Design Details Report

Epidemic Eagle

Design Details

Prepared for SENG3011

Created by Aaron Shek, Sahibpreet Bassi, Daniel Steyn, Liam Treavors and Mihail Georgiev

Context

In recent times, the COVID virus has affected the entire world, due to the rapid infection rate of the virus. But since vaccines have been developed, minimal effort has been made to prevent future outbreaks.

One solution to this issue is to notify government facilities when a potential outbreak is predicted to happen. Resources could then be arranged and handed to medical centres to prepare and reduce the spread of an outbreak.

One method to achieve this solution is to read news articles about reports on diseases, and broadcast a message if patterns of an infection around an isolated area exist.

Although manually web searching reports would solve this issue, it would be very labour intensive and time-consuming to collect and process all the data. That is why we have designed an API that will automate and summarise these articles and reports into uniform data which can be viewed easily.

API Model

For the routing of the API, we have thought about which routes to provide to the user. GET requests send information such as article bodies and will be a required route. But other requests such as POST and PUT are not included, since users will not need these functionalities. Additionally, admin routes will not be needed since we plan to generate and provide data for our API automatically through the use of a web scraper.

Web Scraper

For our API the web scraper will process numerous websites such as ProMed, then collate the relevant data and return it to the user. The web scraper will also automatically update or delete outdated reports since it continually rechecks websites. Since a web scraper is used for our API, this means POST, PUT and DELETE requests will not be included.

Caching Data

Web scraping is a powerful tool, but has the drawback of high computational time. So if the web scraper is called every time a GET request is made to the API, it would also be a pain point to the user experience. This is why our team has decided to cache data from the web scraper to send to the user, and instead intervally run the web scraper. This will, in turn, lead to rapid interactions with our API which benefits the user. However an issue with cached information is outdated information. The cache will be updated every 4 hours from the web scraper, to ensure the information is up-to-date.

Response and Storage Objects

Data from the web scraper will be stored in a database, as a JSON Object. This was favourable over such types as XML and due to compatibility with programming languages.

Data that is sent from our API will also be in a JSON format, with a supporting status code such as 200 or 404. This was done since it reflects the industry standards and conventions.

Pagination / Filtering

An issue that affects users is the heavy loading of a singular route, such as getting all reports from a certain date. That is why our API will include pagination to reduce the load on the user as well as server usage. Our current plan is to limit results to 10 items, which then page_number will be passed as a path parameter to access further pages.

Filters such as keywords and symptoms of a disease will also be incorporated, for similar reasons stated above.

Versioning the API

Our API will continually be developed and pushed in 3 phases.

However if our API is updated, some functionalities of old routes may be preferred over new versions of the same route. This is why our API will include different versions of our API, via changing the route call such as “/v1/api/” to “/v2/api/”.

Web Hosting

Hosting our API locally on a local server is extremely expensive and involves a lot of manual labour to start up and maintain. That is why we decided to choose an external service which will host our API instead of personal local hosting.

Design Details

Parameters

For our API we will use path parameters to pass in information such as filters.This is comparable to a JSON body, but errors will be easier to detect in a path compared to body. The internals of our API are described below.

Route	Method	Params (required)	Optional Params	Response	Definition

/api/articles	GET	start_date: <string::date>
		end_date: <string::date>
		key_terms: <string>
		location: <string>
			page_number:
			<string>
				{
				articles : [list of 10 <object::article>s],
				num_pages : 1,
				page_number : 1,
				}
					gets all articles in the dates submitted
					can be filtered by key_words and location

/api/articles/{:article_id}	GET	article_id:<integer>		{<object::article>}	gets article data (all reports from an article)

/api/reports	GET	start_date: <string::date>
		end_date: <string::date>
		key_terms: <string>
		location: <string>
			page_number:
			<string>
				{
				reports : [<object::report>],
				num_pages : 1,
				page_number : 1
				}
					gets all reports from dates.
					can be filtered by key_words and location

/api/reports/ {:report_id}	GET	report_id:<integer>		{<report>}	gets singular report data

/api/search	GET	start_date: <string::date>
		end_date: <string::date>
		key_terms: <string>
		location: <string>
			page_number:<string>	{
				results: [list of 10 <object::search>s],
				num_pages : 1,
				page_number : 1,
				}
					gets shortened results in the dates submitted.
					can be filtered by key_words and location

Model	JSON Object
article	{ url: <string>, date_of_publication: <string>, headline: <string>, main_text: <string>, reports: [<object::report>]

	}

report	{
	diseases: [<string>],
	syndromes: [<string>],
	event_date: <string>,
	locations:
	[
	country:<string>,
	location:<string>
	],
	}

search	{
	article_id:<string>,
	url: <string>,
	date_of_publication: <string>,
	headline: <string>
	}

Error Model	Status Code	JSON Response

Item not Found	404	{
		"message":{
		"The item with given
		id was not
		found."
		}
		}

Validation Error	422	{
		"detail": [
		{
		"loc": [
		"string"
		],
		"msg":
		"string",
		"type":
		"string"
		}
		]
		}

Result Gathering (REDO)

We will run the web scraper at a constant time interval every 4 hours. At each interval, the web scraper will read all articles from multiple websites such as ProMed, and break it down into articles and reports based on keywords within the body of text. It will then store these articles and reports within our Postgres database. Articles will store the main body of text, the headline of the original article, the url and the publication date of the article, while reports will store the disease, the location, the syndromes and the reported date of the illness. Reports will also link to the article they originated from, and can only link to that one article.

The user of our API won't see any of this scraping however, when they are searching for certain diseases, the API will query our database, and find matches on disease, location and time periods, without interacting with ProMed itself. This will greatly reduce waiting time for users, as searching a database will be much quicker than scraping through articles and then comparing key terms. Matching entries in the database will then be returned to the user as a list of JSON Objects, with each Object being the requested type of data, such as an article or report.

Sample HTTP Calls

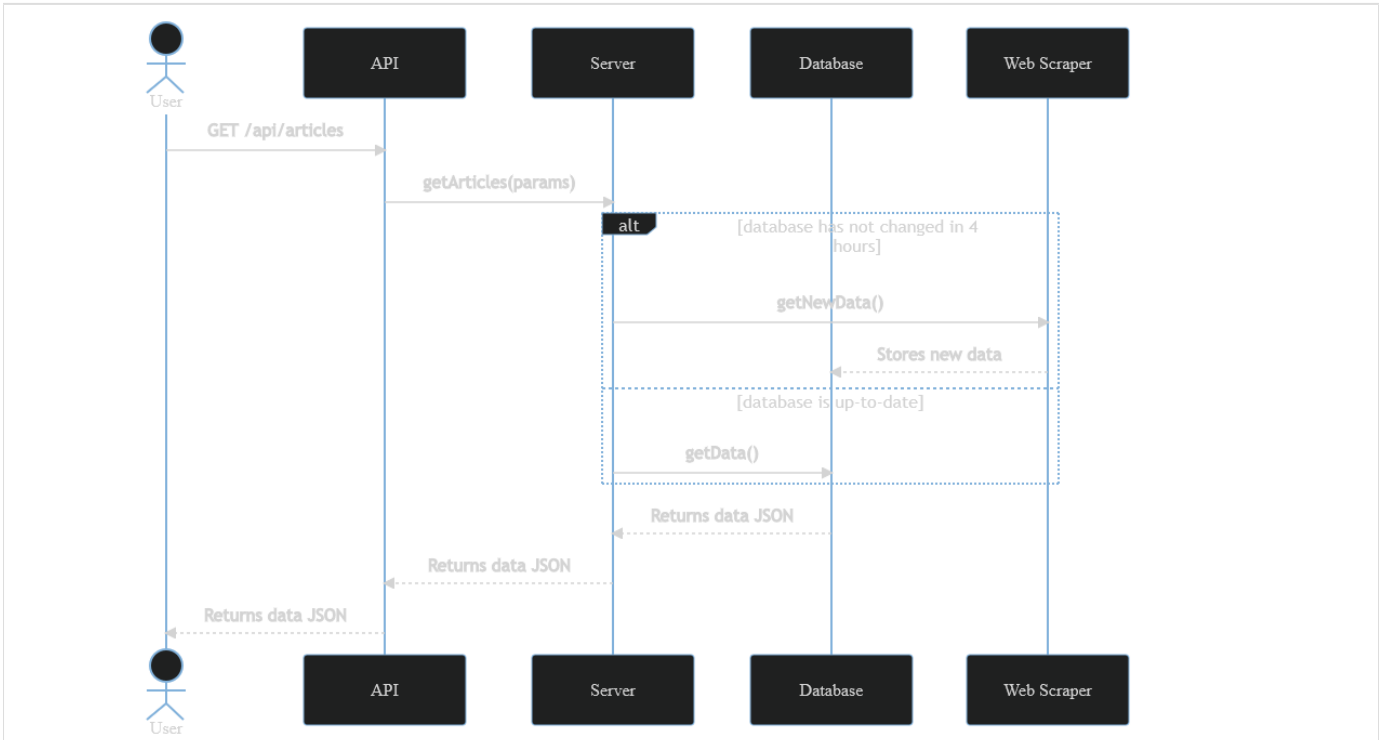
Side Note

For the sample HTTP Calls, errors have not been shown. This has been intentional since it would present repetitive information about our routes. One route has been fully documented named '/api/articles/1'.

In order to visualise our API, sample HTTP calls have been added below.

Request : /api/articles/?start_date=2018-12-12&end_date=2018-12-13&key_words=influenza&location=Vietnam

Response:



{ "articles": [{ "url": "www.who.int/lalala_fake_article", "date_of_publication": "2018-12-12", "headline": "Outbreaks in Southern Vietnam", "main_text": "Three people infected by what is thought to be H5N1 or H7N9 in Ho Chi Minh city. First infection occurred on 1 Dec 2018, and the latest is reported on 10 December. Two in hospital, one has recovered. Furthermore, two people with fever and rash infected by an unknown disease.", "reports": [{ "event_date": "2018-12-01 to 2018-12-10", "locations": [{ "country": "Vietnam", "location": "Ho Chi Minh" }], "diseases": ["influenza"], "syndromes": ["Acute fever and rash"] }] }], "page_number": 1, "page_count": 1}

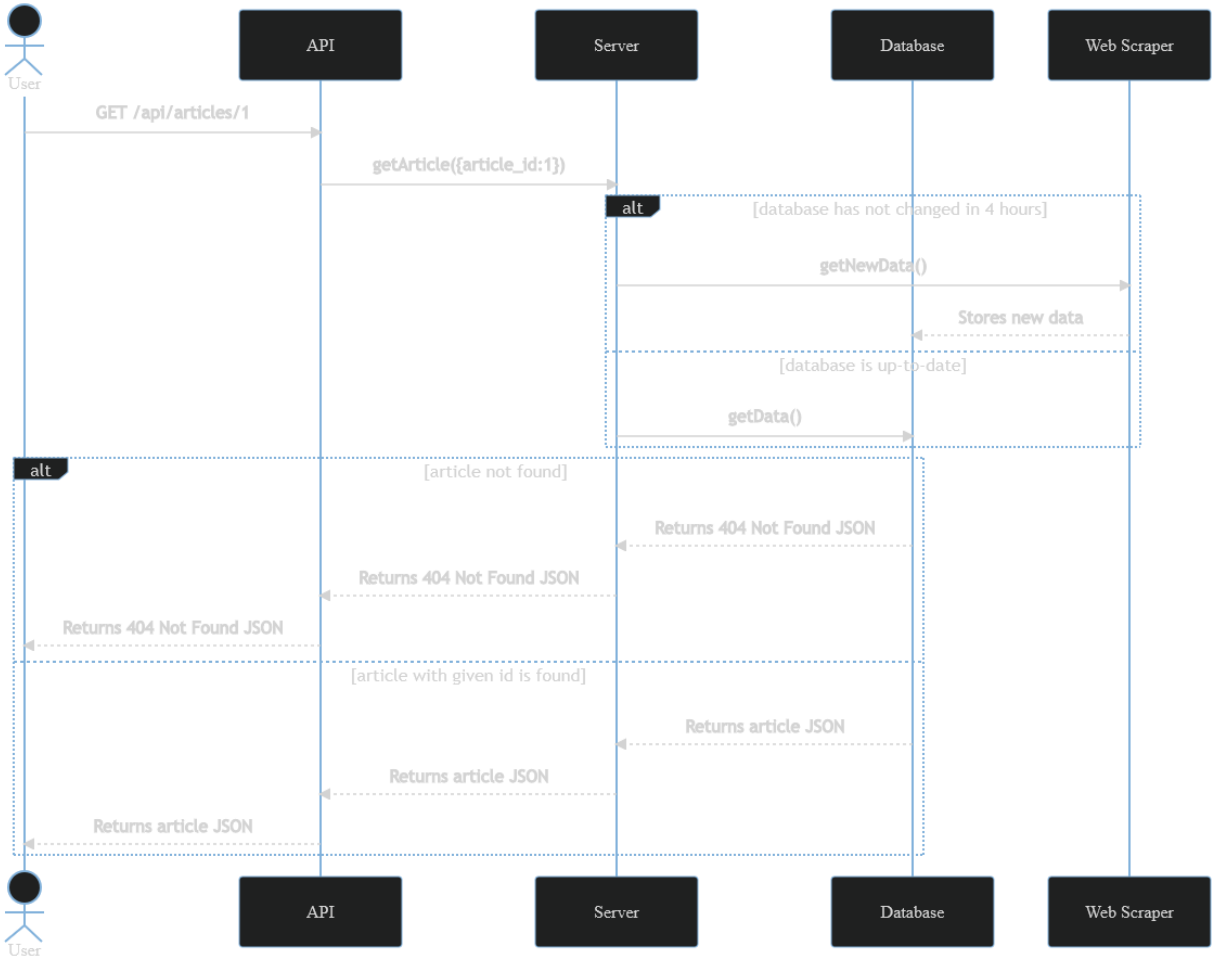
Request : /api/articles/article_id=1

Response:

{ "url": "www.who.int/lalala_fake_article", "date_of_publication": "2018-12-12", "headline": "Outbreaks in Southern Vietnam", "main_text": "Three people infected by what is thought to be H5N1 or H7N9 in Ho Chi Minh city. First infection occurred on 1 Dec 2018, and the latest is

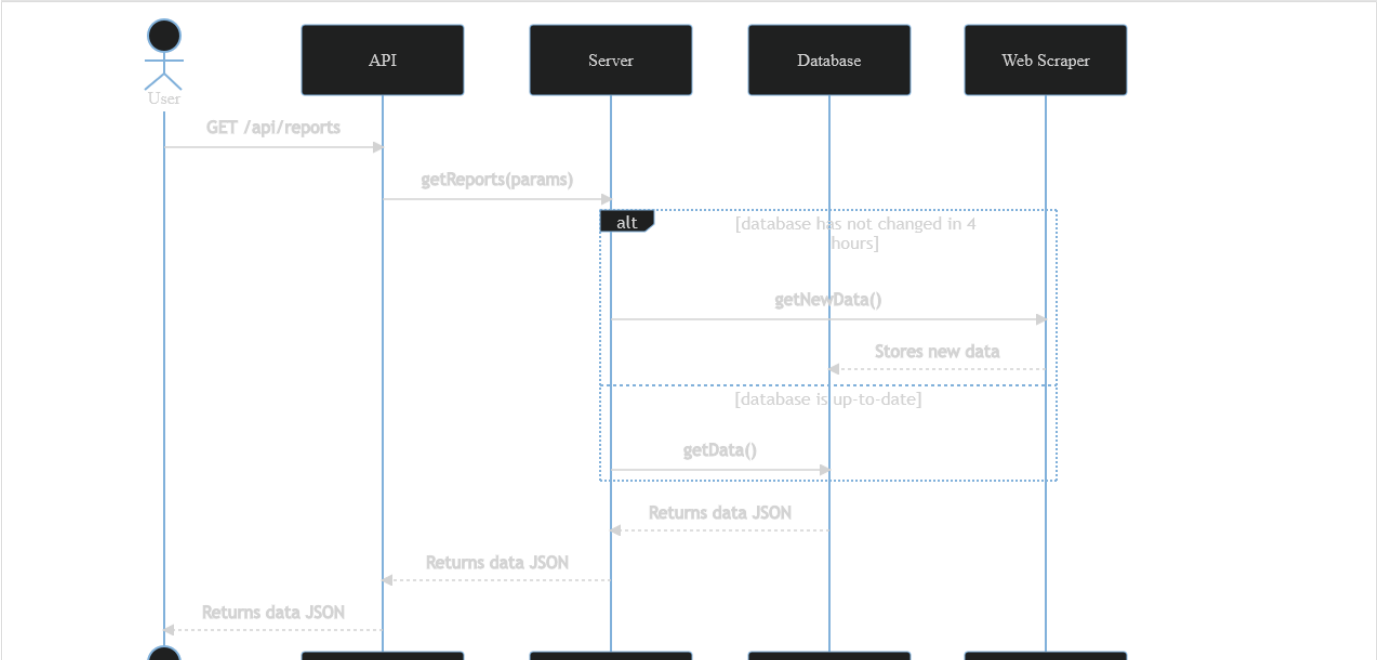
reported on 10 December. Two in hospital, one has recovered. Furthermore, two people with fever and rash infected by an unknown disease.", "reports": [{ "event_date": "2018-12-01 to 2018-12-10", "locations": [{ "country": "Vietnam", "location": "Ho Chi Minh" }], "diseases": ["influenza"], "syndromes": ["Acute fever and rash"] }]

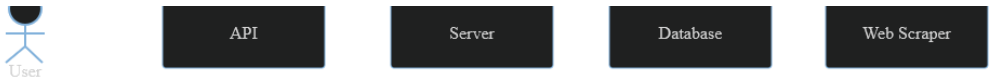
}



Request : /api/reports/?start_date=2018-11-12&end_date=2018-12-12&key_words=ebola&location=Democratic Republic of Congo Kivu

Response:



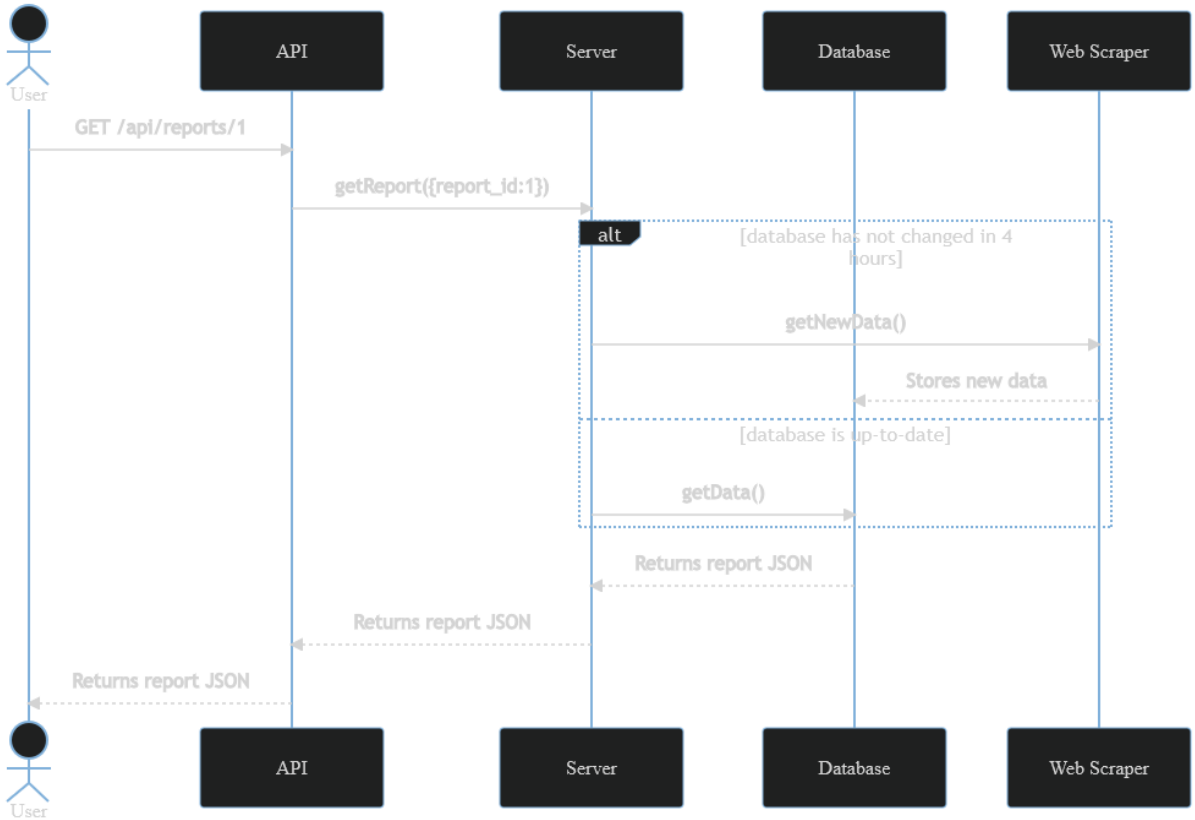


{ "reports": [{ "event_date": "2018-11-13 to 2018-11-20", "locations": [{ "country": "Democratic Republic of Congo", "location": "Kivu" }], "diseases": ["Ebola"], "syndromes": ["Fever and headache", "Abdominal pain", "Unexplained haemorrhaging"] }, { "event_date": "2018-12-04 to 2018-12-10", "locations": [{ "country": "Democratic Republic of Congo", "location": "Kivu" }], "diseases": ["Ebola"], "syndromes": [] }], "page_number": 1, "page_count": 1}

Request : /api/reports/?report_id=1642

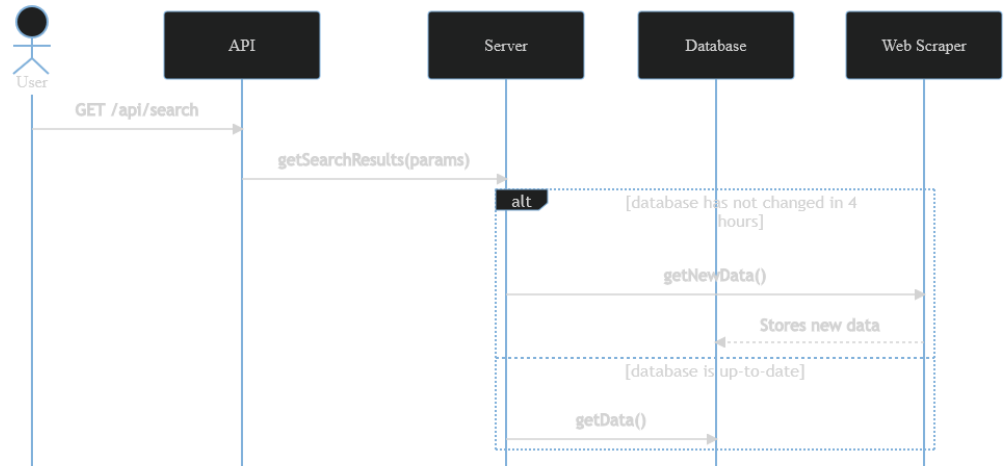
Response:

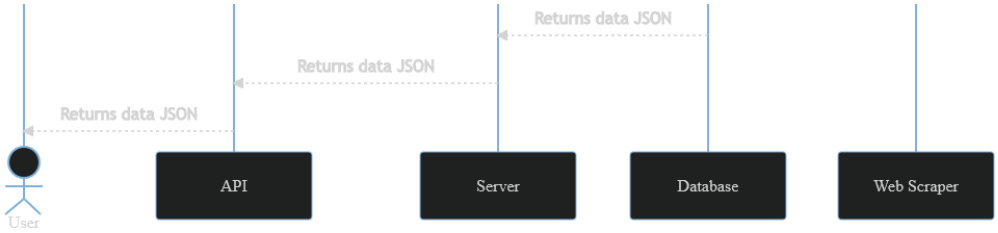
{ "event_date": "2018-11-13 to 2018-11-20", "locations": [{ "country": "Democratic Republic of Congo", "location": "Kivu" }], "diseases": ["Ebola"], "syndromes": ["Fever and headache", "Abdominal pain", "Unexplained haemorrhaging"]}



Request : /api/search/?start_date=2019-02-10&end_date=2022-02-10&key_words=measles&location=Afghanistan

Response:





{

"results": [

{ "article_id": "1809", "url": "www.who.int/emergencies/disease-outbreak-news/item/measles-afghanistan-paktya", "date_of_publication": "2022-01-08", "headline": "Measles Outbreak in Paktya" },

{ "article_id": "1756", "url": "www.who.int/emergencies/disease-outbreak-news/item/measles-afghanistan-balkh", "date_of_publication": "2021-08-14", "headline": "Measles Outbreak in Balkh" },

{ "article_id": "1429", "url": "www.who.int/emergencies/disease-outbreak-news/item/measles-afghanistan-kunduz", "date_of_publication": "2019-05-27", "headline": "Measles Outbreak in Kunduz" },

{ "article_id": "1398", "url": "www.who.int/emergencies/disease-outbreak-news/item/measles-afghanistan-zabul", "date_of_publication": "2019-04-18", "headline": "Measles Outbreak in Zabul" }],

"page_number": 1,

"page_count": 1

}

Development, Implementation and Deployment

API's Development

For our API, we had to make several choices on how the API was structured as well as what dependencies are needed such as libraries or external services.

To start we decided to focus firstly on what structures / systems we wanted for our API.

Database System

Since our API requires a database in order to serve information to the user, we thought about the benefits and drawbacks of NoSQL vs Relational Databases (SQL).

Feature	NoSQL	Relational Databases
Schema	Dynamic schema, this makes changes the predefined schema simple and inexpensive	Predefined, structured schema. Altering the schema can be very expensive
Data Structure	Unstructured data with no relations. Data is stored in singular tables with all information about a record stored in that table	Highly structured data stored in a predefined schema. Data that is added must match the schema
Scaling	Horizontally scalable. Inexpensive to expand with multiple low cost servers	Vertically scalable. Can prove a challenge as scaling becomes quite expensive - it requires adding/upgrading resources for your existing machine
Performance	Capable of handling even the biggest datasets	Not as performance efficient for massive datasets
Procedures/Functions	No stored procedures, difficult to identify patterns	Plenty of stored procedures to understand the data and its patterns. Functions can be made to select specific data from multiple tables.
Availability/Consistency	Databases are not always consistent, which can be confusing for the user. (Trades consistency for partition tolerance - CAP Theorem	Availability depends on the server. Data is consistent. (Trades off partition tolerance for availability and consistency - CAP Theorem
Rigidity	Data can be inserted in any form and is not necessarily in the correct format	Data is guaranteed to be in a specific format. Entries stored in predictable structure
Analytics	Better suited for real-time analytics	Better suited for analyzing past datasets

Team Experience	Our team is inexperienced in NoSQL databases, both in theory and application. Time would need to be dedicated to learning the software	Our team has considerable experience in relational databases and no time would need to be spent learning the software

Obvious drawbacks of RDBMS in comparison to NoSQL are scaling and performance. The cost of scaling can be expensive as it can only be done vertically and RDBMS performance does not scale as well as NoSQL does as the dataset grows. However, we feel these drawbacks are mitigated by the benefits they provide.

All factors considered, the datasets we will be working with are more suited to a structured, predefined schema. Our records are very predictable and will always match the schema. Analyzing past datasets is simpler, there are plenty of functions and procedures to manipulate the data to find patterns and the data is always consistent. Given all these reasons, and our team's experience with relational databases - and inexperience with NoSQL - an RDBMS is the most logical option.

Web Hosting Services

Since hosting a server locally is expensive and has heavy startup costs, we needed an external hosting service for our API. In this case we compared Infrastructure as a Service (IaaS) to Platform as a Service (PaaS). Although we are comparing different types of services, we decided to use real examples as these hosting services vary depending on the host. In our comparison we compared AWS IaaS and Heroku PaaS.

Article	Heroku	Amazon Web Services (AWS)
Service	Platform as a Service	Infrastructure as a Service
Initial Production	Infrastructure is provided by Heroku and fast to deploy.	AWS requires manual setup of the environment, being a relatively complicated process.
Traffic	Heroku can handle small/medium amounts of traffic.	AWS scales with internet traffic and can handle very high loads.
Cost	The cost of hosting on Heroku is average, and increases greatly for large applications.	AWS is extremely cost effective, as the service is a PAYG model.
Monitoring	Includes an application monitoring system with error responses.	No monitoring system.
Database Management	Heroku has an extension named “Heroku Postgres” for database	Databases are manually set up by the user, but can be chosen to fit the application.
Requirements	Heroku provides a ready-to-use environment.	AWS requires prior knowledge of Unix.

Ultimately, our team decided to go with Heroku as our web hosting service, which is a PaaS. It is not a perfect solution as it is known for high inbound and outbound latency, lower network performance and it is less suited for heavy-computing. Given our circumstances, however, we feel it is the best option for our project. Creation of a server is a much simpler process than on AWS, it offers rapid deployment with a ready-to-use environment, minimal downtime during system updates and a more startup-friendly learning curve. It also provides a monitoring system and a database manager. Heroku requires no infrastructure maintenance, which is very desirable for such a small team as we can dedicate more team members to development tasks. Given all the advantages listed, we are willing to take a slight hit on performance as we feel we will be able to produce a better product with Heroku as we can focus more on development.

Programming Language

Python was chosen as a main language for our API. Although other programming languages with web hosting exist such as Ruby on Rails and JavaScript NodeJS, Python was the only programming language which everyone had experience with coding.

Python was also chosen for the reasons below:

- Python is a flexible language in the sense that it does not restrict developers from developing a certain sort of application
- Memory allocation is not required, compared to languages such as C
- It is a popular tool for data science and analytics
- There are hundreds of Python libraries to use, which are easily imported compared to languages like Ruby which does not have custom imports
- It is an efficient, reliable and fast language that can be used in multiple environments such as mobile and desktop applications, web development, hardware programming and more.
- Python has a huge community for support and multiple issue fixes, saving time compared to lesser known like Ruby on Rails.

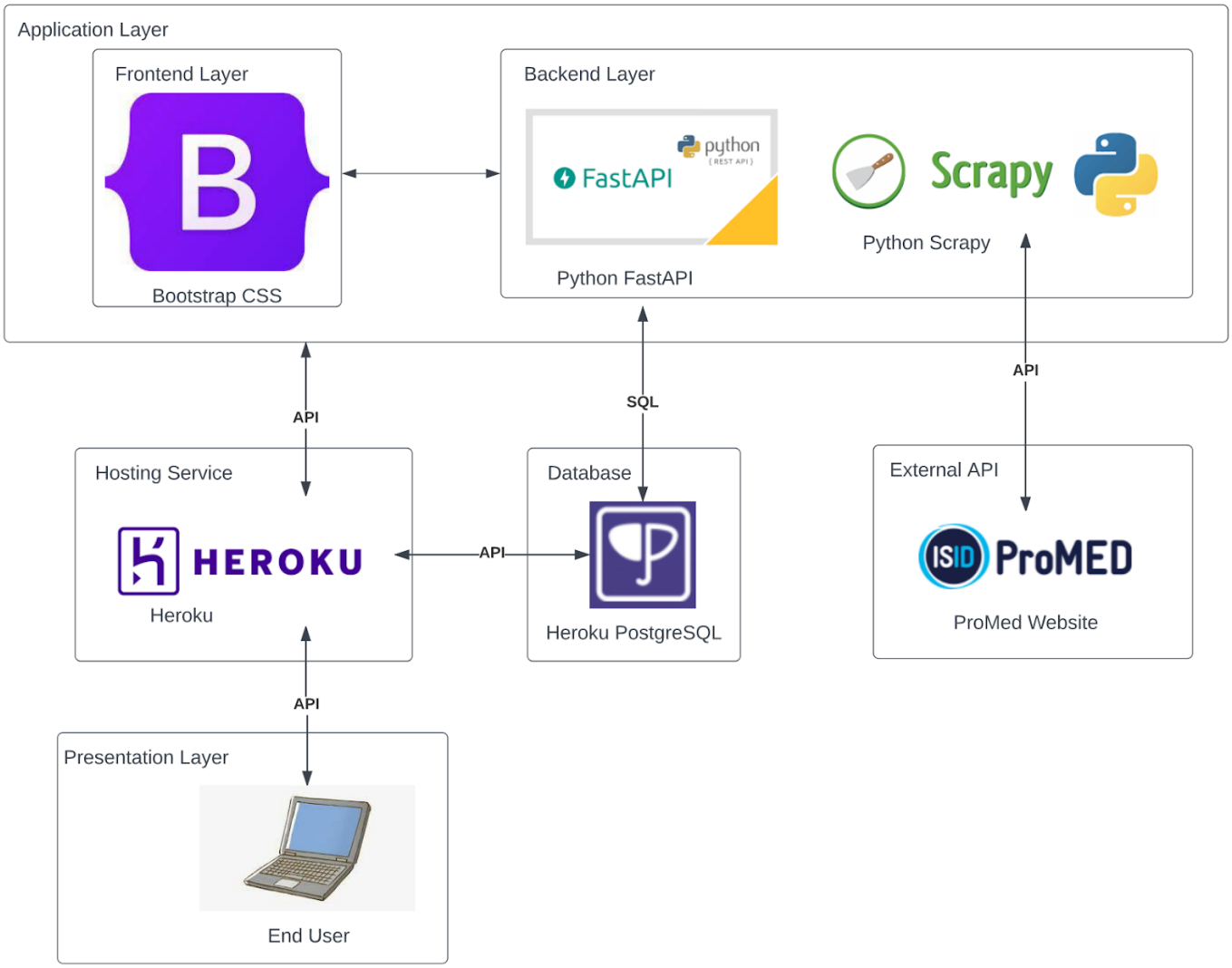
Python also proves useful when working with JSON data. Compared to languages like Ruby on Rails or other web hosting languages, Python has inbuilt functions that can treat dictionaries like JSON Objects and vice versa. Consequently our team can reduce the time spent coding and more time on testing and designing an end product.

Implementation and Software Architecture

For our EpidemicEagle API’s development, our team has chosen the following for its tech stack:

- The ‘Scrapy’ web crawling framework
- The ‘FastAPI’ web framework
- The ‘PostgreSQL’ RDBMS hosted by Heroku Postgres
- Heroku, a cloud service platform
- The Linux Operating System

The tech stack for the API has been visually created below along with the justification for these choices.



Justification of Software Architecture

Back End Software

Scrapy

As our API is automated, we needed a web scraper that fit our personal needs and also created in python

We compared Selenium, BeautifulSoup and Scrapy.

Article	Scrapy	Selenium	BeautifulSoup
Extensibility	Scrapy is designed for middleware to be customisable allowing for custom functionality. It is easily integrated into a data pipeline.	Selenium is the best choice when dealing with Javascript.Selenium is also beginner friendly.	Beautiful Soup helps with maintaining code simple and flexible. Ideal for low to mid level complexity projects.
Performance	Scrapy has a high performance because it is able to utilise asynchronous system calls.	Selenium is similar in performance to Beautiful Soup however is not equivalent to Scrapy.	

			Beautiful Soup can be quite slow when performing certain tasks however this issue can be overcome through the help of multithreading.
Ecosystem	Scrapy has a good ecosystem, it can use proxies and VPNs to automate the task and send multiple requests from multiple proxy addresses.	Selenium has a good ecosystem for development but it is unable to utilise proxies easily.	Beautiful Soup has a lot of dependencies in the ecosystem making it not ideal for a complex project.

Despite there being numerous options for web scraping, Scrapy was the better choice for our team for the following reasons:

- In comparison to Selenium and Beautiful Soup, Scrapy is the best for dealing with large projects where efficiency and speed are required.
- It is also asynchronous, unlike the other two, meaning it can make requests in parallel rather than one at a time, and it is memory and CPU efficient as a result.
- It is a 'complete' web scraper framework with built in crawling, whereas Selenium for example is a parsing library online that manually needs to be built in an infinite loop for crawling.

The only drawback this scraper may have is its learning curve, but Scrapy is clearly powerful enough to outweigh this.

FastAPI

For our choice of web framework when it comes to developing APIs, we discussed whether to use Flask, Django or FastAPI. We chose FastAPI mainly due to the ease of use and adaptability but some other comparisons are briefly explained below:

- is based on the Pydantic library also built on Python, for parsing and validating data (which Flask does not have). This is useful when it comes to enforcing type hints at runtime.
- It is built on the Asynchronous Server Gateway Interface (ASGI) specification, which means multiple asynchronous requests are handled easier than Flask or Django.
- is based on the open standards for APIs.
- has flexibility code wise, and doesn't restrict the code layout, unlike Django.
- automatic swagger doc creation, unlike Django or Flask

Overall Django, Flask and FastAPI in comparison have differences that are negligible in the end on a practical analysis. However FastAPI was best when it came to the overall development time it would take to produce a product.

Heroku PostgreSQL

Selecting a server to host our database is an important decision. A server poorly equipped for our application's demands will negatively impact both us as developers and the user experience. Conversely, it is possible to overestimate the demands of our application; this would lead to us only utilizing a fraction of the capabilities of our server and overpaying for what we're getting. Our server needs to be reliable, secure, scalable and cost effective.

Feature	Heroku PostgreSQL	PostgreSQL Server	PostgreSQL Server on Dedicated Server
Cost	Flexible. The lowest tier package is free, the standard package is $50/month. Package can be upgraded as required	None	Either cost of building, running and maintaining a dedicated server or cost of renting a virtual private server
Performance	Decent across all packages. Standard provides 4GB of RAM. Low downtime tolerance, high performance ceiling	Limited by local machine resources. Not reliable for external users - more suited for developing and testing. Not considered stable for the end user	Scales with cost - the more you pay the more you get. High performance ceiling
Scalability	Fully scalable as required	None, can only alter the amount of resources available to the server on the local machine	Fully scalable, can scale up with additional resources as required, can become costly
Security	Highly secure, compliant with industry standards (PIC and HIPAA). Can use own encryption key	Responsibility is on the developers to maintain the database/server's security	Responsibility is on the developers to maintain the database/server's security
Setup	Setup is fairly simple and well documented	Relatively simple to set up locally, more complicated to set up for external use	Quite challenging to set up the infrastructure
Uptime	A minimum uptime of 99.5% with the lowest package and 99.9% with the standard package	Associated with local machine's availability and network stability	Using a dedicated machine: same as Postgres Server VPS: dependant on company hosting from
Drawbacks	Hosted on Heroku so will fail if Heroku fails	Very limited in performance	Maintaining a dedicated server requires lots of time and expertise

As one might extrapolate, a PostgreSQL Server hosted on a local machine is inadequate for our application's demands - it would simply hinder us too much and result in an unstable, unreliable server for the user. Where a locally hosted server lacks the scalability and performance, a dedicated server more than makes up for. It is the most flexible option, allowing vast scalability and a high performance ceiling. The server can

be customized to improve performance by altering memory management, caching, buffer pools and more. However, this comes at a cost. The amount of time required to set up and maintain this server is enormous and would require an immense amount of expertise. As our team is entirely inexperienced with network security and server maintenance, it would be illogical to choose this option. We would be out of our depth and would undoubtedly encounter more problems than avoid. It's for this reason we have gone with Heroku PostgreSQL.

The service we have access to is the same service Heroku itself uses. There are a plethora of package options and upgrading is simple, meaning we can grow our server as our application and database grows.

Additional features:

- Feature-rich: offers useful features like forking and follower instances.
- Reliable: downtime tolerance of less than 4 hours per month at the lowest package, less than 1 hour per month with the standard package and any package above is less than 15 minutes per month (99.98% uptime).
- Rollback: offers a 4 day rollback with the standard package.
- Size restrictions: standard package has no row limitations.
- Storage capacity: standard offers 64GB of storage, but the package can be upgraded to increase this.
- Performance metrics: database metrics available for analysis.

All this makes it the ideal choice for our application.

Front End Software

Bootstrap

EpidemicEagle's front end options came down to choosing from ReactJS, Bootstrap and Vue. We compared the three frameworks on a number of features for our final decision inside a table below.

Article	ReactJS	Bootstrap	Vue
Performance	Fast, lightweight Component based architecture means more robust apps, reduced code clutter, reduced DOM manipulation Component based architecture significantly makes loading pages faster, and more pleasing UX.	Criticised for slower performance Offers extensive customisation features to increase performance, despite being content heavy Workarounds and best practices can pull the time to load 1.3 MB sized pages to 2.1 seconds.	Performance can worsen with each new feature added, or each component extended. Uses lazy loading to improve runtime.
Application Architecture	No built in architecture View layer application made up of components, Render underlying UI as data changes.	Supports Model-View-Controller architecture pattern Bootstrap's role in the MVC is the view component	Uses a 'ViewModel' approach, in a Model-View-ViewModel pattern. The 'ViewModel' handles synching between Model and View parts of code. The view is the entry point to an application rather than the controller. Business logic is decoupled from UI
Ease of Testing	Quite easy to test, offers test runners E.g. Jest, Mocha test suites Reduced time to market Speeds up testing and productivity.	Needs external plugins for testing apps E.g. Chrome Developer Tools, DesignModo, BrowserStack This external testing eliminates cross browser bugs and checks app's consistency and uniformity	Uses Jest, Mocha, Chai or Vue Test Utils.
Scalability	Yes React is purely javascript traditional methods, virtual DOM and component reusability can make a React app scalable	Yes Class components, interactive grid system	No. Significant work (forking a Vue application and using multiple repositories) is required.
Suitability for Building Complex Apps	Suitable Builds highly interactive single page apps. Uses external server side rendering architecture such as Redux, Flux, Next.js	Suitable Mobile first framework means an app will be supported on various platforms, devices, screens with ease. broad range of ready-made components, templates, themes for use. 12 column grid system useful for complex web layouts across platforms.	Preferred for developing small-time applications.
User Experience	More likely to be adopted by users. High quality UI	Responsive design Consistency across all platforms	Creates an interactive and eye-catchy UI.

	Fast rendering When built correctly, React apps deliver great performance. Frequently updated.	Apps made in bootstrap are always trusted and valuable because of this consistency. Can enhance UX through external UI kits.	
Rapid Development	Requires in depth understanding of router system, configuration specifics, and lots of Javascript experience	Quick development time due to the following: extensive documentation, developer friendly, doesn't require much coding knowledge, libraries consist of several class components, only requires HTML and CSS, ready made components, JS plugins, cross browser compatibility.	Quick development time due to the lightweight nature of Vue and add on tools and rendering features.
Application Size	Not a full featured frontend framework; it is simply a library. App sizes can be quite large.	Varies depending on content, need to be wary of library packages and their unused components. Following best practices can minimise the overall size.z	Vue is a lightweight framework. Application size is not heavy.
Learning Curve	If developers have JavaScript experience, then it takes little time to learn. Uses API structures and data flow in Javascript. Only versions after 16.0 are up to date.	Developer friendly, very little learning curve when prerequisites of HTML and CSS are covered.Effort might be needed in learning CSS classes and Bootstrap components. Extensive documentation exists, there is a large community of developers online for learning guides.	Easiest to learn out of the three choices, basics can be covered in less than a week. Requires basic understanding of ES6 functions, and fundamental JavaScript skills. Documentation is understandable but not rich.

There were considerable pros and cons for each but Bootstrap stood out for the following reasons:

1. Our team has chosen a rapid development time as a priority.
2. The learning curve is small for Bootstrap, and our team is not very experienced / does not have time available to spend learning and mastering JavaScript. Focusing on HTML and CSS means we can spend more time being productive, rather than worrying about learning.
3. Bootstrap still has potential to produce quality apps.
4. Our application will not require much complexity.

Further Justifications

Some choices which cannot be shown in the software architecture diagram have been added below.

PyTest

For testing our API, two choices were considered: Unittest and Pytest.

	Pytest	UnitTest
Description	Popular open-source testing framework, third-party dependency	Default python testing framework that comes with the python package
Team Experience	Major	None
Test Types	Function based testing. Tests are independent functions.	Method based testing. Tests are sub-methods of a defined class.
Assertion Cases	Builtin assertions	Builtin assertions
Test Results	Highlighted text with code snippets and error message	Provides line and class errors.
Test Structure	Tests are independent of each other	Tests are created inside a class.
Platforms	Python Only	Python and Java(JUnit)

Although UnitTest is an integrated feature of python, we decided to use pytest mainly due to team experience.

Summarizing other reasons why we chose of Pytest over Unittest:

- Our team has more experience with Pytest than UnitTest
- Pytest functions are independent functions instead of inside of a class
- Test errors are clearly highlighted based on function compared to UnitTest displaying a line error

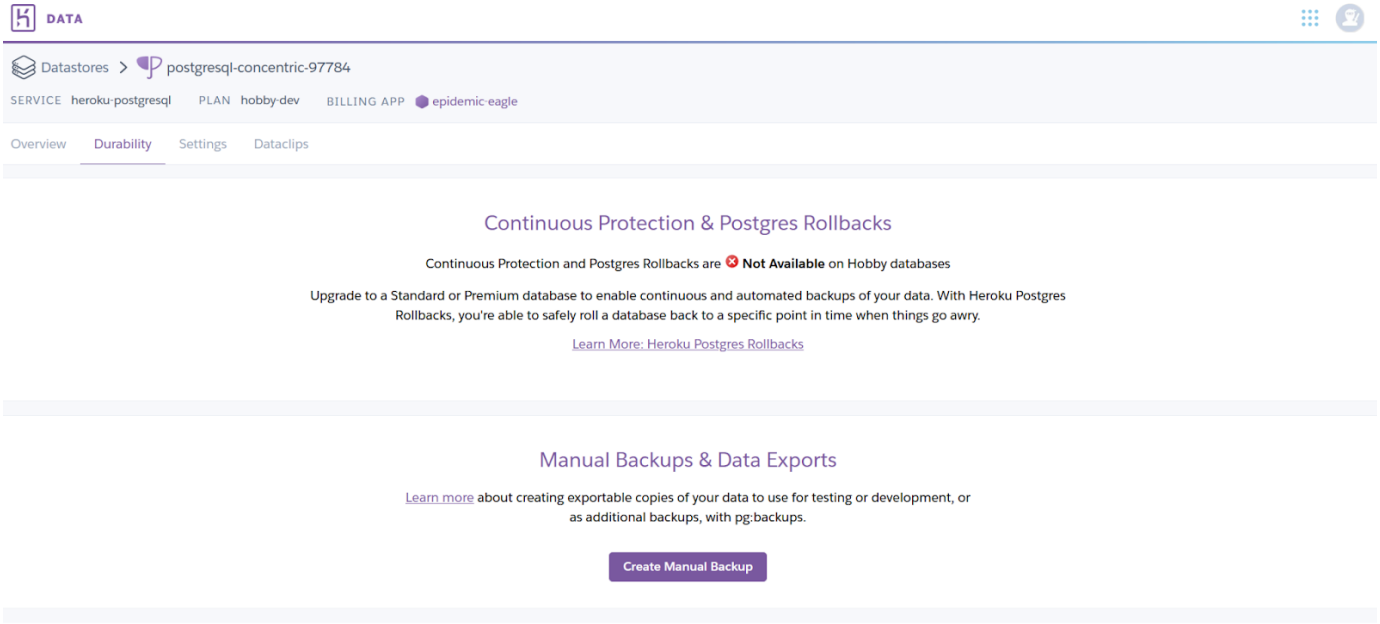
Challenges addressed and shortcomings

Although our choices for our software architecture have been justified against other examples, there are also some overlying issues that need to be addressed about the tech-stack as a whole. Currently we have no solution to fix the below problems.

Heroku and Heroku PostgreSQL coupling

Our API is heavily reliant on services given by Heroku, as our web hosting and database system.

In the case that Heroku has maintenance or goes offline, our API will not be viewable to the public. Additionally all data stored inside Heroku PostgreSQL will not be accessible.

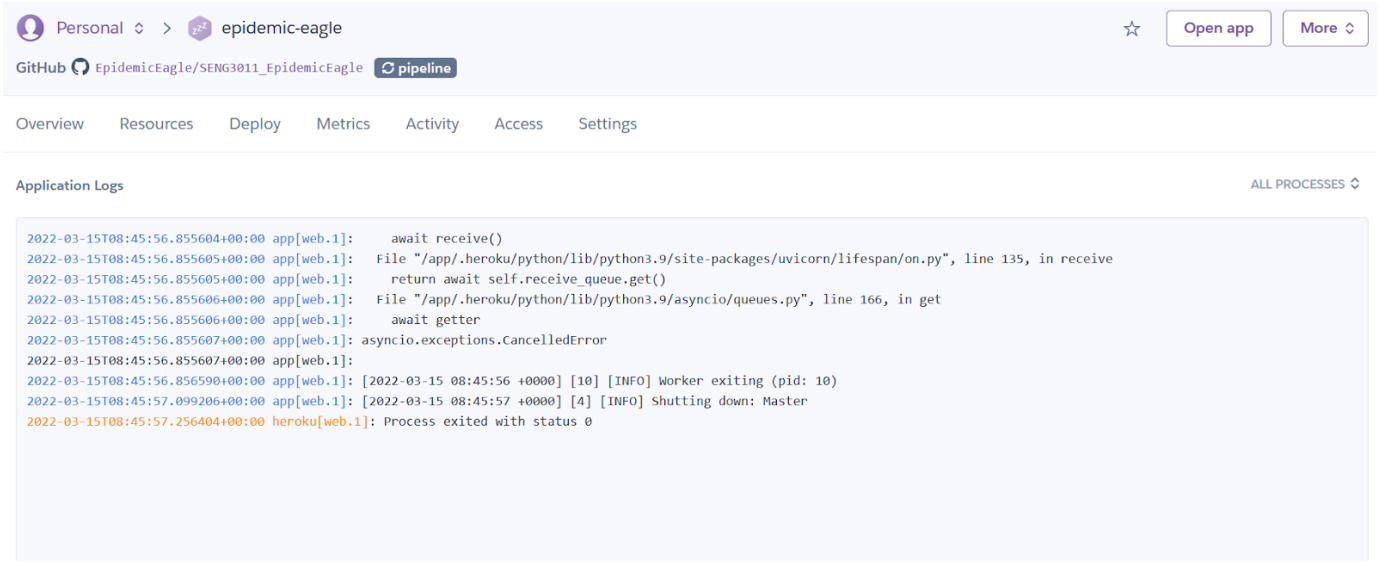


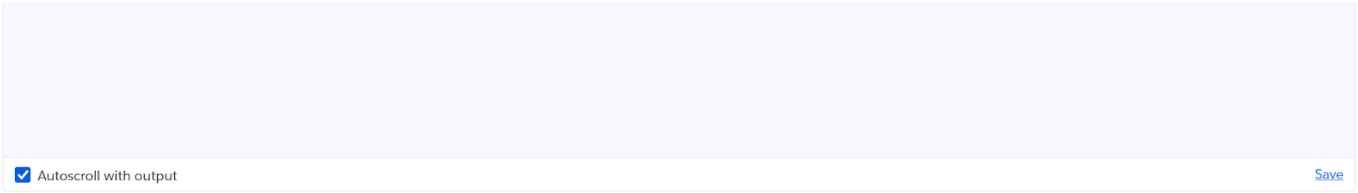
Heroku Timeouts

One of the issues with Herokus free tier for web hosting is timeouts logging during inactivity.

When our API is inactive, a fragment of error code will be logged in our log file, using up space.

When there are a lot of these error logs, then logs of previous calls to our API will be deleted and not be able to be viewed again. Additionally when our API startups from inactivity, the response time is extremely long, having a negative effect on user experience.





Web scraper with Dynamic Websites

Attempting to scrape ProMed revealed problems with current and future websites, we found that the website was based on JavaScript and it dynamically loads its data. This means that we cannot use a simple function to use a URL and parse HTML classes to extract data for our API. One solution would be to send a post response with a body and parse data from that URL.