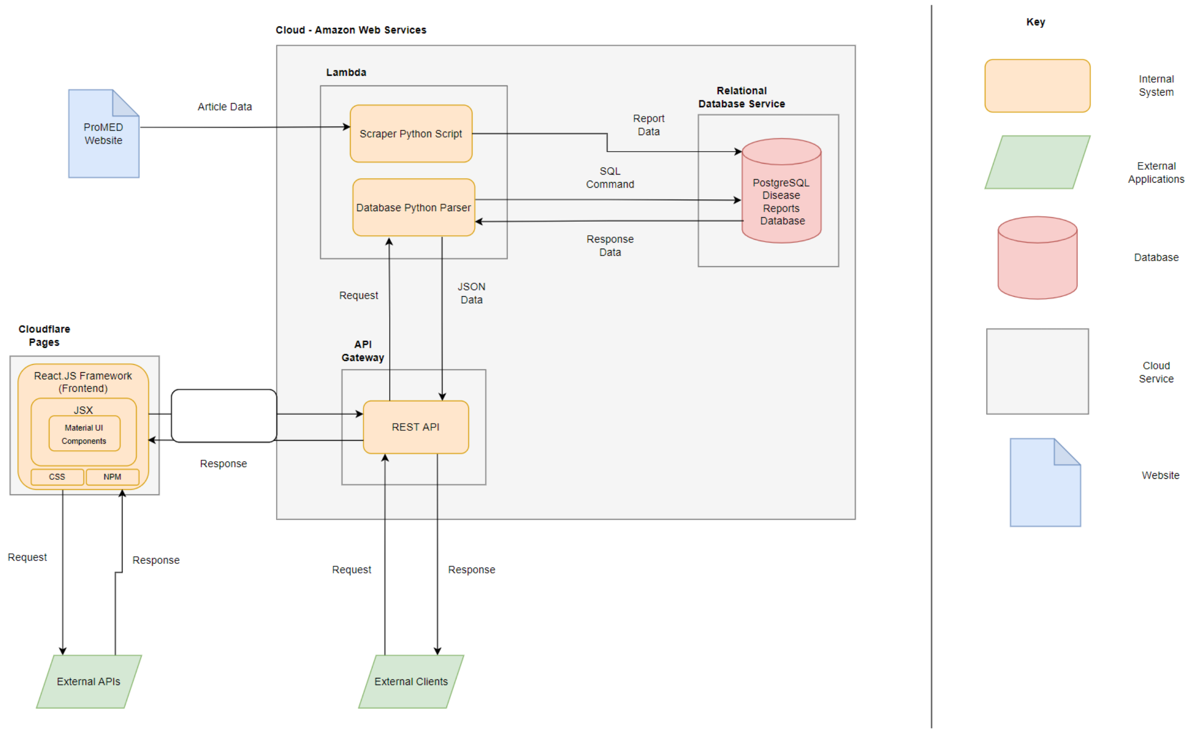

Figure 1: Architecture Diagram

REST API Design Principles

In order for users to interact with our API smoothly and efficiently, we wish for them to be able to pass query parameters and obtain valid results as easily as possible. For this reason, we have chosen to create a RESTful API that will adhere to a number of common design principles. This includes the use of well-known HTTP methods, meaningful endpoint names, versioning, and providing examples.

HTTP Endpoints

Firstly, all API endpoints will utilise common HTTP methods and responses. Namely, since the user is simply using our API to retrieve information from a database, each of our four endpoints use a GET request and receive URL query parameters to define the search. The response status codes returned by our API also align with the HTTP standard for codes such as 200 to indicate a successful response, 400 for a bad request or 404 for a non-existent resource.

The endpoint names themselves have also been designed to be logical and meaningful so that they simply describe their function. An example of this is the "/proMedApi/search" endpoint which the user interacts with to search for relevant information. To ensure that our API is URL friendly we will also omit special characters like +, &, = and ? from our query parameters.

Versioning

Additionally, the API's design will employ URI versioning to cater for the inevitable changes to the system. For example, this may include changes in the response type or user request parameters to be used to extract reports. We will use the "v[x]" syntax where the x represents an integer indicating the version of our api for example "/proMedApi/v1".

JSON Responses

The requests handled by the API will return JSON responses containing information extracted from the backend database that holds the disease data scraped from the ProMED website. We are using JSON responses as they are simple and widely used. Once the API has received its parameters, it will perform a search and filter on the database of standard format disease reports. This database will be constructed from web scraping and each disease report will be stored in JSON format. On a successful call, the API will therefore return a list of standard format disease reports as a list of JSON objects. On error, this list will only contain one element, a JSON object describing the error that occurred. In either situation, an appropriate HTTP status code is also returned to help the user interpret the response. The user will be able to easily collect results from the API in this JSON list format to extract for use in their own application.

Documentation and Examples

To further aid the ease of use of our API, we have provided a number of sample HTTP calls and responses in Table 1 below. Additionally, some common errors and their responses are displayed in Table 2. These examples and more are also provided as extensive Swagger documentation. This documentation outlines the structure of our API in a clear, graphical format. Additionally, Swagger also acts as a simple client to call our API and the user is able to view the sample input and responses in order for them to get a better idea of how to use the endpoints themselves.

Web Service Mode and Deployment

The disease reporting API module we are developing will be created using the Amazon API Gateway and will inherently be using an Amazon web server for web service. Initially, we will determine the requirements of the API which is targeted towards external consumers or developers who want to integrate it as part of their application. The functional requirements have been stated in the project context however in a non-functional sense it is expected that the API will have a fast response time and reliable uptime which will be facilitated by Amazon Web Services. User privacy and system security is inherently dealt with by Amazon due to the reliance on their web services.

Following this, we have designed an API that follows RESTful principles as outlined above. As well as being simple for users, REST is a widely accepted standard with predefined HTTP requests that can be made from various browsers. Information scraped from the proMED website will be stored in a web-based database, again utilising Amazon Web Services. For development, this database can be integrated with our API Gateway.

Accordingly, we will need to test our API for which we will develop test scripts that cover a range of possible inputs and ensure they are handled correctly. We need to test the API to ensure that it will function in different scenarios such as when a high number of GET requests are called in a given period of time.

Finally, we will deploy our API onto the cloud due to its low cost and low management benefits. During web service mode, the API can be called through the Amazon API Gateway which allows for one million API calls per month in the free tier. Once further scaling is required, upgrading to a higher tier will be necessary to handle increased calls and API caching would also be implemented for improved API execution performance.

API Interaction and Examples

API Endpoint Description

Search

Our API will perform initial web scraping to develop a large database of disease reports. From then, we will conduct additional scraping every night to update our database and ensure that each API call will result in the most up-to-date information possible. Therefore, there are only a few endpoints that the user is required to interact with. The most important of these is the search endpoint which is represented by an HTTP GET request to the path “/proMedApi/search”. This endpoint is vital because it will provide the main functionality for our API. Specifically, this will allow users to input their search parameters and be presented with a list of relevant disease reports that they can then use to analyse data trends and make predictions about future epidemics. Example usage of this endpoint is shown in Table 1.

This search endpoint requires at least three input fields: the period of interest, key terms, and location. Since the period of interest must contain a start date and an end date, these fields are encoded in the API URL pattern as the four parameters “start_date”, “end_date”, “key_terms”, and “location”. The start_date and end_date parameters must be of the format “yyyy-MM-ddTHH:mm:ss” while the key_terms parameter is a comma-separated list (which may be empty) of common search words. There is an optional fifth input parameter “timezone” which may also be set in the same way but will do nothing if left blank. If provided, this parameter will help the API to further analyse the database of disease reports and return results that will be more valid.

Pagination

Additionally, if the user performs a very broad search, it is possible that there will be hundreds of matching disease reports. In this scenario, we will implement pagination of responses through the backend so that the user does not load all results at once. This involves the use of a further “page” parameter. By default, this will be 1 and return the first 25 search results. However, when this parameter is increased, the next page of 25 results will be shown instead. For example, if the page was set to 4, the API would return search results 76 to 100, the fourth set of 25 matches. If the parameter is increased beyond the number of relevant matches, only the last 25 results will be returned, regardless of input.

Additional Endpoints

Additional endpoints are included for users to retrieve a specific disease report or article. These endpoints are also HTTP GET requests and require a valid ID corresponding to either a report or article. For these endpoints, only a single JSON object is returned either containing the report or article itself or an appropriate error message. Examples of successful and unsuccessful calls to these endpoints, “/proMedApi/report” and “/proMedApi/article”, are also included in Table 1 below.

Finally, we include a “live” endpoint for users to query to determine whether or not the API is currently active. This endpoint will also act as a health check and return the current version of the API as well. This will be useful for users who are making frequent API calls, for example through their own application, and wish to avoid any errors due to the API not being available for use. Again, an example of the “/proMedApi/live” endpoint is included below.

Table 1: Sample HTTP Calls and Responses

Sample Input Parameters	**HTTP Call and URL**	**HTTP Status Code and JSON Response**

Period of Interest: “1st January 2022 - 7th January 2022”
Key Terms: “Measles, Outbreak”
Location: “Sydney”
Timezone: N/A
	GET
	
	/proMedApi/search?start_date=2022-01-01T00:00:00&end_date=2022-01-07T00:00:00&key_terms=measles,outbreak&location=sydney
		200 OK
		[
		  {
		    "url": "www.promed.com/measles-outbreak-sydney-2022",
		    "date_of_publication": "2022-01-05 xx:xx:xx",
		    "headline": "Measles Outbreak in Sydney",
		    "main_text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis elit at leo finibus, a ultricies enim accumsan. Donec elementum sagittis quam, ac dapibus erat aliquam id. Phasellus et libero vitae mauris volutpat dignissim. Cras facilisis dui eleifend viverra sagittis. Cras eu augue vitae elit vestibulum accumsan quis at dolor. Phasellus blandit maximus mi, quis lacinia odio viverra ac. Phasellus dapibus vel nunc eu lobortis.",
		    "reports": [
		      {
		        "event_date": "2022-01-05 xx:xx:xx to 2022-01-05",
		        "locations": [
		          {
		            "country": "Australia",
		            "location": "Sydney"
		          }
		        ],
		        "diseases": [
		          "measles"
		        ],
		        "syndromes": [
		          "outbreak"
		        ]
		      }
		    ]
		  }
		]

Period of Interest: “6:15:20pm 19th November 2019 - 2:50:05am 27th February 2022” Key Terms: “N/A” Location: “Tokyo” Timezone: UTC+9	GET /proMedApi/search?start_date=2019-11-19T18:15:20&end_date=2022-02-27T02:50:05&key_terms=&location=tokyo&timezone=utc+9	200 OK

		[{ "url": "www.promed.com/tokyo", "date_of_publication": "2022-01-05 xx:xx:xx", "headline": "Recent Diseases in Tokyo", "main_text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis elit at leo finibus, a ultricies enim accumsan. Donec elementum sagittis quam, ac dapibus erat aliquam id. Phasellus et libero vitae mauris volutpat dignissim. Cras facilisis dui eleifend viverra sagittis. Cras eu augue vitae elit vestibulum accumsan quis at dolor. Phasellus blandit maximus mi, quis lacinia odio viverra ac. Phasellus dapibus vel nunc eu lobortis.", "reports": [{ "event_date": "2019-12-25 xx:xx:xx to 2022-01-05", "locations": [{ "country": "Japan", "location": "Tokyo" }], "diseases": ["measles", “plague”], "syndromes": ["outbreak", “infectious”] }] }]
Period of Interest: “7th January 2022 - 1st January 2022” Key Terms: “Measles, Outbreak” Location: “Sydney” Timezone: N/A	GET /proMedApi/search?start_date=2022-01-07T00:00:00&end_date=2022-01-01T00:00:00&key_terms=measles,outbreak&location=sydney	400 Bad Request [{ “error” : “start_date cannot be after end_date” }]
Report ID: 10587	GET /proMedApi/report?report_id=10587	200 OK { "url": "www.promed.com/tokyo", "date_of_publication": "2022-01-05 xx:xx:xx", "headline": "Recent Diseases in Tokyo", "main_text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis elit at leo finibus, a ultricies enim accumsan. Donec elementum sagittis quam, ac dapibus erat aliquam id. Phasellus et libero vitae mauris volutpat dignissim. Cras facilisis dui eleifend viverra sagittis. Cras eu augue vitae elit vestibulum accumsan quis at dolor. Phasellus blandit maximus mi, quis lacinia odio viverra ac. Phasellus dapibus vel nunc eu lobortis.", "reports": [{ "event_date": "2019-12-25 xx:xx:xx to 2022-01-05", "locations": [{ "country": "Japan", "location": "Tokyo" }], "diseases": ["measles", “plague”], "syndromes": ["outbreak", “infectious”] }] }
Report ID: N/A	GET /proMedApi/report?report_id=	400 Bad Request { “error” : “not a valid report ID” }

Article ID: 76483	GET
	
	/proMedApi/article?article_id=76483
		200 OK
		{
		"url": "www.promed.com/tokyo",
		"date_of_publication": "2022-01-05 xx:xx:xx",
		"headline": "Recent Diseases in Tokyo",
		"main_text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis elit at leo finibus, a ultricies enim accumsan. Donec elementum sagittis quam, ac dapibus erat aliquam id. Phasellus et libero vitae mauris volutpat dignissim. Cras facilisis dui eleifend viverra sagittis. Cras eu augue vitae elit vestibulum accumsan quis at dolor. Phasellus blandit maximus mi, quis lacinia odio viverra ac. Phasellus dapibus vel nunc eu lobortis.",
		}

Article ID: N/A	GET
	
	/proMedApi/article?article_id=
		400 Bad Request
		{
		“error” : “not a valid article ID”
		}

Article ID: 76483	POST
	
	/proMedApi/article?article_id=76483
		405 Method Not Allowed
		{
		“error” : “invalid HTTP method”
		}

N/A	GET
	
	/proMedApi/live
		200 OK
		{
		“live” : “true”
		“version” : “1.0.0”
		}

N/A	GET
	
	/proMedApi/query
		[
		{
		“error” : “page not found”
		}
		404 Not Found]

Table 2: Possible HTTP Errors and Responses

Error	**Reason Encountered**	**HTTP Response**
Invalid input	The user has incorrectly formatted their URL query parameters or provided invalid input	400 Bad Request
Page not found	An invalid endpoint name has been requested	404 Not Found
Invalid HTTP method	An invalid HTTP method has been requested to a valid endpoint name	405 Method Not Allowed
N/A	Correct API usage	200 OK

Justification of Development Choices

Python

We have decided to develop our API backend in the Python programming language with its included standard libraries, as well as React JS to implement our front end UI design. We decided to use Python as the primary backend language for the API as all of our team members have some level of experience having done previous courses on Python such as COMP1531. In addition, it is relatively simple to work with to build API backends due to its readability, flexibility and versatility as well as having more portability than other open-source languages such as C. This means we can run the same code on different platforms without having to change anything.

Another reason we favoured Python over other programming languages is because of its compatibility with other tools and frameworks. Many of the popular tools and SDKs used to work with different areas of development are built for Python as the primary environment on which scripts are run. According to many sources including northeastern.edu, Python is the most popular programming language used by developers today. Because of this, there is a large active community of developers that use Python, making it easier for them to use our API as part of their development projects.

However, this doesn't mean Python has no drawbacks. A disadvantage of Python is that it isn’t native to Android or IOS, so it would be unsuitable to deploy Python programs on mobile platforms. This did not affect our project however, since we are not deploying our API on an Android or IOS app. It is also slower than compiled languages such as C and C++ and is prone to runtime errors, making it more difficult to debug.

Another programming language we could have chosen was Java. Although Java is also a very popular choice of language among developers, it wasn’t suitable for us because its parent company, Oracle, charges a monthly licence fee for using the Java Development Kit. It also does not support a lot of the libraries that are used in Python.

Python Libraries

There are a number of Python libraries and SDK's we will use to work with our DBMS, web server and UI front end development. To run SQL queries and functions on our backend, we will be using psycopg2. It is the most popular PostgreSQL driver for the python programming language. Psycopg2 will allow us to be able to connect to the database on our python interface and run scripted commands that will systematically extract information that is stored on the database in the desired format.

Another Python library we will be using is boto3, which is used to directly manipulate AWS resources in a python script. Boto3 is a scripting library that is based on the Botocore software development kit. Although botocore precedes boto3, the latter has many benefits over the former that come to use when using collections and resources. So we decided to use boto3 as our AWS library as it is the newest and most popular version of the botocore library collection.

We looked at various web scraping tools to determine a suitable way to conveniently and efficiently extract data from the ProMed website into a collection that is easy for us to use. We decided to go with scrapy as our scraping framework within Python. Among the other tools we considered included Selenium, BeautifulSoup and LXML. We chose scrapy as it allows us to extract and collect data into different formats, including JSON, CSV and XML, providing us greater flexibility. Additionally, scrapy has built in spider features. The use of spiders allows us to scrape the web more efficiently using a set of instructions we can define. A key disadvantage of Selenium over scrapy is that it only begins browsing the web page after it has fully loaded, which can significantly decrease the overall speed of the API especially if the client's internet connection is weak.

To extract reports from the scraped raw text, we will be using multiple libraries in combination. To match key terms we will be using the python regex library, re, to match the text for the provided list of key terms. This is a well-known and used library that is robust and well documented, utilising regular expressions. To extract dates and locations from the raw text, we will be using a library called spacy. This is a very sophisticated library that analyses each word and phrase given in a string for a variety of languages. It will identify words such as 'London', 'New South Wales', and 'Sydney' as locations, as well as dates in a variety of formats. However upon testing we have found the library to be prone to occasionally overmatch dates and make mistakes, so we will also be utilising the dateutil library to parse all matched dates. The parser function of this library will match provided strings in a range of formats into datetime objects. This will allow us to filter out poorly identified dates by spacy and also store dates in a standardised datetime format.

React JS

React is a powerful open-source JavaScript library owned by Meta used to create web based UI’s. It is used in popular websites such as Facebook and Netflix. It is a library providing a convenient way of building aesthetic user interfaces. Using React enables us to easily create dynamic web applications with more functionality. There are many advantages of using React that brought us to use it. Firstly, it was a library that everyone in the team had experience with. This makes it easier for team members and eliminates the need for learning a new UI development library which will save crucial time over the course of the 10 week project. In addition, React enables reusable components which allow complex code to be written in blocks, making the application easier to develop and maintain. React is also significantly faster than pure JavaScript as it uses a virtual DOM which exists entirely in memory, making virtual components easier and faster to run. Only changes are applied to the original DOM, instead of re-rendering everything.

Alternatives

An alternative to React is Angular which is owned by Google. Angular is a cross-platform web development framework which has better performance and is arguably considered more readable and portable than React. Although Angular is a popular choice of framework among web developers, we decided not to use it in our project for a number of reasons. Firstly, as mentioned earlier, not everyone in the team is familiar with angular. It would have been less time efficient for the team to learn Angular or other frameworks within the short 10-week time frame. For this reason, we decided to spare the time in learning Angular and chose React as our option.

Another alternative to React that we considered is Vue, an open-source front end JavaScript framework for designing UI’s. Notable websites made using Vue include Gitlab and Trivago. Unlike React, Vue uses HTML instead of JSX. However, it also uses a virtual DOM just like React. Despite this, both React and Vue present similar performance. Vue enables developers to create websites that are light, fast and efficient and is usually recommended for new developers learning front end UI design due to the decoupling of HTML, CSS and JavaScript. Most of our team members were only familiar with React. Because of this, we decided to go with React due to the time constraints of our project.

React Libraries

Some additional libraries we will be using include Node Package Manager (NPM) which is a global online open-source repository which will be used to manage packages and dependencies for our project. It will be used to install, upgrade and configure packages. It is essential to our project as it will help prevent or resolve dependency conflicts by ensuring all versions of packages are the same to prevent bugs during development.

An alternative to NPM is Yarn. However, NPM and Yarn have few differences and since certain team members are more familiar with NPM, we will not be using Yarn to reduce learning curves for team members in a short timeframe.

Material UI is a React library that will allow us to import and use different components which will form the face of our UI application. These components may include a range of features including but not limited to buttons, lines, scrolls, sliders, drop-down menus, textboxes and much more. It also allows greater use of typographical features such as fonts and text wrapping. By using Material UI, we intend to create a web application that provides an exceptional and satisfying digital experience to users.

An alternative we could have chosen is Storybook, which is an open source component tool. It provides greater functionality to MaterialUI with integrated testing, greater options, and the ability to review components with your team. However, it is much more complicated than MaterialUI with a full desktop application. No team member has also used Storybook before, hence to save time we will use MaterialUI.

Database

For our relational database management system, we will be using PostgreSQL, hosted on AWS RDS (discussed later). This is a system that most team members are familiar with as it is used in the core SENG stream course on database systems. This saves time learning an unfamiliar RDBMS, which is crucial for this project which has very little development time before the deadline. It is also JSON compatible, allowing for greater design freedom later on. Postgres also features a well documented python library for postgres access called psycopg2, which most team members are also familiar with. Postgres is also open source, meaning no financial costs will be required with running database instances. It is also widely used and documented, with frequent updates released. A notable disadvantage of postgres is that it can be slow and unstable for large scale businesses, which we are not. It can also be difficult to update versions. However, since we are a short-term project there is no need to do so.

SQL vs NoSQL

We have chosen to use SQL over NoSQL. SQL as a relational model offers reduced data storage due to normalisation, maximising performance. A drawback of SQL however is it is very rigid once created and changes to schemas often cannot be applied without reloading data. NoSQL does not have this issue as it is non-relational and allows changes to be made while in use. It is however easy to make mistakes due to type and write inconsistencies which must be considered by the users. Since no member of our team has experience using distributed systems or NoSQL, we have decided to stick with SQL due to time concerns.

DBMS Alternatives

One of the SQL alternatives considered was Amazon Aurora. It is highly integrated with Amazon RDS and hence would be much more reliable and easy to use. It also features greater performance and security than Postgres. However, Aurora is not included in the free tier of AWS, meaning costs would be incurred. It is also a new system that no team member has used, so would include a time cost.

Another alternative considered was MySQL. It is very similar to PostgreSQL, so would have a relatively low learning curve, and hence time cost for the team. Just like Postgres, it also features extensive documentation and community support. It also has greater security features compared to Postgres. However, since the database instance will be hosted by Amazon RDS, it will already be isolated from external access. MySQL would also be free to use as it is also open-source. Overall, since there are few additional benefits to using MySQL over Postgres in the case of our project, Postgres will be used due to familiarity.

Finally, MongoDB was also considered. Overall, MongoDB has a considerable learning curve as it does not use SQL. It is relatively easy to set up with cloud services and is very powerful for data analysis. However, MongoDB is a paid service. It offers a free version that could be used for academic purposes, however it’s function is limited. Hence due to the learning curve and reduced functionality, postgres is a more attractive option for our project.

Cloud Services + Deployment

Our API will be hosted on a cloud service. This will allow for frequent scheduled web scraping, as well as a constant, reliable offering of our API. We have chosen to use Amazon Web Services. It is the most popular cloud service, with some team members having experience using it. It has regions all over the world, including Sydney. It has hundreds of services including those for database management, API hosting, and a computing platform. It is also the recommended cloud service provider for the course. It has a free tier which offers limited use of many of its services for free.

Alternatives

One of the main competitors for AWS is Microsoft Azure. It also features a free tier with much of the same services offered. There seems to be little difference between the limits offered by the free tiers of Azure and AWS. Azure however, according to their website, is mainly targeted at Windows and Microsoft software. Since some of our team members use MacOS paired with familiarity with AWS, means Azure will not be used.

The other main competitor is Google Cloud. Their free tier however had significantly lower limits than those offered by AWS and Azure. Since we will be creating an API, with significant data scraping and storage required, we decided to stick with AWS.

Database Service

For the database service, we will be hosting our PostgreSQL database on the Amazon Relational Database Service (RDS). It is designed to be used with Postgres, hence has significant documentation and tutorials for the project. It also allows for strict access controls defined by our team and controlled by Amazon. It offers 750 hours of database usage per month under the free tier as well as 20GB of storage. It also allows for usage of the Postgres command line tool, PSQL.

Computational Service

For the computing platform, we will be using AWS Lambda. Python scripts will be run at set intervals with the use of cron jobs, to scrape the ProMed articles for the required information and storing them in the Postgres database. They will also be set up to fetch from the database when requested by the API. The free tier offers 1 million requests per month, with up to 3.2 million seconds worth of compute time. This should be more than enough for the purposes of our project.

API Host

To host the API, we will be using Amazon API Gateway to host our RESTful API. This will be connected to lambdas which will then fetch the required data from the database. The free tier offers 1 million API calls per month, which should be ample for our project.

Web Application Deployment

The web application that we will develop will be hosted on Cloudflare Pages. This will be integrated with GitHub, with any code pushed to main instantly reflected on our public URL. It is compatible with React and has built in CI/CD which triggers on every code change. It also allows easy testing with preview URLs for every git branch and pull request. There is also a free tier limited to 500 builds per month, which will be more than enough for our project.

Alternative deployment services considered were Vercel, Github Pages, and AWS Amplify. All have free tiers Amplify and GitHub Pages were considered as they could have had better integration with the services already used. However, AWS is not used for the frontend, so would not be better integrated. GitHub Pages also did not allow for development sites. Cloudflare also allows for multiple users in the free tier, which Vercel does not. For these reasons, Cloudflare will be used to deploy the web application.

Monitoring

We will be using Amazon Cloudwatch to monitor our API and Lambda services. This gives an auto generated, detailed, and regularly updated log which can be referred to. Since it is integrated with Amazon, it can be activated very easily. The only other alternative considered was a non-cloud based log file which would be updated every time the API or lambda functions were called. However, this is not only harder to set up, but also less detailed and more likely to fail.

Challenges

Throughout the development of our API so far, we have come across a few challenges that we have had to overcome in order to produce a quality product. In particular, these included gaining familiarity with Amazon Web Services, searching and utilising a number of new code libraries and performing successful web scraping from the ProMED website. Additionally, time spent resolving these issues meant that not all of our API functionality was up and running by the submission of this deliverable. These shortcomings are also described in this section.

AWS

Firstly, we experienced some difficulty getting used to the many web services offered by Amazon. For many of us, this was the first time that we were creating an API that would be deployed on the internet for other people to use. Therefore, after taking a look at a few different deployment platforms, we settled on Amazon Web Services. This prompted us to also take a look at their other tools to help streamline our development.

However, despite the convenience offered by using services such as API Gateway, Lambda and Elastic Beanstalk, this took quite a while for us to set up and understand. Specifically, we had issues with Amazon's permission system. As one of us was the root user and the rest of us were only given a subset of privileges, this proved to be a stumbling block throughout the development process as we would have to constantly rely on each other to complete simple tasks.

Despite this, we eventually found ways around these issues by manually modifying and adding new privileges to other group members so that we were able to work more independently.

Code Libraries

The development of our API included many complex code aspects that many of us had not experienced or used before. Namely, these included detailed web scraping, location and date matching as well as intensive database integration. For many of these tasks, we decided to use some code libraries found online to help us so that we weren't writing all of this code from scratch.

However, in doing so, we introduced further complexity for ourselves by attempting to integrate a number of external libraries together into the one program. In particular, we had difficulty installing some of these packages and ensuring that all necessary dependencies were operational and up to date. We then needed to deploy these complex code packages to an AWS lambda function which proved to be a further challenge due to the need for complex zipping and uploading. Despite these challenges, the benefits of using code libraries for this task certainly outweigh the trouble we would have faced if we were attempting to write code for each aspect of the project ourselves.

Web Scraping

An additional significant challenge faced in the completion of this API was the web scraping process. This was required to retrieve article data from the ProMED website in a way that could be easily formatted and returned by our API. Our team only had limited web scraping knowledge and past experience had been based on simple websites. Therefore, we found it quite challenging to scrape data from the ProMED website which is complex and designed largely as a single-page application.

This prompted us to explore some other options for retrieving website data including Selenium, BeautifulSoup and LXML. However, we found that these tools were unable to return data in the formats we were after. Therefore, we persisted with the use of the Python library scrapy to scrape ProMED in more of a traditional manner. In addition to this, ProMED's access limitations prevent fast scraping by returning a 403 and denying access to the website. We tried to utilise free proxies to bypass this however it was unsuccessful, hence a single thread approach with request auto-throttling was used to minimise the number of 500 responses. Consequently however significantly lengthening scraping time at around 20 pages a minute.

Due to the early difficulty that was encountered in this area, we were unable to complete all of the necessary web scraping for this deliverable. Instead, only about one week's worth of articles has been added to our database. Furthermore, we did not have time to find a way to extract valid reports from content due to the initial time it took to develop the scraper. However, we were able to extract the diseases, syndromes, event dates and locations from the raw text which we placed in a single report object linked to an article in the database.

Incomplete Features

Since we were spending a lot of time on this project learning new skills and technologies and overcoming difficulties as mentioned above, we ran out of time to implement all aspects of our API exactly as we would have liked. In particular, we were unable to satisfy datetime data type constraints or implement pagination. It is our intention to restore full functionality of these sections as soon as possible so that users will have access to the full features of our API.

Firstly, although the project specification outlines that all datetime formats should be "yyyy-MM-ddTHH:mm:ss", we were experiencing a great amount of difficulty in parsing data in this form. This was due to two main factors. The data being scraped from the ProMED website itself was often not in this convenient format. Instead, articles may outline dates and times in words such as "17th March 2022". We found this natural language difficult to parse into a logical data structure. Similarly, our database structure and interaction required a specific form of datetime input that would not allow for time of day. For these reasons, we have modified our search endpoint input parameters to only accept start and end dates in the "yyyy-MM-dd" form. All other functionality surrounding this input is still working as expected.

Additionally, we ran out of time to implement a pagination system for our search endpoint. As described in the API Interaction and Examples section above, we had planned to include pagination to improve our API’s response time and latency while not bombarding the user with too much data. However, this feature proved to be too much for us with the time constraints enforced for this project and has unfortunately been left out for the time being. Fortunately, the impact of this deficiency has not been too great since, as mentioned in the Web Scraping section above, our database is not yet active at its full capacity.