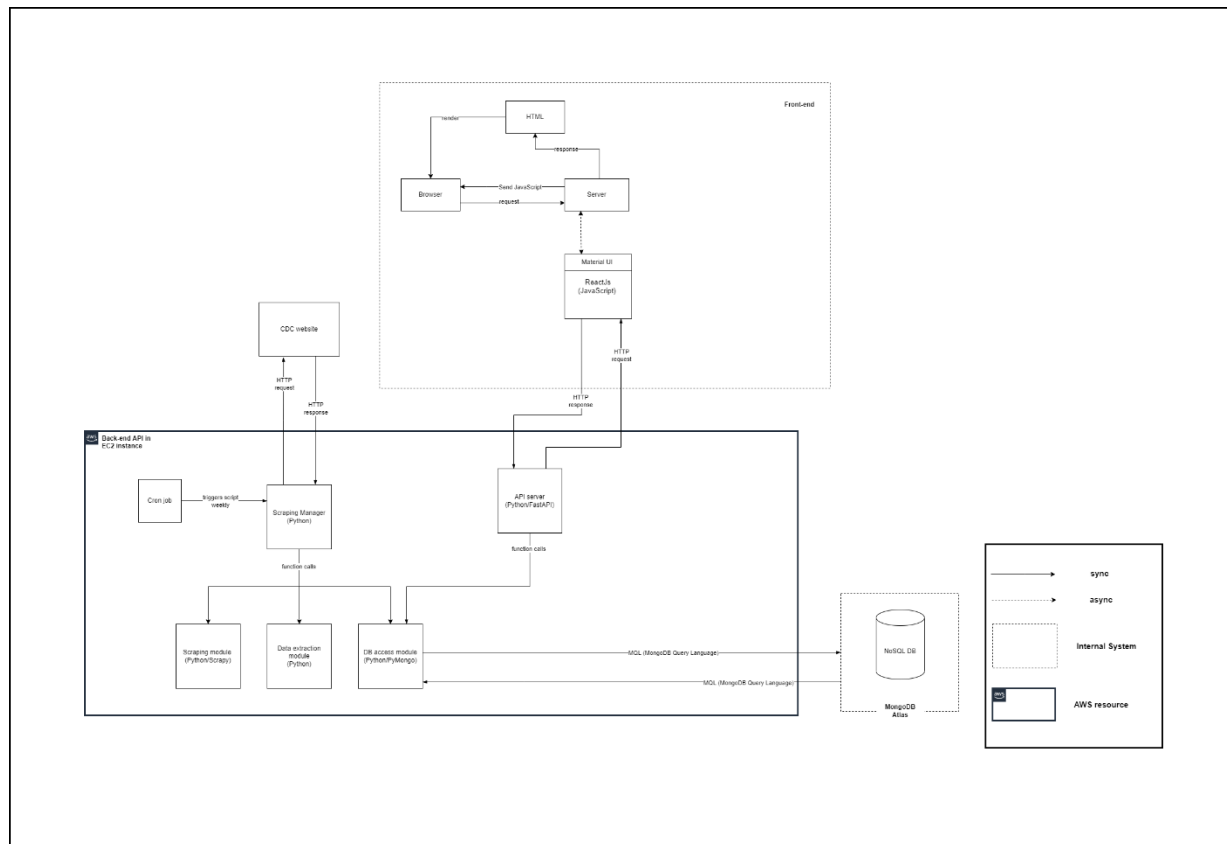


Design Details Report

System & API Design

An API, or Application Programming Interface is a number of collective defined rules that allow several tasks or devices to connect and communicate with each other aiming to accomplish a certain goal. This can be done through following a REST methodology or REST structural design. Applying REST principles through having uniform interface to control the resources, client-server separation design, stateless requests where the system is layered with cacheable responses, we present our scraping REST API solution. It explains our tools and environments selection based on several advantages.



Our system is based around the use of AWS to compartmentalise specific modules. These modules are the scraper and API, which will utilise Scrapy and FastAPI respectively. The scraping module scrapes data from the CDC website and is run on a schedule. This data is stored using MongoDB and their external NoSQL database with MongoDB Atlas. The API

module as stated above is hosted on a separate AWS instance. Calls to the module can be made from a client application through specified API endpoints.

API Design

API Design Principles

The design of our API module for this project should aim to achieve two key principles. These being platform independence and service evolution. Platform independence describes how any client should be able to call the API regardless of how it may have been implemented. Service evolution describes the evolution of the API, which should be able to occur independently from client applications. These principles are especially important for this project considering the chance that other groups who have chosen different implementation for their applications may want to use our API without fear of any updates breaking their application.

API Design Standards

We will use a set of standards to achieve these two principles. RESTful APIs will be used as the architectural approach for our group. REST aids platform independence in how it uses open standards. This avoids coupling between the implementation of the API and the client application. RESTful APIs are also commonly built upon HTTP protocol, meaning that they inherit methods like GET, POST, PUT and DELETE. The use of these methods in our API allows for clear distinctions between API endpoints, making our API easier to use. Furthermore, using these methods allows for the implementation of standardised HTTP status codes, which should again help achieve platform independence.

API Design Practices

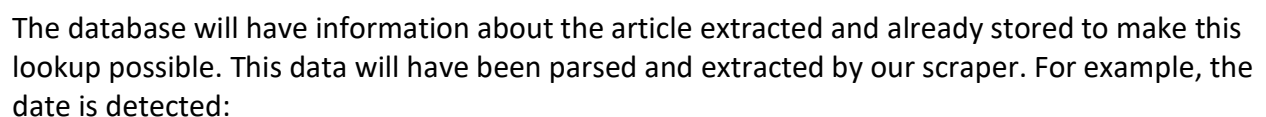
On top of these standards, our group will adopt some further practices to aid the achievement of our two key principles. We will adopt the use of URI versioning to update endpoints in a simple manner. The main disadvantage of URI versioning is that it can become cumbersome if a large number of versions need to be supported. We anticipate that this weakness should not be too big a deal if we plan and implement our endpoints effectively. Versioning will have a large impact on achieving service evolution as updating endpoints will not affect web applications using a previous version.

Integration with the Tech Stack

We have selected parts of our tech stack to help achieve the principles we have listed above. For example, the use of AWS help encapsulate the API module. This means reduced coupling and promotes platform independence. For example, our web application and database

Sample Request

Using our API, we therefore specify the search date we are concerned about, provide our location (“USA”) and specify our relevant key-words (“Listeria, Listeriosis”). This sends the request to our API, which searches through our database to return the response given in the table below. In this case, it will find this article as it is stored in our database:



Recalled Food





Packaged salads produced by Dole

- Solid under multiple brands:
 - Ahold
 - Dole
 - HEB
 - Kroger
 - Lidl
 - Little Salad Bar
 - Marketside
 - Naturally Better
 - Nature's Promise
 - President's Choice
 - Simply Nature
- Products include mixed greens, garden salads, Caesar kits, and many other types of salads in bags or clamshells
- "Best if used by" dates from 11/30/21 through 01/09/22
- Product lot code begins with the letter "B," "N," "W," or "V" in the upper right-hand corner of the package
- See Dole's recall notices [11/22/2021 CS](#) and [01/03/2022 CS](#) for the full list of recalled salads

Investigators found the outbreak strain of *Listeria* in packaged salads produced by Dole.

CDC is also investigating another *Listeria* outbreak linked to packaged salads produced by Fresh Express. See the [Fresh Express recall notice CS](#).

This date is then stored under the 'event_date' category of the reports attached to this article. We parse information from relevant sub-pages too, such as this table describing locations where this outbreak alert is relevant.

| Data Table | |
|--|-----------------------|
| State of Residence | Number of Sick People |
|  Idaho | 1 |
|  Iowa | 2 |
|  Maryland | 1 |
|  Michigan | 1 |

This is all then isolated by our modules and stored in our database, allowing our return to have the specific data the user needs.

As the US resident, we will see therefore that there has been an outbreak in our specified time period and location, and can see the list of locations it has been reported in, as well as other relevant information.

| Request type | URL | Response |
|--------------|---|---|
| GET | https://theimplementers.com/api/v1/articles?start_date=2021-11-29T00:00:00&end_date=2022-01-10T00:00:00&key_terms=Listeria,Listeriosis&location=USA | <p>200 response with JSON</p> <pre>{ "url": "https://www.cdc.gov/listeria/outbreaks/packaged-salad-mix-12-21/index.html", "date_of_publication": "2022-02-01 xx:xx:xx", "headline": "Listeria Outbreak Linked to Packaged Salads Produced by Dole", "main_text": "[PLACEHOLDER - Excluded for report readability]", "reports": [{ "diseases": ["listeriosis"], "syndromes": ["Acute fever and rash"], "event_date": "2021-11-30 xx:xx:xx to 2022-01-09", "locations": [{ "country": "USA", "location": "Idaho" }, { "country": "USA", "location": "Iowa" }, { "country": "USA", "location": "Maryland" }, { "country": "USA", "location": "Michigan" }] }] }</pre> |
| GET | https://theimplementers.com/api/v1/articles?start_date=2021-11-29T00:00:00&end_date=2022-01-10T00:00:00&key_terms=Listeria&location=paddy's pub | <p>400 response with JSON</p> <pre>{"message": "Incorrect syntax for query parameters"}</pre> |
| GET | https://theimplementers.com/api/v1/articles?start_date=2022-01-10T00:00:00&end_date=2021-11-29T00:00:00&key_terms=Listeria,Listeriosis&location=USA | <p>400 response with JSON</p> <pre>{"message": "Provided start date cannot be after end date"}</pre> |

Technology Stack Justifications

API Framework

The implementation of our API module will require the use of a backend API framework. We have decided to program the backend in Python. This is because we all have greater experience with API programming in Python compared to alternatives like JavaScript. Furthermore, Python provides us with a wide choice of backend frameworks that we can choose between for the purposes of this project.

When selecting an API framework, we based our selection around our knowledge of Flask, which is one of the most popular backend frameworks available for Python. We therefore decided to look at potential alternatives that appeared to be similar to Flask. Two of these alternatives are listed below in comparison to Flask.

| Flask | FastAPI | Sanic |
|--|---|--|
| Micro Framework | Micro Framework | Micro Framework |
| Cannot handle asynchronous requests | Can handle asynchronous requests | Can handle asynchronous requests |
| Very large community, extremely well documented | Medium sized community, well documented | Medium sized community, well documented |
| Fast performance, reliably used in backend production environments | Very fast performance | Fast performance |
| No built in documentation | Uses SwaggerUI to generate documentation | Uses Readthedocs to generate documentation |
| Need to use external client | Built in Pydantic support for quick testing | Need to use external client |
| Inbuilt support for database | Inbuilt support for database | No inbuilt support for database |

After weighing the benefits of each backend framework, we decided upon using FastAPI. We made this decision based upon the ease of development that FastAPI could offer to us. For example, it offers built in SwaggerUI integration to help automatically generate documentation, which will be required in later stages of our project. Other ease of use related advantages include built in Pydantic support. Pydantic helps with data validation and parsing among other features, which may be especially useful for this project, where data will need to be collected

and parsed. In addition to these features, FastAPI offers better performance than Flask, helping us speed up our API calls. On top of this, FastAPI is extremely similar to Flask in coding style, both defining API interfaces using decorators. This means that our group's prior knowledge of Flask should carry over relatively well to FastAPI.

Scraper

There are many useful and convenient web scrapers tools for the project , since Python is used for the backend code, we all decided to use Scrapy to extract the data from the source web based on our project specifications. Scrapy is not only a python library but also a framework, the Scrapy framework is the best option to extract data from a particular website for our intended solution. Scrapy is a free and open-source web-crawling Python-written framework that handles HTTP requests in asynchronous, automatic and fast fashion. Furthermore, the crawling speed in Scrapy is automatically adjusted using the AutoThrottle extension. Such extension helps to conserve download delays in concurrent requests dynamically. In addition, Scrapy has built-in support for selecting and extracting data formats using XPath or CSS expressions, unlike other frameworks or tools where scrapable data formats require additional very involved libraries. Scrapy project architecture is built around "spider" bots, which are self-contained crawlers that are given a set of instructions, which we can deploy through AWS (an EC2 instance) and set it up to run periodically. Although Scrapy can be difficult to set up and not beginner-friendly, these shortcomings don't negatively affect our project, as this could be a challenge for us to learn new knowledge. Hence, using the Scrapy framework is suitable for this project.

For further details why we chose Scrapy compare to others web scraper, here is a table we have researched about other scrapers in python:

Web scrapers in Python

| Scraper | Description | Advantage | Disadvantage |
|----------------|---|--|---|
| Scrapy | <ul style="list-style-type: none">• Web crawling framework• Provides spider bots that crawl sites and extract data | <ul style="list-style-type: none">• Asynchronous (multiple HTTP calls can be made asynchronously)• Does not need to be used in conjunction with requests unlike parsers like BeautifulSoup• Can add plugins to increase functionality (like Splash to extract data from dynamic websites)• Great documentation• Low CPU and memory usage | <ul style="list-style-type: none">• Can be difficult to set up• Not beginner-friendly• Is not necessary for simple projects |
| Beautiful Soup | <ul style="list-style-type: none">• Python library for parsing HTML and XML• Should be used with a library like requests since it's a parser | <ul style="list-style-type: none">• Can be combined with other parsers like lxml.• Is beginner friendly with great documentation | <ul style="list-style-type: none">• Is slower than parsers such as lxml• Difficult to scrape data from non-static sites |
| lxml | <ul style="list-style-type: none">• Python library for parsing HTML and XML• Should be used with a library like requests since it's a parser | <ul style="list-style-type: none">• Faster than most parsers• Lightweight• Pythonic API | <ul style="list-style-type: none">• Documentation is not very beginner friendly• Difficult to scrape data from non-static sites |

| | | | |
|----------|---|--|--|
| Selenium | <ul style="list-style-type: none"> • Python library originally for automated testing of web applications | <ul style="list-style-type: none"> • Can scrape dynamically loaded pages • Can mimic user behavior on a page | <ul style="list-style-type: none"> • Very slow as it needs to load and run JS • Difficult to set up • High CPU and memory usage • Not suitable for larger scale projects due to speed and resource usage |
|----------|---|--|--|

There are some alternatives apart from scrapers in python, which allow non-coders to use for scraping, that we consider reasonably not to use for instance Parsehub, Octoparse, etc. Because of many shortcomings of those scrapers, they become improper for our project, like for Parsehub (Parsehub is a web scraping tool where no Python code is written, scraper and scraped data are stored on their cloud systems, they can be accessed via an REST API request. We could either directly use this data ourselves, or we could then store it in our own database), although coding knowledge is not required, we don't know what is under the hood for the scrapers, as we aren't coding it directly. Which is a pain for us when testing and debugging. Besides the benefit that Parsehub deploy the scraper for us and maintain it, there are some drawbacks which strongly affect our choice of scrapers:

- Large limitations in the free version.
- Our system is largely dependent on their system.
- Harder to scale with our own system due to payment limitations. Cf; our own scrapers which are deployed to AWS (an EC2 instance) and scheduled to run periodically. All we pay for is exactly what we need, and we are only paying for the computer, not any under-the-hood maintenance which we wouldn't need.

In conclusion, systems like Parsehub are designed for more corporate or enterprise use with a huge amount of overhead, BeautifulSoup is more suitable for smaller projects that involve scraping only a few web pages, Scrapy is suitable for large projects as it is a scraping framework with sufficient functionality, Selenium is most useful for scraping data from JS heavy sites, which is not applicable to our source web (CDC). Therefore, compared to other scrapers, Scrapy is the most appropriate for our project despite all of its shortcomings.

Database

Why use a database?

When deciding on our tech stack, we considered the possibility of not using a database at all. That is, our scraper would simply run each time an API call was made, and the API would return the results of that particular scrape. However, we decided against this for numerous reasons. Firstly, the **speed** of our API would be substantially slower. With no database, we would have to wait on a scraper to initialize, connect to the CDC website, wait for a response, parse this response, and then return it every time an API call would run. This would get even slower if multiple API requests were being made simultaneously, as multiple instances of the scraper would need to be running (or at least configured to run), wasting important resources.

Secondly, our API would have **too close coupling** to the CDC website if it were to call the scraper directly upon request. We have no control over the CDC website. If it were to go down for maintenance or be particularly slow due to excess load on their servers, our API would fail or slow down as well. Equally, if the website changes structure (e.g., an HTML tag changes), or the important URLs change, then our scraper will no longer work. With no database storing the information, this would cause our API to fail as well.

What type of database?

We now turned to looking at the advantages and disadvantages of relational databases (PostgreSQL, MySQL, etc.) vs. non-relational databases (DynamoDB, MongoDB, etc).

Relational Databases

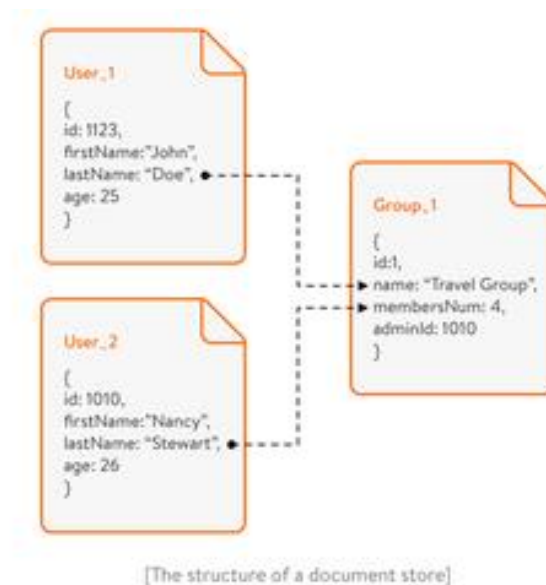
A **relational database** such as PostgreSQL would have the advantage of providing very clearly defined data structures, which promotes a strong degree of data integrity. It also gives good security through well-defined access permissions. Additionally, through compliance with ACID (Atomicity, Consistency, Isolation and Durability) principles, a SQL database will ensure that any transactions made to the database finish completely and correctly before being committed to the database and being accessible by other users. This is a hugely important factor where entries to database tables are constantly being added, changed, and read simultaneously.

Despite SQL's numerous advantages, in our context most are not as important as they otherwise might be. With our CDC database, users will only ever read information from it, never actually inserting new rows, or updating values. Changes to our database will only occur on a scheduled basis whenever our scraper runs, and users do not need a guarantee that the information they are fetching is up to date to the second. While this lack of compliance with ACID principles is problematic in large-scale, transaction heavy databases where users are reading and writing to the database (such as banking applications), here it serves no large benefit. Given our database is only storing articles and their information, clearly defined data

schemas are also not as important as they might be in other contexts, such as a banking database. The benefit of relying on a SQL database is therefore limited.

Non-relational databases

Consequently, it would make the most sense to use a **document-oriented** (non-relational) database. This would give us a much higher degree of flexibility with how we enter data, as it can simply be stored as a file rather than a series of carefully defined schemas. It also improves efficiency overall, as we can store data directly in a JSON format, which is already the format in which it will be delivered to the user and returned by the scraper. Finally, a document-oriented database gives us a horizontally scalable system that can operate across multiple servers, which is good for the future scalability of our product.



Credit: <https://yalantis.com/blog/how-to-choose-a-database/>

Our final choice

In the end we decided on using MongoDB, a widely popular document-oriented database solution. We chose it for the following main reasons:

1. Its industry popularity and huge amount of documentation. This will provide us with a huge amount of support as we develop our system.
2. MongoDB's Python library PyMongo allows us to easily and directly map Mongo data structures to Python data structures, making our API more efficient.
3. MongoDB has a nicely flexible query system. It will allow us to intuitively examine information stored in the JSON and easily find records which are appropriate for what we are looking for.

While we could install Mongo on an AWS instance ourselves, it makes the most sense to use MongoDB Atlas, which will run the database on their cloud and provide us with an interface and API key to communicate with it. This saves us the overhead of setting up the database completely from scratch, while still giving us the complete power over the database system.

Our Python back-end will connect to the Mongo database using the PyMongo module. For the scraper, we will create a MongoDB database user with read/write permissions, so that it can properly update the collections. For the API, we will create a database user with only read permissions, so that it can fetch and return the relevant information when it is called without having the ability to write data to the database. This separation of privilege will help to improve the security of our system. We can then use Atlas' provided connection string in the following format to connect to our database clusters with the relevant user: `client = pymongo.MongoClient(<Atlas connection string>).`

Deployment

Deployment infrastructure importance

Choosing the necessary tools and environments accounts for accomplishing the project deployment requirements; selecting our tools and settings offers familiar, reliable, available, and scalable advantages.

Why AWS?

One of the prominent comprehensive cloud platforms in the computing industry is the well-known Amazon Web Services (AWS). A worldwide cloud infrastructure supports highly-detailed scalable computing resources and other pre-configured solutions. Generally, Amazon solutions are considered a user-friendly option where developers can access a wide range of deployment-related documentation resources to deploy and maintain their projects. This familiarity benefit brings a scalable and reliable system to developers' hands with an available and technically supported infrastructure.

Other cloud service providers present different capabilities, but did not fit our solution scope.

| Other cloud service providers | Advantages | Disadvantages |
|-------------------------------|---|--|
| DigitalOcean | <ul style="list-style-type: none">• More affordable prices compared to AWS. <p>Provides similar alternatives to popular AWS services such as EC2 and S3.</p> <p>Has a developer friendly interface.</p> <p>Provides comprehensive guides.</p> | <ul style="list-style-type: none">• Offers limited free options. <p>Only supports Linux-based servers, which limits choice of OS.</p> <p>Is an IaaS provider, which means all server management should be conducted by developers. AWS offers PaaS services that handle much of the setup and maintenance work for developers.</p> |
| Heroku | <ul style="list-style-type: none">• Offers free tier with some limitations. <p>Similar to Elastic Beanstalk to functionality.</p> <p>Great ease of use.</p> <p>Good scalability.</p> | <ul style="list-style-type: none">• More expensive compared to AWS. <p>Only has regions in the US and Europe regions.</p> <p>Slower deployment for larger applications.</p> <p>Less flexible.</p> |
| Google Cloud | <ul style="list-style-type: none">• One year free trial. <p>More affordable prices compared to AWS.</p> <p>User-friendly platform.</p> <p>Great integration with other Google services.</p> | <ul style="list-style-type: none">• More limited choice of programming languages. <p>Less flexible, with only around 50 services compared to AWS's 200+.</p> |

| | | |
|-----------------|--|--|
| Microsoft Azure | <ul style="list-style-type: none"> One year free trial for popular services. <p>More affordable prices compared to AWS.</p> <p>High availability and scalability.</p> | <ul style="list-style-type: none"> Requires platform expertise. <p>Requires management.</p> |
| PythonAnywhere | <ul style="list-style-type: none"> Free account with limited functionality. <p>Easy to use.</p> | <ul style="list-style-type: none"> Limited to Python. <p>Limited functionality compared to other providers.</p> |

Our deployment specifics

Comparing essential computing services offered through AWS, Elastic Beanstalk (EB), alongside its outweighing advantages, appears to have met the requirements. EB is a deploying and scaling web application easy-to-use runtime environment. It deploys many programming languages, most notably Python, applications and their feasible integrations to connect and take complete control of other components in the infrastructure. Additionally, to optimise the resource usage, management is achieved via Amazon CloudWatch and AWS X-Ray tools offered through the one-year-free tier from AWS. Therefore, EB will allow us to select our respective operating systems utilising several virtual machines such as Amazon Elastic Compute Cloud (EC2) instances and link them to other essential resources. Furthermore, since our preferred Database service is MongoDB Atlas, AWS Quick Start self-managed cluster allows MongoDB to run along with the other resources via command-line.

Mainly, EB is a suitable option in comparison to Lambda and EC2 solely.

| Other AWS computing services | Advantages | Disadvantages |
|------------------------------|--|---|
| Lambda | <ul style="list-style-type: none"> • Serverless. <p>Event-driven trigger execution.</p> <p>Includes self-management to infrastructure by AWS.</p> <p>Pay-as-you-go pricing.</p> <p>Extra functionalities including encryption.</p> | <ul style="list-style-type: none"> • Execution time-out limitation. <p>Complex architecture configuration.</p> <p>RDS database infeasibility.</p> <p>Requires provisioned concurrency.</p> |
| EC2 (Elastic Compute Cloud) | <ul style="list-style-type: none"> • On-demand computing resources. <p>12 months with the AWS Free Tier (750 hours per month)</p> <p>Several pre-configured OS instances (VMs) using Amazon Machine Images (AMIs)</p> <p>Container-based deployment.</p> <p>Scalability: dynamic and predictive.</p> <p>Runs and manages RDS or NoSQL databases (SAP HANA, PostgreSQL, Oracle, Microsoft SQL Server, MySQL, Cassandra, and MongoDB) via Amazon Elastic Block Store (EBS) with storage of 30 GB.</p> | <ul style="list-style-type: none"> • Guest instances need few utility managements. <p>involved database configuration process</p> |

Testing Framework

Overview

| | |
|---|----------------------------|
| Scraping and processing modules (Python - Scrapy) | Pytest |
| API module (Python - FastAPI) | TestClient |
| Frontend (HTML/CSS/JS - React) | Jest |

We will need to test the following aspects of our API service and web application:

- Scraping and language processing modules
 - Data is being correctly scraped and processed. I.e., disease, location, timeframe and case information is accurate.
 - Scraped data is correctly formatted and stored in the database.
- Server module
 - Endpoints behave as expected. I.e., given various types of requests, the response objects returned adhere to our specifications.
- Frontend
 - Information is correctly rendered.
 - UI components behave as expected when interacted with.
 - The website is accessible.

Testing Frameworks

A testing framework is a set of guidelines and best practices for defining test cases. The most common testing frameworks are as follows.

| Framework | Advantages | Disadvantages |
|---|--|---|
| <p><i>Linear scripting framework</i></p> <p>Tests are written as a series of simple sequential steps with hard-coded values.</p> | <ul style="list-style-type: none">• Suitable for beginners in test automation.• Fast to produce test cases.• Test scenario is human readable since the steps are sequential.• Supported in most automation tools. | <ul style="list-style-type: none">• Scripts are not reusable in other tests.• Difficult to maintain and scale as even small updates to the application require changes to test cases.• Data is hard-coded so test cases cannot be used with different datasets. |
| <p><i>Modular testing framework</i></p> <p>Breaks tests down into separate units which are tested in isolation. These units can be combined to build larger test scenarios.</p> | <ul style="list-style-type: none">• Allows for tests to be reused.• As tests are modular, they can easily be modified without affecting other tests.• New functionality can easily be tested with the addition of relevant modular tests and utilising existing tests. | <ul style="list-style-type: none">• Can be more difficult to set up and require greater programming expertise.• Data for each test case is still hard-coded, meaning tests cannot be used with different datasets. |

| | | |
|---|--|---|
| <p><i>Data-driven testing framework</i></p> <p>Inputs and outputs are stored separately from tests in spreadsheets, databases or other repository. These values are then passed to test cases as parameters.</p> | <ul style="list-style-type: none"> • Ability to test with different datasets without modifying tests. • Allows for developing thorough tests more quickly. • Data and test scripts are decoupled, making updates easier and less error-prone. | <ul style="list-style-type: none"> • Requires the most programming expertise and time to set up compared to the previous options. • Can be resource intensive. |
| <p><i>Keyword-driven testing framework</i></p> <p>Like the data-driven testing framework, data and test scripts are decoupled by storing data externally. In addition to the data, keywords for various GUI actions are also stored separately.</p> <p>The test script associated with each action keyword is executed with its corresponding data.</p> | <ul style="list-style-type: none"> • An action keyword can be used to describe multiple test cases, making the code reusable. • Less maintenance required compared to non-decoupled frameworks. | <ul style="list-style-type: none"> • One of the most difficult frameworks to set up. • Requires good test automation expertise. • Keywords can be difficult to maintain and scale. |
| <p><i>Hybrid testing framework</i></p> <p>A combination of the above frameworks.</p> | <ul style="list-style-type: none"> • Suitable for an agile model as it provides a flexible automated testing framework. | <ul style="list-style-type: none"> • It may require greater planning to select the most suitable combination of frameworks. |

Selected framework: hybrid testing framework

We can leverage the strengths of multiple testing frameworks to effectively test the different components in our system using the hybrid framework. This also allows us to gain experience with diverse testing frameworks and assess their suitability for our system.

Backend - data-driven testing framework

To thoroughly test our scraping and language processing modules, we will need to run tests over several articles with varying diseases, location types, time periods and number of reports. Similarly, we will need to use a set of request and expected response values to test our server module. The data-driven testing framework will be used to do this efficiently.

Although we do not have significant experience using this testing framework, we will be able to learn and set up a basic test suite within a reasonable timeframe. The initial overhead of implementing a data-driven testing system is offset by reduced time writing scripts for each test case and increased reliability of our backend service.

Frontend - modular testing framework

We will use the modular testing framework to implement predefined UI test cases. The keyword-driven framework could be used for more rigorous testing, but is not suitable for the scope and timeframe of this project.

Linear scripting may be used for quick testing of all components, but will ideally not be relied on as the only form of testing. As FastAPI can return the OpenAPI schema for our API, we will be able to use Swagger to conduct quick manual tests on our endpoints.

Testing Tools

Backend modules (Python)

| Testing tool | Advantages | Disadvantages |
|--------------|---|--|
| Pytest | <ul style="list-style-type: none">• Has simple syntax.• Functionality can be greatly extended with plugins.• Can run tests in parallel.• Supports fixtures which can be used for data-driven testing.• Great community support. | <ul style="list-style-type: none">• Limited compatibility with other testing frameworks. |
| Unittest | <ul style="list-style-type: none">• Built-in module, so no extra installations are required.• Simple test case execution.• Easy test report generation. | <ul style="list-style-type: none">• Requires large amounts of boilerplate code.• Modelled after JUnit, so used the camelCase naming convention. |

| | | |
|-----------------|---|--|
| Robot Framework | <ul style="list-style-type: none"> • Based on the keyword-driven testing framework, allowing to easy creation of human-readable tests. • Supports all operating systems and application types. • Able to integrate with third party tools. • Great community support. | <ul style="list-style-type: none"> • Does not support parallel testing. • Difficult to get started with. • Keyword-driven methodology may take longer to implement. |
| Nose2 | <ul style="list-style-type: none"> • Extends unittest and is part of the Python standard library, so no additional installations are required. • Has many plugins that improve testing • Allows for tests to be run in parallel. | <ul style="list-style-type: none"> • Documentation is not beginner friendly. • Not actively maintained. |

Selected testing tool: pytest

Pytest supports data-driven testing through the built-in `@pytest.mark.parametrize` marker. The test data is stored as a list of tuples, which the marker maps to parameters which are iterated over in the test function. Test data can also be stored externally and easily placed in a list using a helper function.

Pytest's support of data-driven testing as well as our experience from previous comp courses make it the best option for testing our Python modules.

Our server will also be written in Python using the FastAPI framework. We will use the `TestClient` object from `fastapi.testclient` as this allows us to use pytest to test our API as well. Using the same syntax across our backend testing suite will help us implement testing more efficiently.

Frontend (React)

We considered two of the most widely adopted frameworks for testing React applications.

| Teting tool | Advantages | Disadvantages |
|--------------------|---|---|
| Jest | <ul style="list-style-type: none">• Created and maintained by Facebook.• The most popular testing framework for React.• Provides snapshot testing, which makes UI regression tests simpler.• Minimal setup and configuration needed. | <ul style="list-style-type: none">• Poor documentation.• Slower than some frameworks. |
| Mocha | <ul style="list-style-type: none">• Widely adopted for JavaScript testing.• Good documentation and community support.• Can integrate with many tools. | <ul style="list-style-type: none">• Requires greater configuration.• Not as easy to do snapshot testing. |

We decided to choose Jest as it has been specifically developed to aid in React testing and was briefly introduced as a suitable option in COMP6080.

Frontend

We intend to primarily develop the front-end using the ReactJS JavaScript library. This allows us to create reusable JavaScript components which will be injected into our HTML DOM, reducing repeat code drastically and overall substantially improving the maintainability and extensibility of our website. We will also use Material UI to take advantage of a very wide arrange of already made React components, both cutting down development time while allowing our front-end to maintain a consistent appearance throughout. This will provide our user interface with a sleek, refined and modern look all while keeping our code clean and readable.

Conclusion

In the above report, we have justified our use of specific API design principals as well as the tech stack that will support our API and scraper. We will use AWS cloud services to ensure that our modules are hosted remotely and well encapsulated. On top of this, we have decided that Scrapy, FastAPI and MongoDB will provide us with relevant advantages when it comes to scraping, API frameworks and databases respectively.