Final Report

Use Cases and Requirements

GraphQL API

Today, disease outbreak information is widely published online through mailing lists and news websites. However, the lack of APIs backed by structured datasets is a hindrance to the dissemination of this information. For example, the “Outbreak News Today” website provides detailed reports of new disease outbreaks, but only has a simple search function.

By scraping articles from “Outbreak News Today” and applying natural language processing to find diseases, syndromes, outbreak locations and outbreak dates, we can create a structured dataset exposed through a public API. This allows other applications and end users to quickly and easily search for disease outbreaks based on these parameters.

By allowing other websites to integrate this API, we can increase awareness about disease outbreaks and make it useful to end users in a variety of applications. For example, a holiday booking page could show countries that have had a recent disease outbreak. .

The “Outbreak News Today” website had articles under several categories including “US News”, “Asia” and “Headlines”. Scraping individual region-specific categories would have resulted in poor geographical coverage or duplication and a large number of irrelevant articles. “Headlines” on the other hand appeared to contain mostly outbreak reports though with some missing.

User Stories

- As a user, I want to filter articles based on country, location, publication date, disease and syndrome, so I can find the most relevant information on disease outbreaks.

Parameters

Parameter	**Format**
country	String
location	String
start_date	yyyy-MM-ddTHH:mm:ss
end_date	yyyy-MM-ddTHH:mm:ss
disease	String
syndrome	String

Request

```
query MyQuery {
  listArticles(filter: {headline: {contains: "<disease>"},
           date_of_publication: {between: ["<start_date>", "<end_date>"]}}) {
    items {
      url
      date_of_publication
      headline
      main_text
      Reports {
        items {
          Diseases(filter: {name: {contains: "<disease>"}}) {
            items {
              name
            }
          }
          Syndromes(filter: {name: {contains: "<syndrome>"}}) {
            items {
              name
            }
          }
          Locations(filter: {country: {contains: "<country>"}, location: {contains: "<location>"}}) {
            items {
              country
              location
            }
          }
        }
      }
    }
  }
}
```

200 Response

```
{
  "data": {
    "listArticles": {
      "items": [
        {
          "url": <url>,
          "date_of_publication": “<date_of_publication>”,
          "headline": “<headline>”,
          "main_text": “<main_text>”,
          "Reports": {
            "items": [
              {
                "Diseases": {
                  "items": [
                    {
                      "name": “<name>”
                    }
                  ]
                },
                "Syndromes": {
                  "items": [
                    {
                      "name": “<name>”
                    }
                  ]
                },
                "Locations": {
                  "items": [
                    {
                      "country": “<country>”,
                      "location": “<location>”
                    }
                  ]
                }
              }
            ]
          }
        }
      ]
    }
  }
}
```

400 Response

Bad Request due to malformed input

```
{
  "errors": [
    {
      "message": “<message>”,
      "errorType": “<errorType>”
    }
  ]
}
```

Chatbot

Use Cases

Use case: get started

When the user clicks on the get started button, they should receive a personalized greeting message and two buttons to choose whether they want to learn about outbreaks or diseases.

Use case: search for outbreaks

When the user clicks on the button outbreak, they should receive a messaging asking him for the disease name. After the user enters the disease name, all the reports about outbreaks of this disease in the past 6 months will be returned.

Use case: symptoms and emotional comforting

After the user received the reports, they will also be asked whether he has related symptoms. Once the user answered whether or not they have any of those symptoms, they will receive emotional support according to their answer to the previous question.

Use case: search for diseases

After clicking on the disease button after getting started, the user will be asked to enter a location of the disease. After the user enters the location, all the reports of disease outbreak in the given location will be returned.

Other ideas that we had according to the feedback given by our mentor Yi includes:

- Aggregating reports from multiple news providers on a certain outbreak and generate timeline of events and their updates and might consider visualizing the data collected in the aggregate report
- Medical product recommendation according to analysis of symptoms provided
- Integrating this idea and travel apps
- Outbreak spread prediction based on flight patterns visualized on map based on the disease reports
- Social media style outbreak update based on data from reliable sources

But we were unable to finish them due to time constraints.

Persona

- Alice
    - Drinks herbal tea
    - Working - busy person
    - Gets information from news apps, social media
    - During commute time, lunch break, after work
    - Help make decisions about purchases - panic buying
    - Planning trips
    - Investments in stocks

    - Vegan - cares about the environment
    - Lives alone
    - Calls herself “entrepreneur” on Instagram
    - Posts on Instagram a lot
    - In a bunch of group chats with friends and family
    - Quite gullible - believes a lot of stuff on Instagram
    - 9-5 office job as admin support worker

Epics

As a user, I want to get information from a reliable source, so that I can share it with my friends and family

- Displaying the source of the information
- Content and interaction matches Instagram, group chats, Trump campaign stuff
- Content highly accessible on phone

As a user, I want to know whether travelling to a place is risky, so that I can avoid them

User stories

- As a user, I want to know what outbreaks are occurring in my area or my holiday destination, so that I can rearrange my travel plans
    - Location Search
    - “Are there any current disease outbreaks near London?”
    - “What outbreaks have there been in London over the past week?”
- As a user, I want to find out more about all the outbreaks between a specific period of time by giving the start date and the end date, so that I can make appropriate preventative measures
    - Disease Search
    - “Could you tell me more about the COVID-19 outbreaks between 1/1/21 to 5/1/21?”
- As a user, I want to share articles on disease outbreaks on social media, so I can inform my friends and family about them
    - Could you share that article on my Twitter?”

System Design and Implementation

GraphQL API

Description, Justification, Challenges and Shortcomings

GraphQL

At the suggestion of our mentor, we create a GraphQL API rather than a REST API. Instead of using traditional POST, GET, PUT and DELETE endpoints to create, read, update and delete, users instead send queries, mutations and subscriptions in a JSON format to a single POST endpoint to do the same.

As GraphQL is more expressive than REST, clients can perform the same tasks using fewer calls to the server. However, as the specification was written for a REST implementation, we needed to make several compromises.

GraphQL does not have a direct equivalent to Swagger for documentation and interactive testing. Most GraphQL packages are bundled with an interactive GraphQL client such as GraphiQL. However, these clients are typically not enabled in production and document the API in a terse manner. It was thus necessary for us to direct users to use an interactive GraphQL client-hosted elsewhere for interactive testing and to provide more readable examples in our Readme.md.

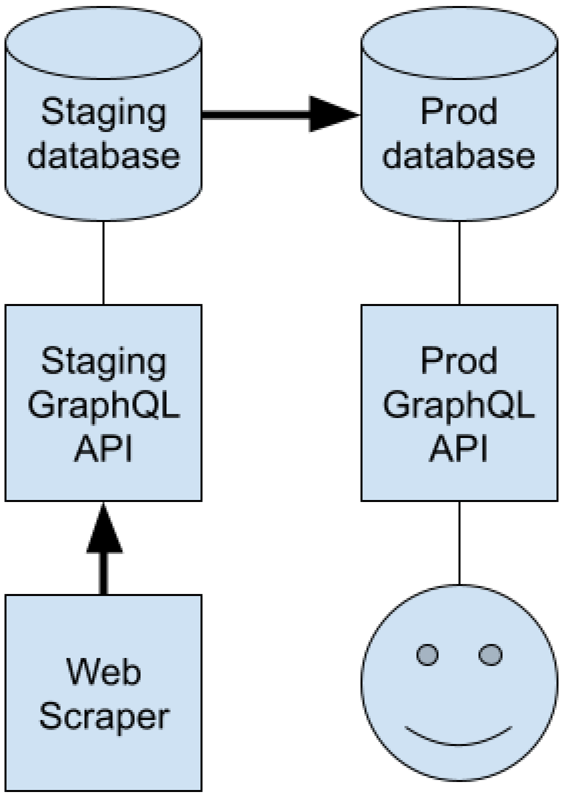
AWS Amplify

Also at the suggestion of our mentor, we used AWS AppSync to deploy the GraphQL API. More specifically, we used AWS Amplify to create a complete backend including a database on AWS DynamoDB, authentication and the GraphQL API using AWS AppSync.

By using AWS Amplify, we were able to interactively create the entire backend. Using the Amplify Admin UI, we were able to specify the database schema using a GUI which automatically creates a corresponding GraphQL API with functions for querying, mutating and subscribing to changes in the database. We are also able to create, read, update and delete rows in the database using the GUI, simplifying management.

Furthermore, using the AWS Amplify CLI, we are able to test a mock of the entire backend on our local machines with a mock SQLite database and a locally compiled and hosted GraphQL API. A GraphiQL client is also made available which allows us to interactively create GraphQL queries.

Using the AWS AppSync console, we were also able to easily change the API key and expiry for the GraphQL API. We were also able to easily enable logging of API calls to AWS CloudWatch, where logs can be easily filtered and statistics displayed graphically. This made fulfilling the logging requirement of the specification simple.



System Diagram

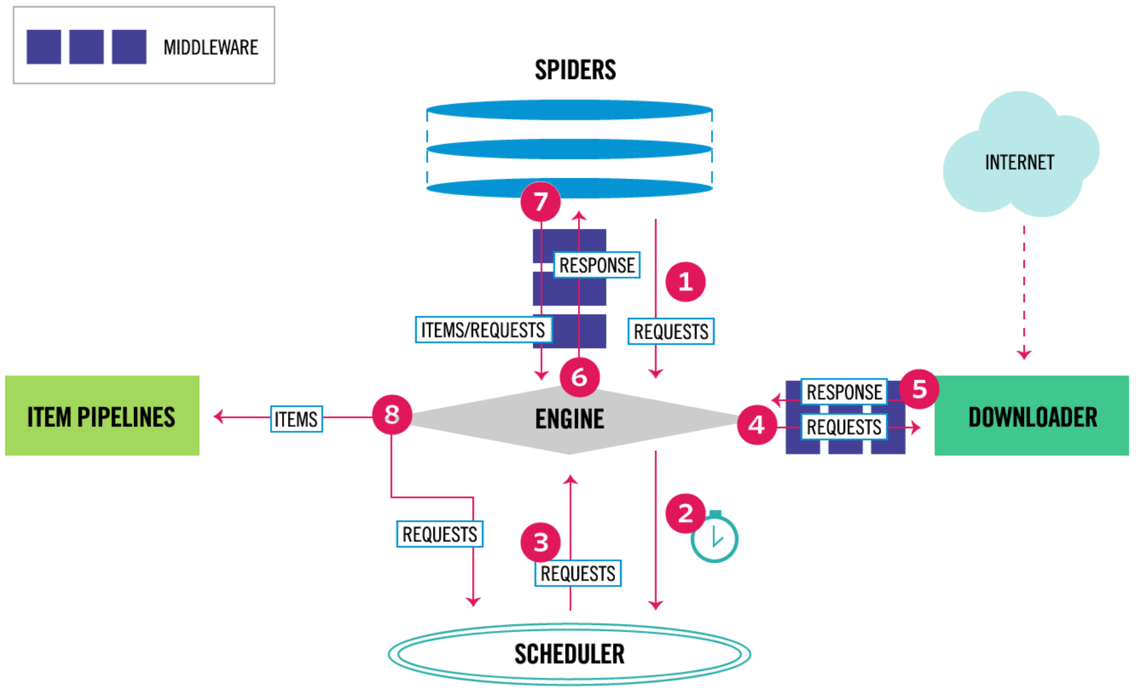
However, AWS Amplify and AWS AppSync are highly opinionated APIs and not without bugs. Firstly, it is not possible to create a GraphQL API without any form of authentication. This means even users of the production API will need to use an API key to access it. Furthermore, limiting users of an API key to only be able to query, but not mutate or delete entries through the Amplify Admin UI or AWS Amplify CLI does not work. Using the other authentication options, Amazon Cognito or AWS IAM, would have required a complex authentication flow only realistic for users if called from a frontend using the corresponding packages for web or mobile clients. Since we expect users to call the backend using simple requests such as through Postman rather than using a frontend, these were not options.

Instead, we resorted to creating a production backend that has the mutation and subscription functions deleted through the AWS AppSync console. This means data from the web scraper must first be uploaded to the staging deployment using GraphQL mutations, then copied using a NodeJS package such as “copy-dynamodb-table” to the production deployment. This is acceptable as data from the web scraper will only be uploaded once. However, were the web scraper to be run automatically, this is clearly undesirable.

Finding a solution for the deficiencies in AWS Amplify likely took more time than writing our own GraphQL server using a framework such as Apollo which also allows enabling an interactive GraphiQL client in production. AWS Amplify and AWS AppSync are thus clearly targeted at web app developers rather than API developers and should not be used for the latter.

Web Scraper

Scrapy



Only one member in our group has previous web crawling experience. Scrapy is an efficient and easy-to-use tool, so we used it as a way to extract data from websites. We can easily extract data using the Scrapy selector by processing the HTML tags. For instance, we can get the headline of each article using the <title> tag. However, even though Scrapy is an easy-to-use tool for beginners, we still encountered a challenge when trying to clean the text of the main content. The articles on the “Outbreak News Today” website have different structures. Some articles have the main content under the <p> tag, while others use different tags. Some articles have dot points in the main content and other irrelevant texts also use dot points or the <p> tag. This took us a lot of effort to solve. Since the date of publication of an article is usually close to the event date, some articles may not contain the date when the event happened, in these cases, the event dates are set to the date of publication.

SpaCy

For the data in the reports list, the diseases, syndromes, and locations are extracted and classified from the headlines and the main texts. Originally, we used SpaCy as an NLP tool to classify them. However, diseases and syndromes are both labelled as "DISEASES" in the SpaCy corpus, which makes the classification process more difficult. For the syndromes part, we still used a combination of SpaCy and some hard coding for the part-of-speech and the lemmatization tag part to obtain the desired result.
For some unknown reasons, MyMerpy did not work on our local machines, but it worked on the CSE Vlab. Memory use increased when we ran the program to process texts using SpaCy, and therefore the program was killed because of the limited memory usage in the CSE environment. We also considered using the AWS Comprehend to classify the

syndromes, but the database is incomplete. For example, COVID-19 was not in the database package, and we could not add it manually either. So we went back to using SpaCy and found that we only need to collect the garbage after each text processing, so the memory usage issue was fixed.

MyMerpy

Merpy is a python package that wraps around MER - the Named-Entity Recognition tool developed and maintained by lasigeBioTM. Given any lexicon and any input text, Merpy is able to return the list of terms recognised in the text, including their exact location (annotations) in the original input. We have forked the Merpy repository and renamed it as MyMerpy to extend the functionalities and make it more suitable for this project.

The medical corpus available for SpaCy is not complete or up-to-date, this has caused an issue with recognizing newly appeared diseases and differentiating disease names from syndromes. MyMerpy takes the advantage of the Human Disease Ontology, which is a comprehensive hierarchical controlled vocabulary for human disease representation and tweaks it to output a few compact lexicon files. With MyMerpy, we are also able to add diseases manually into the lexicons as it gives additional scalability for modification.

Deep down inside MER, it was developed and tested using the GNU stream editors including awk (gawk), sed and grep. There has been an issue with BSD version command line tools on macOS such that the program is not assured to work on macOS without installing GNU utilities.

Another potential difficulty of using MyMerpy is that disease names mentioned in the articles can be abbreviated, such that they are different from the medical terms in our lexicons. There has been a solution provided by lasigeBioTM that returns all possible terms that are similar to the recognized ones. We will look into this solution in the next deliverable as it’s not a vital problem for the current deliverable.

Geonamescache

For the location part, SpaCy identifies both countries, cities and states as “GPE” and has no distinction between them, therefore we had to resort to other libraries to further categorise places names in GPE into country and location. We also considered AWS Comprehend, but it also does not differentiate between country and other types of locations. Therefore, we initially used the Wikipedia API, the definition of the obtained location name on Wikipedia will be checked, if the keyword “country” exists it will be classified as a country and location otherwise. But this approach did not achieve desirable accuracy either, for example, Norway could not be identified but “or way” was identified instead. We also tried using Nominatim API, but as there is a limit to the request number we can make, we did not use it in the end. In the end, we used geonamescache to extract the country and the city as it can extract the country and the city separately. Every city is given a related country code, which can be used to link the country to the city.
However, the database of geonamescache is also incomplete. For instance, Västra Götaland is a county in Sweden, but it is not inside the database. Then, we decided to use another Python library called geograpy, since it does contain the counties in their database. However, geograpy can only output countries, cities and regions. Regions contain all the

data of the country, the city, the county and other stuff (such as diseases). It doesn’t have a method to get the county, and some disease names are being classified as regions, e.g. “Yersinia”. Hence, we continued using geonamescache, even though the database is incomplete, but it works well for now and this package also contains the geonames_id which will help us to do the advanced version of the locations part. However, we will add this in the next deliverable.

API Testing

Using AWS Amplify, we were able to test the GraphQL API using a mock on localhost. This was backed by a SQLite local database and exposed a GraphiQL client for interactive testing.

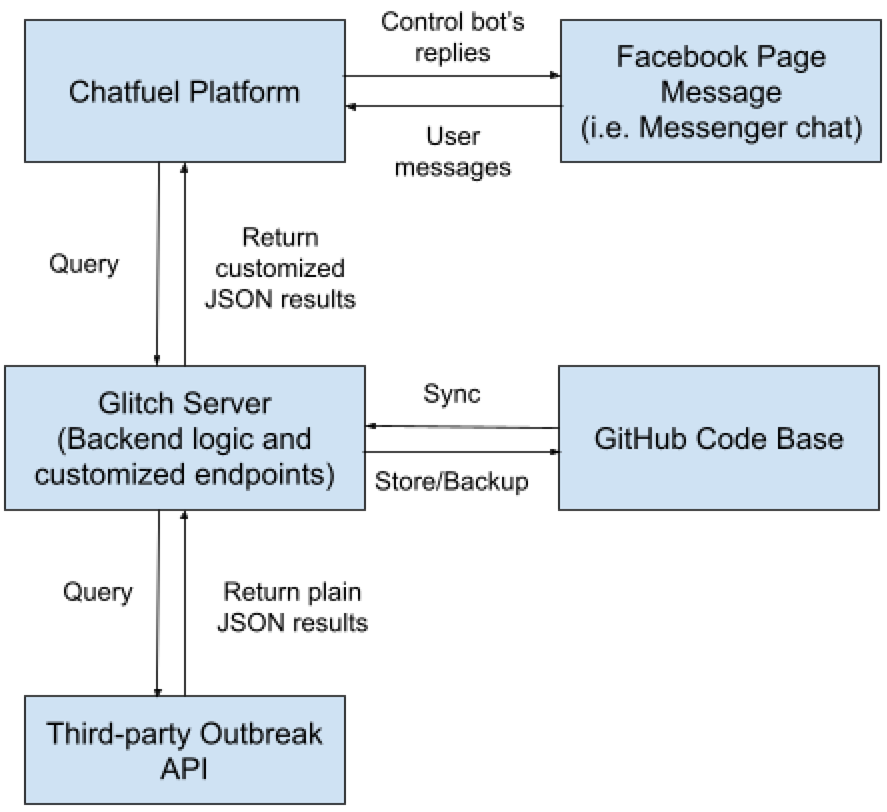
We first created dummy data which we inserted into the local database using GraphQL mutations. We then used Postman to write unit tests and saved them to a collection. Using Postman instead of SuperTest allowed us to test the endpoint as an end user instead of in a testing environment whilst still writing tests in JavaScript.

We tested filtering on all the fields in the API specification and tested performance as well as functionality. The collection runner allowed us to export the results into a JSON file and uploaded these as part of our testing documentation. We also ran our unit tests on the staging API.

We also documented how end users could test the production GraphQL API using Altair in the Readme.md. Since GraphQL endpoints expose usage documentation by default, users can simply input the URL and API Key for the API into Altair and interactively construct queries in a guided manner.

Chatbot

Description, Justification

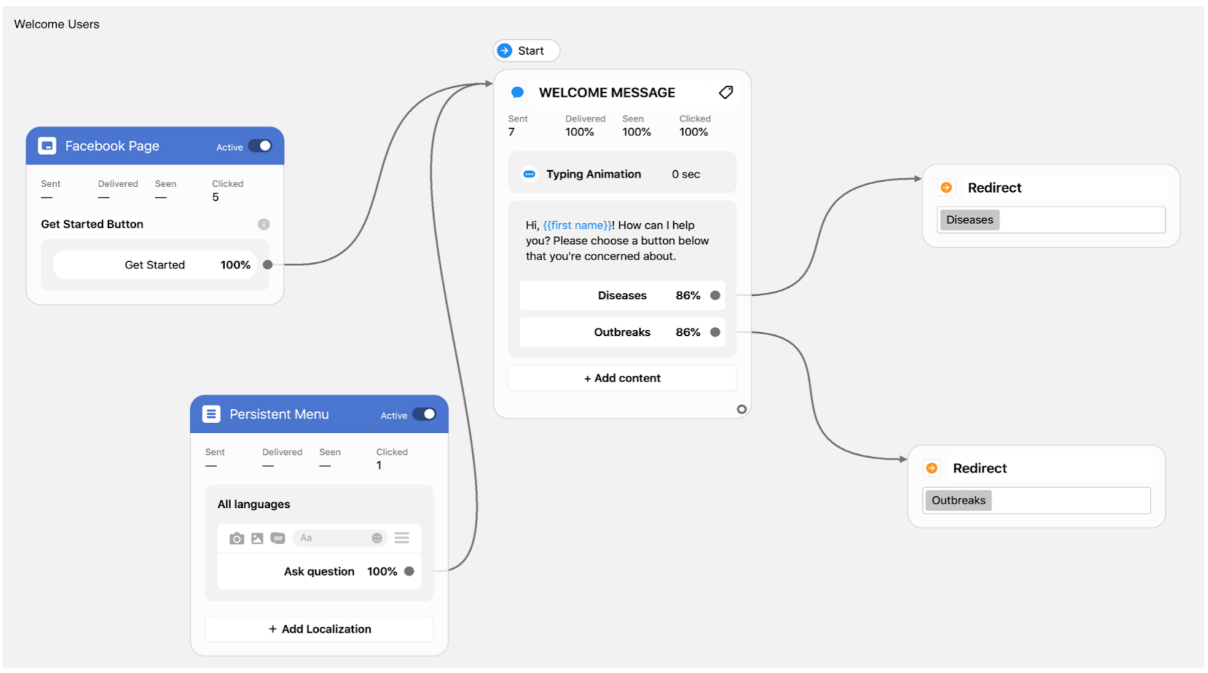

Chatfuel Platform
Control bot's replies
Facebook Page Message (i.e. Messenger chat)
User messages
Query
Return customized JSON results
Glitch Server (Backend logic and customized endpoints)
Sync
Store/Backup
GitHub Code Base
Query
Return plain JSON results
Third-party Outbreak API

System Diagram of Chatbot

Having adopted AWS Amplify service for the last 3 deliverables, we always want to find a platform that is simple and economic to create our chatbot. Since we switched to the idea of Chatbot when there was only one week left for this deliverable, we came up with the above architecture to minimize the workload of configuration.

Chatfuel

We decided to use Chatfuel as the top layer of our chatbot. It can be seen as a Transmit-Receive system with extra functionalities such as extracting a certain type of keywords from the text. It enabled visualization of our message flows and replaced the part of code that we have to implement for flow control.
A big advantage of Chatfuel is that it manages the conversation state(flow) and completely isolated our backend server from state control, such that the backend server is free of huge conditional blocks. Since Chatfuel has built-in NLP tools, we didn't have to invoke local NLP tools in our backend application, which should help keep response times low.
The flow of our Chatbot and the real Chatbot conversation are as follows.


Welcome Users

Start

WELCOME MESSAGE

Sent	Delivered	Seen	Clicked
7	100%	100%	100%

Typing Animation 0 sec

Hi, {{first name}}! How can I help you? Please choose a button below that you're concerned about.

Diseases 86%

Outbreaks 86%

+ Add content

Facebook Page Active

Sent	Delivered	Seen	Clicked
—	—	—	5

Get Started Button

Get Started 100%

Persistent Menu Active

Sent	Delivered	Seen	Clicked
—	—	—	1

All languages

Ask question 100%

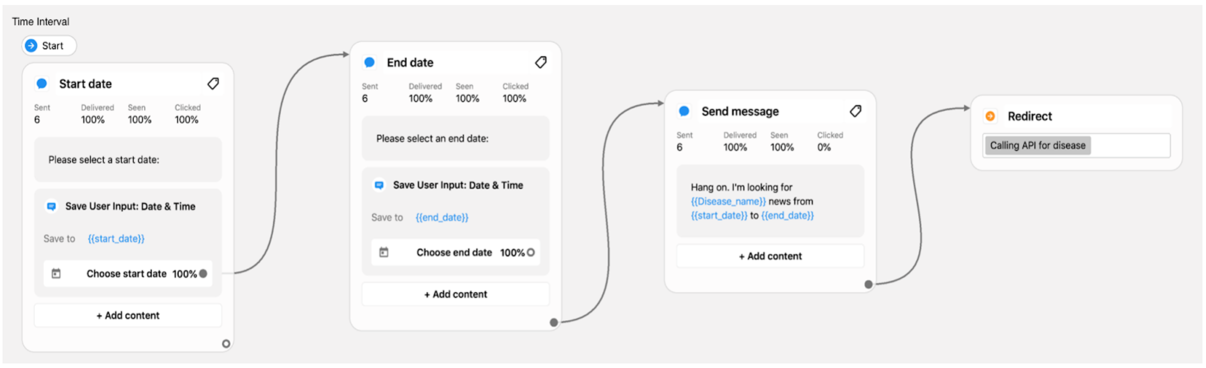
+ Add Localization

Redirect

Diseases

Redirect

Outbreaks


Time Interval

Start

Start date

Sent	Delivered	Seen	Clicked
6	100%	100%	100%

Please select a start date:

Save User Input: Date & Time

Save to {{start_date}}

Choose start date 100%

+ Add content

End date

Sent	Delivered	Seen	Clicked
6	100%	100%	100%

Please select an end date:

Save User Input: Date & Time

Save to {{end_date}}

Choose end date 100%

+ Add content

Send message

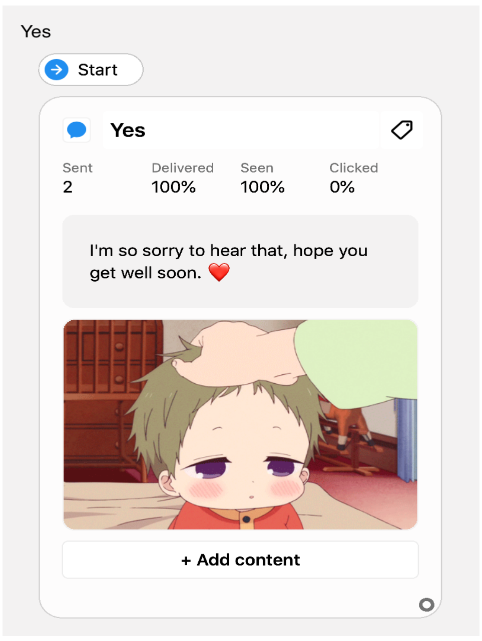
Sent	Delivered	Seen	Clicked
6	100%	100%	0%

Hang on. I'm looking for {{Disease_name}} news from {{start_date}} to {{end_date}}

+ Add content

Redirect

Calling API for disease


Yes

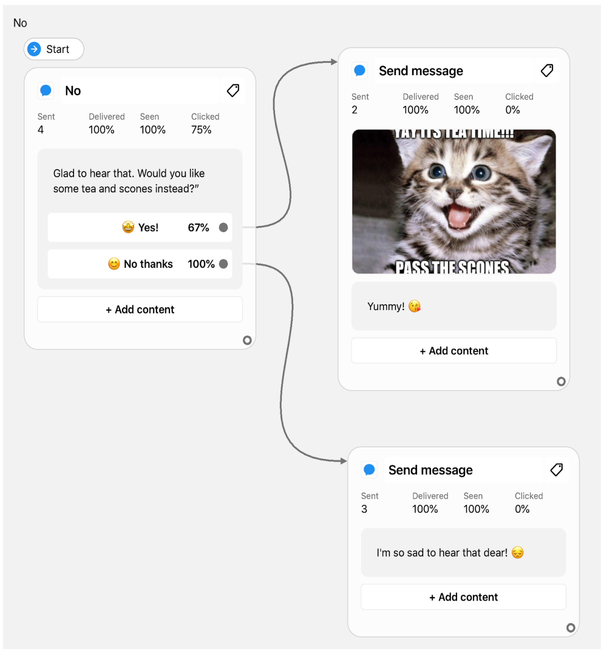
Start

Yes

Sent	Delivered	Seen	Clicked
2	100%	100%	0%

I'm so sorry to hear that, hope you get well soon. ❤️

+ Add content


No

Start

No

Sent	Delivered	Seen	Clicked
4	100%	100%	75%

Glad to hear that. Would you like some tea and scones instead?"

🤩 Yes! 67%

😊 No thanks 100%

+ Add content

Send message

Sent	Delivered	Seen	Clicked
2	100%	100%	0%

Yummy! 😘

+ Add content

Send message

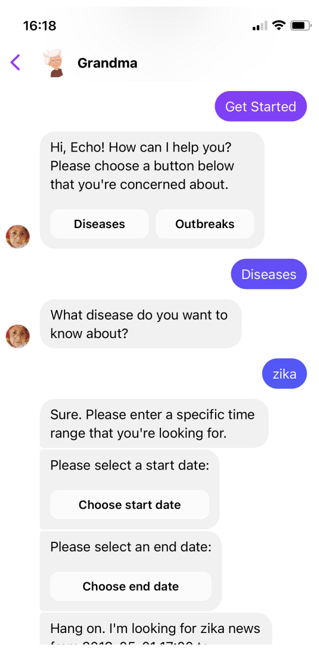
Sent	Delivered	Seen	Clicked
3	100%	100%	0%

I'm so sad to hear that dear! 😔

+ Add content

Flows of Chatbot

Get Started by disease name:



16:18

Grandma

Get Started

Hi, Echo! How can I help you? Please choose a button below that you're concerned about.

Diseases | Outbreaks

Diseases

What disease do you want to know about?

zika

Sure. Please enter a specific time range that you're looking for.

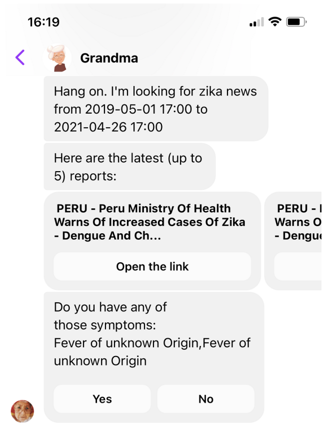
Please select a start date:

Choose start date

Please select an end date:

Choose end date

Hang on. I'm looking for zika news



16:19

Grandma

Hang on. I'm looking for zika news from 2019-05-01 17:00 to 2021-04-26 17:00

Here are the latest (up to 5) reports:

PERU - Peru Ministry Of Health Warns Of Increased Cases Of Zika - Dengue And Ch...

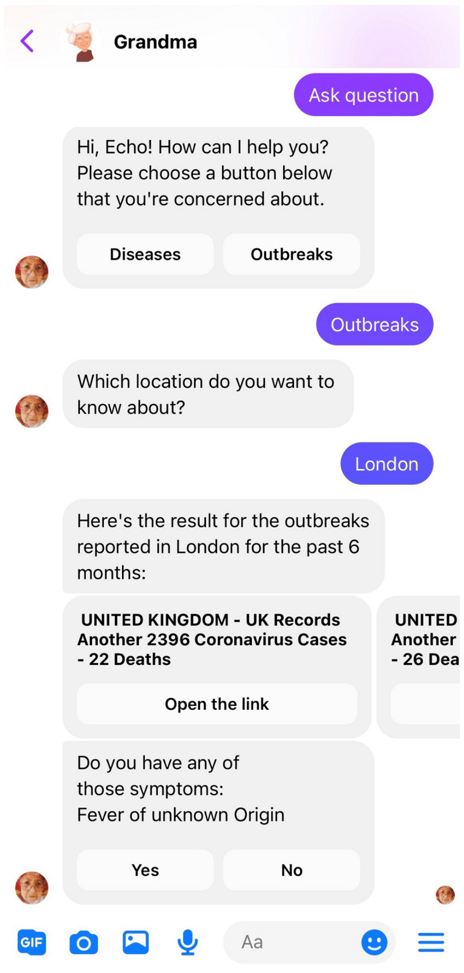
Open the link

PERU - I Warns O - Dengu

Do you have any of those symptoms:
Fever of unknown Origin,Fever of unknown Origin

Yes | No

Get outbreak news by menu entry:



Grandma

Ask question

Hi, Echo! How can I help you? Please choose a button below that you're concerned about.

Diseases | Outbreaks

Outbreaks

Which location do you want to know about?

London

Here's the result for the outbreaks reported in London for the past 6 months:

UNITED KINGDOM - UK Records Another 2396 Coronavirus Cases - 22 Deaths

Open the link

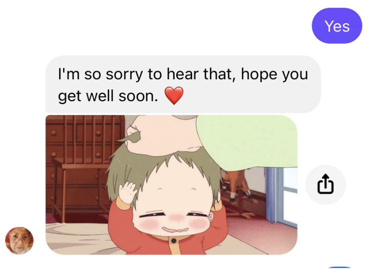
UNITED Another - 26 Dea

Do you have any of those symptoms:
Fever of unknown Origin

Yes | No

Aa

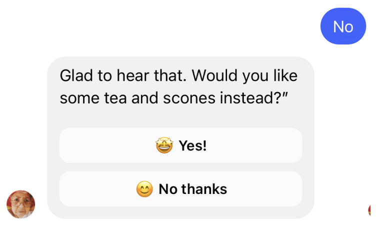
Yes (Have symptoms):



Yes

I'm so sorry to hear that, hope you get well soon. ❤️

No (No symptoms):



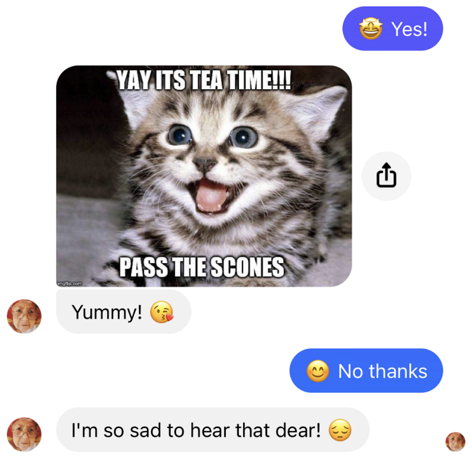
No

Glad to hear that. Would you like some tea and scones instead?"

🤩 Yes!

😊 No thanks

Tea or Scones:



🤩 Yes!

Yummy! 😘

😊 No thanks

I'm so sad to hear that dear! 😔

Glitch

The Glitch server is where the majority of our code was deployed on. It has been connected to our GitHub repo for automatic code update. Glitch is able to automatically deploy in real time onto the cloud servers, so there's no provisioning of servers or management of infrastructure, just the joy of creating.
The code published to Glitch has three main responsibilities, first is to analyse keywords from user messages, second is to query the third-party Outbreak API based on these keywords, and last is to modify the response in order to fit required format by Chatfuel.
Glitch offers up to 5 projects(applications) to premium users without rate limit or server sleep, such that we could eventually set up multiple backend servers(applications) that were linked to and queried by different nodes in the message flow. This great feature is a semi-implementation for parallel processing of requests.
Compared with our previous Amplify approach, Glitch has the ability

- to host the application instantly,
- to implement automated deployment in real time onto the cloud servers,
- to split the strain on a single Amplify server between multiple sub-servers,
- to omit configurations for web tokens(while security issue persists for public projects)

Third-party Outbreak API

We used the API from Christopher Joy's team, it is because their data source is structured and therefore the results are more reliable.

Challenges

As we have integrated Chatfuel to our hosted backend application to call the API, we needed to write code to get the results from the outbreaks API. There were a few problems that we have encountered during this process. One of the problems is that the Glitch server was returning 403 most of the time, this was due to the Glitch server going to sleep after 5 minutes if they are not used if the account is not premium.

Shortcomings

The identity verification cycle for publishing Facebook Page to the public, would take up to 21 days. Due to our short development period, we didn't have enough time to finish the verification so that only our group members have full access to this Chatbot. Although Facebook Developer allowed us to add test users, the addition could only be done manually and there was no way for us to message our Chatbot from all test users at the same time. From that point, potential performance issues persist with our Chatbot.

Management Information

The team was divided into pairs, a group of 2 and a group of 3. They each had a Scrum master and product owner. Collaboration was through Microsoft Teams video calls and Facebook Messenger. We had weekly team meetings alongside the mentoring session which was also the standup where we presented the status of the project and each of the pairs.

Meeting Frequency

- Standup: Wednesday 1:40pm-2pm
- Other meetings as necessary, on average three hours per week

Project Management Tools

Communication:
- Facebook Messenger
- Microsoft Teams

Coordination and Information Gathering:
- Google Docs
- Assignment Specification

Tracking User Stories:
- GitHub Issues
- Agile Methodology

Version Control:
- Repositories under a GitHub organisation

Desired Competencies

Team Member	**Desired Competencies**
William	Unit Testing APIs
Echo	Web Scraping
Meilin	Frontend
Tina	Frontend
Seyed	Backend, Authentication

Responsibilities

Deliverable 2

Component	**Team Members**	**Role**
Web Scraper	Meilin	Developer
	Tina	Product owner
	Echo	Scrum master
AWS Infrastructure	William	Product owner
	Seyed	Scrum master

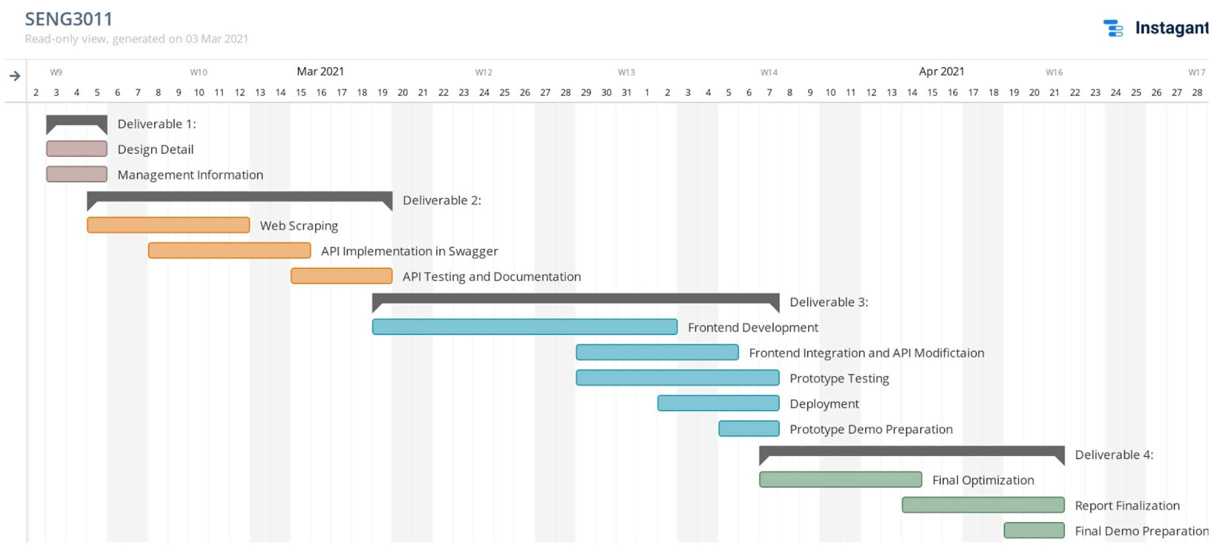
Deliverable 3

Component	**Team Members**	**Role**
Figma Prototype	Meilin	Developer
	Tina	Product owner
	Echo	Scrum master
PowerPoint Presentation	William	Product owner
	Seyed	Scrum master

Deliverable 4

Component	**Team Members**	**Role**
Chatfuel and Glitch servers	Meilin	Developer
	Tina	Product owner
	Echo	Scrum master
Facebook and API Integration	William	Product owner
	Seyed	Scrum master

Project Timeline



Instagantt: https://app.instagantt.com/shared/s/UOAgieR4GTIUehucxiOo/latest

Deliverable 2 Retrospective

What went well?

- Mentor -> No-one else has as complete documentation for their API and is using GraphQL (ambition, flexibility)

What could have been better?

- Communication between parts of the team
- Coding before researching technology (Complex authentication with Amplify backend and somewhat broken permissions)
    - Also Python/SpaCy (Amazon Comprehend medical as an option)
- Tunnel vision (focussed on getting location/country pairs when coordinates were also available as a location option)
- Individual assignment of responsibility for the scraper (accountability only through meetings)
- Waterfall implementation instead of continuous integration and continuous delivery of the scraper (fixed specifications before technology worked out)

What skills do we wish we had beforehand?

- Assigning AWS IAM permissions for access
- Handling web tokens and AWS authentication flows
- Proficient skills in Scrapy and NLP tools
- GraphQL experience - different query style compared with RESTful

What will we do differently next time?

- more catch-ups and retrospectives
- More pair programming (accountability)
- More agile implementation (spikes before coding)
- Be more honest (flagging issues, things that don’t work -> we can change the design)
    - That’s why having a single product owner is a bad idea

Deliverable 4 Retrospective

What went well?

- Produced a product that met the requirements and impressed mentors
- Product performed during the demonstration
- The persona of Grandma was relatively realistic

What could have been better?

- Communication between groups (queries, user flow, user stories changed from those in Google Doc)
- Allocation of credentials for platforms - Not everyone had access to the Glitch servers / Chatfuel
- Time management - lots of development at the last minute
- Discussion about existing skills before choosing a platform
- Stability of the product - Glitch servers needed to be restarted often

What skills do we wish we had beforehand?

- Proficient in Javascript
- API testing
- Familiarity with cloud services
- Ability to debug quickly - test driven development
- Familiarity with webhooks

What will we do differently next time?

- Start earlier rather than leaving everything to the last minute
- Document changes in Google Doc
- Compare frameworks before choosing one
- Stability and performance testing before the demo