香 港 大 學

THE UNIVERSITY OF HONG KONG

COMP4801 Final Year Project
Final Report

Inbox Genius – Your Next Productive Email Client

[Previous Title: Intelligent Automated Mail Filtering Service for Microsoft Outlook Using GPT-3.5]

Brandon Tay Jian Wei (3035767102)

Date of Submission: 26 April 2024

Acknowledgement

We would like to thank our supervisor, Dirk Schnieders, for guiding us throughout this process. Your help is truly appreciated.

Table of Contents

1. Introduction …… 5

2. Backend Methodologies …… 5

   2.1. Developmental Technologies …… 6

   2.2 Infrastructure Technologies ……..... 6

3. Backend Foundations …… 7

   3.1 Authorization & Identity Management ……... 7

   3.2 Database Design & Infrastructure …… 11

4. Development and Integration …..…….…….. 16

   4.1 API Development ……...……..... 16

   4.2 Integration with Frontend & AI Services …….. 18

5. Additional Improvements …… 20

   5.1 SPF & DMARC Checking System ……... 20

6. System Architecture …… 22

7. Conclusion …… 24

8. References …… 25

List of Figures

Figure	Page Number
Figure 1: OAuth 2.0 authorization flow used in our project	9
Figure 2: MSAL token management flow. Application credentials are obtained from EntraID and used to acquire tokens via acquireTokenSilent() or acquireTokenPopup() in the case of token refreshes.	10
Figure 2: Redis cache-aside pattern in action	13
Figure 3: Read replication process. Data is asynchronously replicated from the primary instance to secondary ones, allowing for concurrent read requests from multiple instances.	15
Figure 4: Postman user interface and our testing suite	18
Figure 5: CORS preflight check process flow. An OPTIONS request must be answered before the actual request can be sent.	19
Figure 6: DNS querying in SPF & DMARC checking	21
Figure 7: Overall system architecture of our application	23

List of Tables

Table	Page Number
Table 1: List of API routes implemented	17

List of Abbreviations

Abbreviation	Definition
API	Application Programming Interface
NoSQL	Non-Structured Query Language
OAuth	Open Authorization
MSAL	Microsoft Authorization Library

1. Introduction

This comprehensive report builds upon the progress made in the front-end development and further delves into the advancements achieved in the backend. This project's definition of "backend" refers to a connecting component between the frontend and AI services, thus including tasks like Application Programming Interface (API) development and database integration. This report is split into 5 main parts:

- Backend methodologies: This part introduces the various technologies we used in the backend and provides an accompanying explanation on why we chose them.

- Backend foundations: This part heavily involves laying the groundwork for work to be completed on the backend. Sections in this part include authorization and identity management as well as database design and infrastructure.

- Development and integration: This part delves deeper into our implementation of backend APIs and integration process with both the frontend and AI services.

- Additional Improvements: This part covers all remaining work done that looks to enhance existing processes within the backend.

- System Architecture: This part provides a big-picture overview of the overall system architecture.

2. Backend Methodologies

This section introduces the technologies we used in the backend portion of the project, first touching upon developmental technologies (Section 2.1) and then on infrastructure-related technologies (Section 2.2).

2.1 Developmental Technologies

2.1.1. Development Framework: Flask

Flask was implemented as the back-end development framework of choice thanks to its versatility in development, considering that it can efficiently integrate any third-party Python library [1]. Furthermore, owing to Flask's lightweight nature, it is perhaps the framework with the lowest learning curve [2]. These factors accelerate our development process and significantly decrease the time to market.

2.1.2. API Testing Library: Postman

Postman is a widely used API testing library that provides a user-friendly interface and supports various features for testing and validating APIs such as automatic generation of test cases and documentation. In particular, we chose Postman for our API testing needs thanks to its relative ease of use and human-readable syntax which allow for faster development of API tests. Postman's testing suite is also built on JavaScript, a language we were familiar with, further reducing development times for test cases.

2.2 Infrastructure Technologies

2.2.1. Database: MongoDB

MongoDB is a versatile and scalable Non-Structured Query Language (NoSQL) database that also boasts fast response times for querying operations [3]. Since the nature of our data is generally unstructured, NoSQL databases such as MongoDB become prime choices for database solutions. Moreover, MongoDB's cloud-based deployment solution known as MongoDB Atlas provides us with a quick and easy way to query data from the cloud [4]. Finally, MongoDB was the NoSQL database we were most comfortable with, hence development times could be reduced.

2.2.2. Database Cache: Redis

Due to the heavy read load of our application, we decided that a database caching solution was needed to improve query performance. Redis was chosen as it is a powerful and popular choice to be used as a database cache due to its exceptional performance and versatility. As an in-memory data structure store, Redis excels at storing and retrieving data rapidly, making it an excellent option for caching frequently accessed information [5]. Lastly, Redis also boasts a simple and easy-to-use Python library that can be integrated easily within Flask [6].

3. Backend Foundations

3.1 Authorization and Identity Management

In order to facilitate user login using their Outlook credentials, it is essential to establish a strong authorization and identity management system. This entails integrating with the Outlook authentication infrastructure and ensuring a seamless login experience for users. To accomplish this, we have implemented Microsoft's Open Authorization (OAuth) 2.0 framework, leveraging Azure EntraID and the Microsoft Authorization Library (MSAL) for ReactJS. This effort results in a secure and user-friendly Single Sign-On (SSO) experience directly within the web application.

3.1.1. Overview of OAuth 2.0

OAuth 2.0 is an industry-standard authorization mechanism that “enables a client application to obtain access to a user’s protected resources” such as personal files or email [7]. OAuth 2.0 has grown rapidly over the years and is currently the only authorization flow that supports access to Outlook emails [7].

The OAuth 2.0 flow involves several components that interact with each other to obtain a time-limited access token that grants the client application access to the user's protected resources. Below is a breakdown of each component:

- Resource owner

  The resource owner refers to the user who owns the protected resource, in this case, the Outlook email account. The resource owner is the individual who grants permission to the client application to access their protected resources. In our project, the users of our application are the resource owners [8].

- Client application

  The client application is the application that is trying to access the protected resources on behalf of the resource owner. In the context of our project, our application acts as the client application [8].

- Resource Server

  The resource server is the server that hosts the user's protected resources. In the case of accessing Outlook emails, Outlook's servers act as the resource server. The resource server verifies the validity of access tokens and serves the requested resources if the token is valid [8].

- Authorization Server

  The authorization server is responsible for handling identity verification and token management. In our case, Azure EntraID serves as the authorization server. Before implementing the OAuth 2.0 flow, we had to create client credentials and define scopes. Client credentials are unique identifiers for our application, and scopes define the specific resources our application is authorized to access [8].

Figure 1 shows a diagram illustrating the OAuth 2.0 flow that was adapted for our use case.

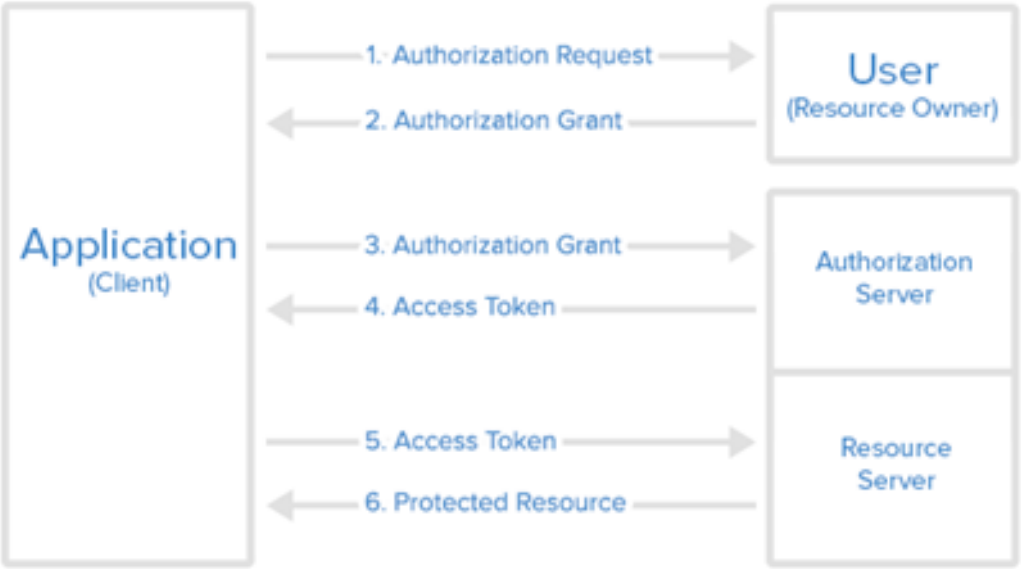

Figure 8: OAuth 2.0 authorization flow used in our project [9]

Below is a step-by-step breakdown of the authorization flow used in our application:

1. Authorization Request: Our application sends an authorization request to the user (resource owner). This takes the form of a sign-on page where the user is requested to permit our application to carry out actions on their Outlook account on their behalf.
2. User Authorization: The user grants authorization to our application via SSO on frontend.
3. Authorization Grant: The user provides our application with an authorization grant.
4. Access Token Request: The application requests an access token from Azure EntraID (acting as the authorization server), presenting its credentials and the authorization grant obtained from the user.
5. Access Token Issuance: Azure EntraID validates the request and issues a time-sensitive access token to the application. This access token must be refreshed once it expires.
6. Protected Resource Access: The application uses the access token to request Outlook emails (the protected resources) from the Outlook server (the resource server) using the Microsoft Graph API. If this access token is valid, the Outlook server returns the requested protected resources to the client application for processing.

3.1.2. Token Management with Microsoft's MSAL Library

Obtaining and refreshing the access token through interacting with the various components in the OAuth 2.0 flow is usually a highly complicated endeavour. Fortunately, Microsoft's MSAL authorization library greatly simplifies the process by providing a streamlined approach to obtaining, managing and refreshing access tokens from Azure EntraID [10]. Vitally, the MSAL library enables fine-grained token management with well-documented function calls in ReactJS's Javascript core, shortening the time we would have taken to implement the authorization flow from weeks to just days.

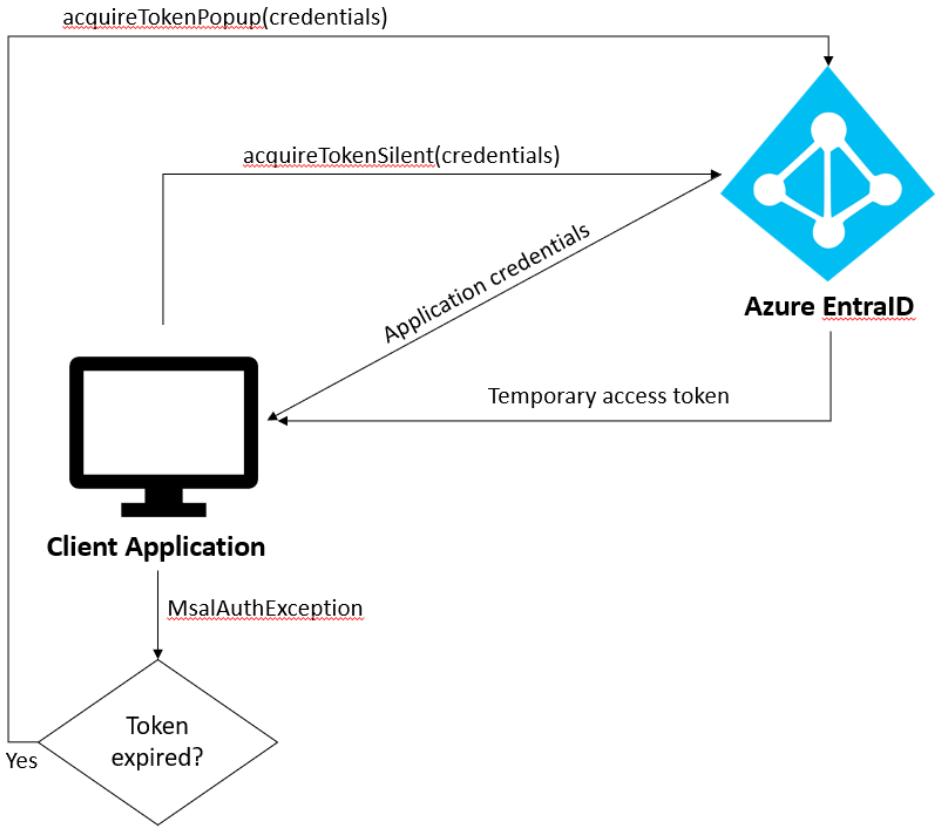

Figure 9: MSAL token management flow. Application credentials are obtained from EntraID and used to acquire tokens via acquireTokenSilent() or acquireTokenPopup() in the case of token refreshes.

Figure 2 illustrates the MSAL token management flow used in the project. Once application credentials are obtained from Azure EntraID, the MSAL library obtains an access token using the acquireTokenSilent() function call. The access token can then be used to obtain email data from Outlook's servers. If an error in obtaining protected resources is encountered, a

MsalAuthException is raised, prompting the MSAL library to check if the current token has expired. In the case of a token expiry, MSAL refreshes the token with the acquireTokenPopup() function call [11].

3.2. Database Design and Infrastructure

A robust design and implementation of a database is key to any high-performant application. This section explores methods we engaged in to engineer and architect our NoSQL database on MongoDB.

3.2.1. Database Engineering

Because MongoDB was used as our primary database, sufficient database engineering was conducted to plan out the necessary collections in anticipation of API development. These collections have evolved throughout the software development lifecycle, but the following 5 schemas represent the finalized schemas for functions related to the backend.

- Users(_id, email, firstName, lastName)
- Emails(_id, outlookId, userId, subject, receivedTime, body, cc, bcc, bodyPreview, category, recipients, sender)
    - userId refers to the _id field of an entry in the Users collection
    - senders, recipients, cc and bcc fields are string arrays
    - receivedDate is an integer representing epoch time
- EmailAiMetrics(_id, emailId, timesClicked, timeSpent, outlookId, category, importanceScore, cSub, cBody)
    - emailId refers to the _id field of an entry in the Emails collection
    - timesClicked, timeSpent and category are integer fields
    - importanceScore is a float value generated by the AI module
- Ics(_id, emailId, icsFilename)
- UserPreferences(_id, userId, prefList, whitelist)
    - prefList and whitelist are string arrays

All _id fields are ObjectId objects that can uniquely identify entries within their collection. All other fields not elaborated on above are string objects.

3.2.2. Database Infrastructure

We opted to use MongoDB Atlas, a cloud-based implementation of MongoDB, as our database hosting solution. This decision offers us numerous advantages, including [12]:

- Scalability
  MongoDB Atlas allows us to easily scale our database as our needs grow. We can seamlessly add more storage, increase throughput, and handle higher levels of traffic without experiencing significant downtime or disruptions.

- High Availability
  MongoDB Atlas ensures high availability of our database by replicating our data across multiple servers and data centres. This replication provides redundancy and fault tolerance, reducing the risk of data loss or service interruptions.

- Automated Backups
  With MongoDB Atlas, we benefit from automated backups of our data. Regular backups are taken and securely stored, allowing us to restore data in case of accidental deletions, hardware failures, or other unforeseen incidents.

- Security
  MongoDB Atlas implements robust security measures to protect our data. It offers features such as encryption at rest, network isolation, and user authentication mechanisms, ensuring that our data remains secure and accessible only to authorized individuals.

- Monitoring and Management
  MongoDB Atlas provides comprehensive monitoring and management tools. We can easily track database performance, analyse query patterns, and identify areas for

optimization. Additionally, the platform offers a user-friendly web interface and APIs for efficient management of our database.

Overall, MongoDB Atlas offers a reliable, scalable, and secure solution for hosting our database in the cloud, empowering us to focus on developing our application without worrying about database infrastructure management.

3.2.3. Database Caching with Redis

To enhance performance in our read-intensive application, we opted for Redis, an in-memory database caching solution. By leveraging Redis's in-memory capabilities, we were able to achieve significant performance enhancements compared to using MongoDB alone. Specifically, we implemented Redis's cache-aside pattern, which is a widely adopted caching strategy designed to optimize read queries. Figure 3 illustrates this process in detail.

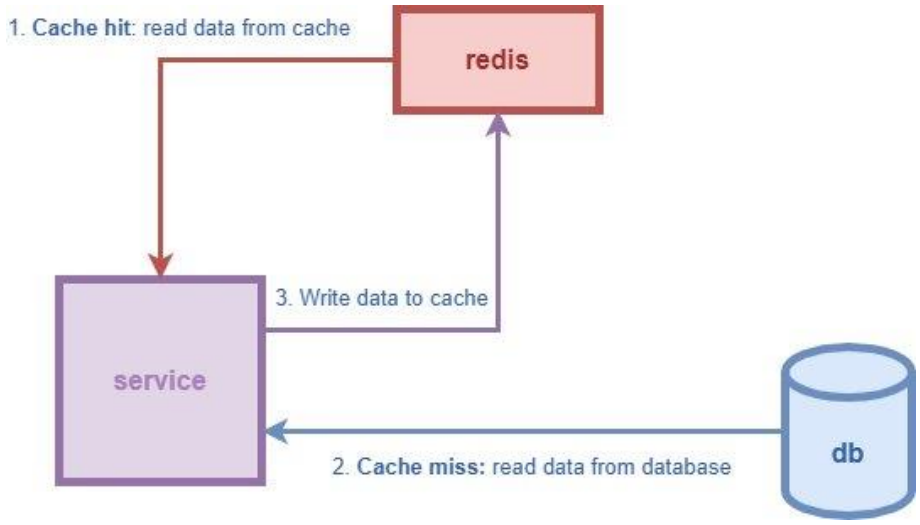

Figure 10: Redis cache-aside pattern in action [13]

In essence, the cache-aside pattern involves 3 key steps [13]:

1. <u>Data Retrieval</u>
   When a read request is received, the application first checks if the requested data is present in the Redis cache. It does this by using a unique key that corresponds to the requested data. If the data is found in the cache (a cache hit), the application retrieves it directly from Redis, bypassing MongoDB Atlas.

2. Cache Miss

   If the requested data is not found in the Redis cache (a cache miss), the application falls back to the primary data source (MongoDB Atlas in our case) to retrieve the data. It fetches the data from the primary source and then stores it in the Redis cache for future use.

3. Cache Update

   After retrieving the data from the primary source, the application updates the Redis cache by storing the data with the corresponding key. This ensures that subsequent requests for the same data can be served directly from the cache, avoiding the need to fetch it from the primary data source.

By following this cache-aside pattern, the application leverages Redis as a caching layer to reduce the load on the primary data source and improve performance. The pattern allows frequently accessed data to be efficiently cached, while less frequently accessed or stale data can still be retrieved from the primary source when needed. It strikes a balance between data freshness and read performance, enhancing overall system scalability and responsiveness.

3.2.4. Database Read Replication

In addition to employing database caching with Redis, we have implemented read replication on MongoDB Atlas to enhance query efficiency. Read replication involves creating multiple copies of the primary database, also known as replicas, and allowing read operations to be distributed among them. When data is written to the primary database instance, this same data is asynchronously replicated to the read replicas in near-real-time, ensuring data consistency across all instances [14]. When a read query is executed, this can be distributed across any of the database instances, resulting in improved response times and increased throughput. Figure 4 illustrates this read replication process.

Read replication was directly configured on the settings of our MongoDB Atlas database instance by enabling key settings. As a result of both Redis caching and read replication, we achieved a roughly 40% reduction in API response times on average when compared to using MongoDB Atlas on its own

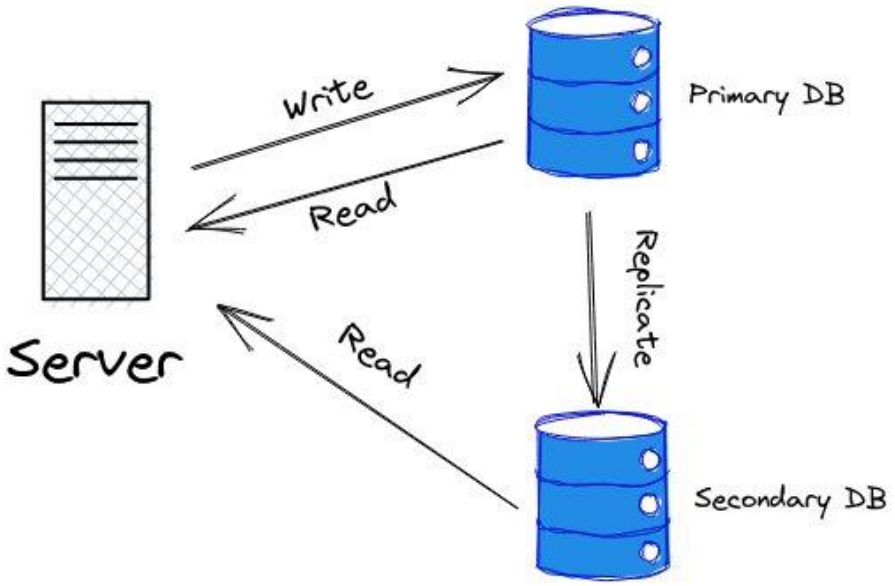

Figure 11: Read replication process. Data is asynchronously replicated from the primary instance to secondary ones, allowing for concurrent read requests from multiple instances. [8]

4. Development and Integration

4.1 API Development

In order to achieve the desired backend functionality of our application, a series of API routes has been implemented based on the principles of the Representational State Transfer (REST) architecture. REST provides a standardized and straightforward approach for building web services, which is widely adopted due to its simplicity, scalability, and ease of integration.

The implementation of these API routes will facilitate the exchange of data and actions, allowing users to interact with the application's functionality, hence providing a seamless communication medium between front-end and back-end.

Table 1 presents the API routes produced that define the endpoints that users can perform various operations, such as retrieving data, or updating resources within the application.

Name	Method	Request Parameters	Request body [JSON]	Response
/emails	GET	page=Int		{error, totalEmails, emails: [{subject, bodyPreview, id, sender,time}]}
/emails/{id}	GET		{error, email: {subject, body, sender, cc, bcc, ics}}	
/emails/getByCategory	GET	page=Int, category=String		{error, totalEmails, emails: [{subject, bodyPreview, id, sender, time}]}
/emails/changeCategory/{id}	POST		{newCategory}	{error}
/emails/getSummary/{id}	GET			{error, summary}
/metrics/recordTime/{id}	PUT		{timeSpent}	{error}
/metrics/recordClick/{id}	PUT			{error}
/pref/updatePreferences	POST		{whitelist, prefList}	{error}
/pref/getPreferences	GET			{error, prefList, whitelist}
/emails/dailySumary	GET			{error, summary}
/emails/generateICS/{id}	GET			ICS file
/emails/smartSearch	GET	searchString =String		{error, totalEmails, emails : [{subject, bodyPreview, id,sender, time}]}
/emails/search	GET	searchString =String		{error, totalEmails, emails : [{subject, bodyPreview, id,sender, time}]}

Table 1: List of API routes implemented

As previously stated in the backend methodology section, to ensure the reliability and correctness of our API routes, we conducted thorough testing using Postman. Figure 5 shows a screenshot of the Postman user interface along with our testing suite. All tests written will also be submitted along with the source code.

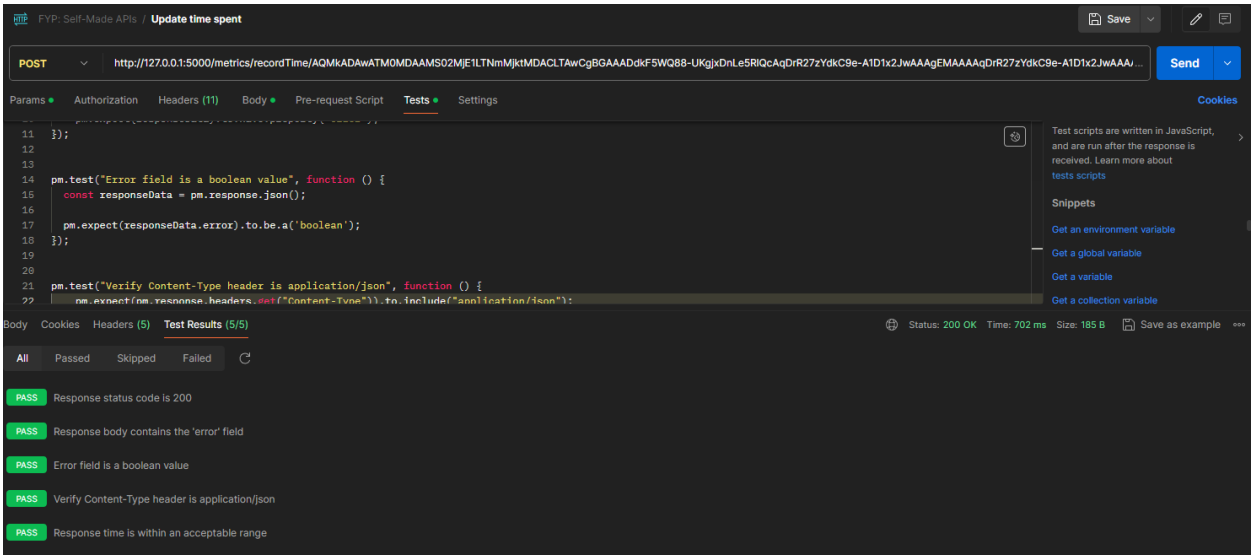

Figure 12: Postman user interface and our testing suite

4.2 Integration with Front-End and AI Services

This section discusses the details on how we integrated the backend with both the front-end and AI services in detail, including how we tested connectivity and conducted integration tests.

4.2.1. Front-End Integration

To enable the integration of the front-end with the API routes in the backend, we needed to specify Cross-Origin Resource Sharing (CORS) policies. CORS is an universal security mechanism enforced by web browsers that restricts cross-origin HTTP requests made by client-side scripts. CORS policies were implemented using Flask's "flask-cors" library, which offers CORS policy configuration in Flask with a simple API.

In addition to CORS policies, because our application required the use of custom headers in HTTP requests, CORS preflight checks had to be implemented as an additional requirement. These preflight checks are initiated by the client in the form of an OPTIONS request sent to the server prior to the actual request. The server's responses to these preflight checks provide crucial information such as the required methods, headers, origins, and other relevant fields [15]. Such a process is illustrated in Figure 6 below.

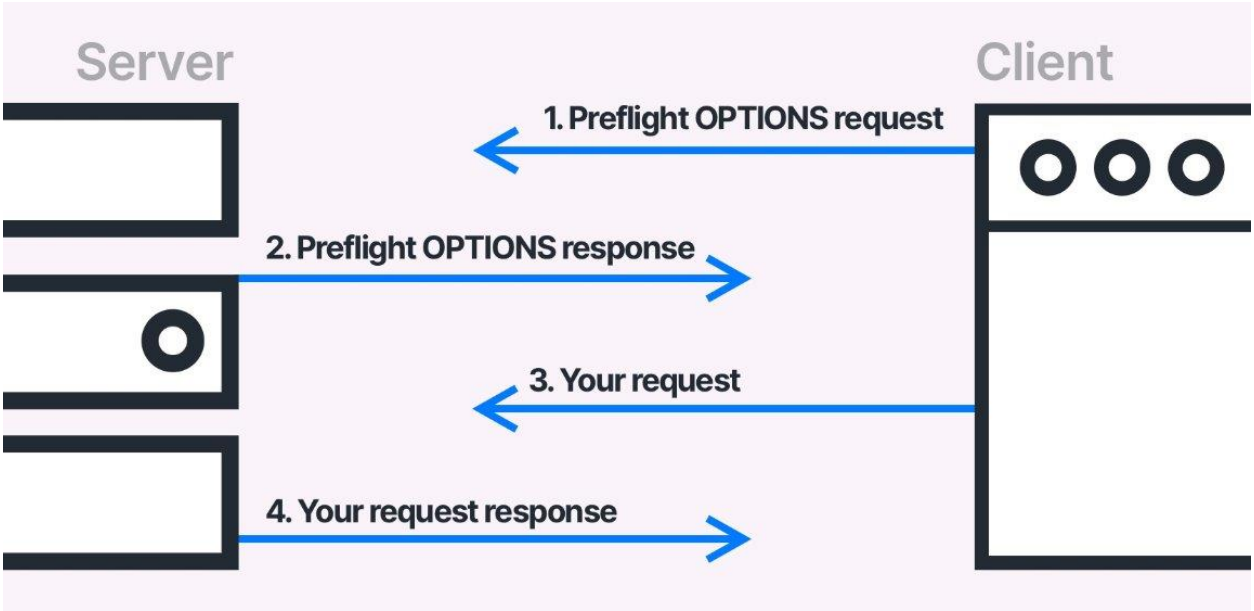

Figure 13: CORS preflight check process flow. An OPTIONS request must be answered before the actual request can be sent. [9]

Both CORS policies and preflight checks were programmatically implemented following the principle of least privilege to adhere to cybersecurity best practices. Following such an approach ensures that the client was granted only the necessary privileges and access rights required to perform its intended functions. This helps minimize potential security vulnerabilities and restrict unauthorized access or misuse of sensitive resources [16].

To allow the front-end to test connectivity with the backend, we also implemented a simple health-check endpoint that the front-end is allowed to ping before a request is sent to ensure the backend is live.

4.2.2 AI Service Integration

Since the AI service was implemented as a Python module, we were easily able to integrate it with the backend by simply importing the module and using its functions as with any other module or package. Ease of integration is the main advantage of implementing the AI service as a Python module. Python's modular design and extensive ecosystem make it simple to incorporate external functionality into existing projects. The AI module seamlessly integrated into our backend architecture, allowing us to leverage its powerful features.

To ensure the smooth integration of the backend with the AI module, we conducted comprehensive integration tests using the pyTest framework. pyTest is a popular testing framework in the Python ecosystem known for its simplicity and flexibility. It provided us with a reliable and efficient way to validate the integration of our backend code with the AI module. By leveraging pyTest, we were thus able to automate the integration testing process, saving time and ensuring consistent and reliable results.

5. Additional Improvements

5.1 SPF and DMARC Checking System

To improve the efficiency of our AI categorization, we implemented an SPF and DMARC checking system on the backend. Sender Policy Framework (SPF) and Domain-based Message Authentication, Reporting, and Conformance (DMARC) are email authentication protocols used for identifying spam and potential phishing attacks.

- SPF records specify which mail servers are authorized to send emails on behalf of a particular domain. Thus, when an SPF check passes, a recipient’s mail server is able to verify that the sender’s mail server is authorized. In the case of a failure, the email is usually marked as potential spam [17].

- DMARC provides a method for email senders to specify their email authentication policies and recipients to enforce those policies. It allows domain owners to publish a DMARC policy in their DNS records, which instructs receiving mail servers on how

to handle emails that fail authentication checks [17]. Similarly to SPF, when a DMARC test is passed, a sender's mail server can be deemed with a high probability as authentic.

We initiated the checking of SPF and DMARC certificates by sending recursive DNS queries to the sender's mail server, which was obtainable from Microsoft's Graph API. By doing so, the backend server is able to obtain SPF and DMARC certificates corresponding to the sender's mail server. Figure 7 shows this process in more detail.

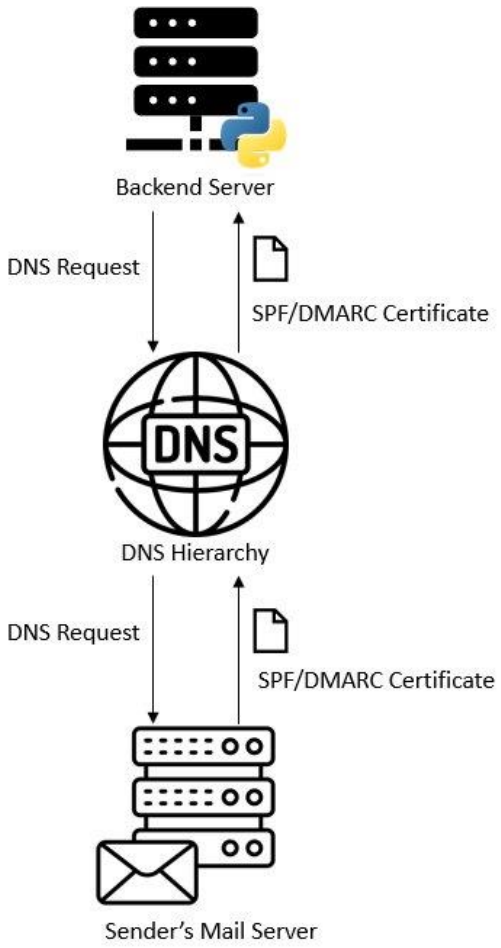

Figure 14: DNS querying in SPF & DMARC checking

The failure results from this SPF and DMARC checking system were used to negatively tilt the AI module's categorization of emails. This seeks to improve on our system without skewing results too much in the case of a false negative result. The formula for this tilt is as follows:

$$Final\ Category = AI\ Category - Number\ of\ Failed\ Checks$$

The AI category is a number between 0 to 10 inclusive. An example of this tilt in action is as follows:

> If an email fails both SPF and DMARC checks and is assigned an AI category of 8 (which represents a 'Most Important' rating), its final category will be reduced to a 6 (which represents a 'Less Important' rating). This reflects the higher probability that the email could be spam since its sending mail server is not authenticated as per results from the SPF and DMARC checks.

6. System Architecture

To conclude the technical discussion for the backend, we present a top-down view of our application's system architecture. Figure 8 shows the overall system architecture of our application.

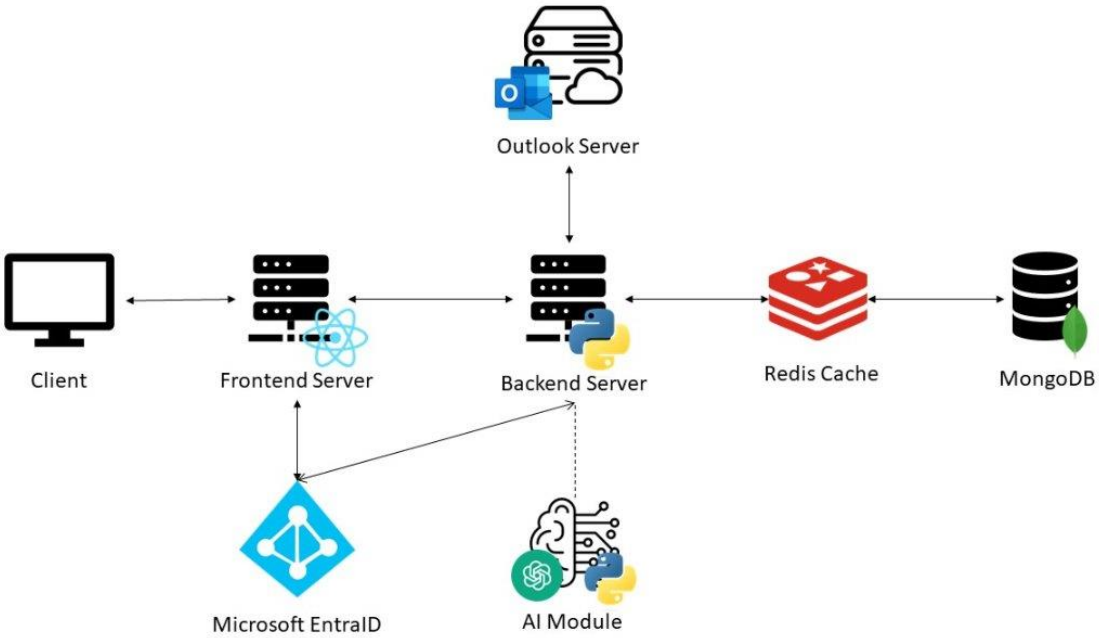

Figure 15: Overall system architecture of our application

Below is a breakdown of the system architecture:

- The client (or user in this case) only interacts with the frontend server
- Both the frontend and backend servers interact with Microsoft Azure EntraID to effectively carry out the OAuth 2.0 authorization flow
- The frontend server interacts with the backend server via REST APIs
- The backend server
    - Integrates the AI module to carry out the categorization of emails and other AI-related functions
    - Interacts with the MongoDB Atlas database with a Redis caching layer
    - Interacts with Outlook's servers via the Microsoft Graph API after an access token is obtained following the OAuth 2.0 authorization process

7. Conclusion

This report has provided an in-depth overview of the various backend technologies and implementations used in our development of Inbox Genius. Throughout the development process, careful consideration was given to selecting appropriate backend technologies that aligned with our project requirements. The utilization of robust and scalable technologies such as Flask and MongoDB has enabled us to build a highly efficient and reliable system, which will be of utmost importance to users. These technologies have not only facilitated seamless communication between the frontend, backend and AI services but have also contributed to the overall responsiveness and security of Inbox Genius.

References

[1] M. Grinberg, Flask web development: Developing web applications with Python. O'Reilly Media, Inc, USA, 2018.

[2] Viduka D. et. al., Choosing the best Python web framework for beginners according to experienced users. International Conference, p. 100, 2021.

[3] C. Gyorodi et. al., "A comparative study: Mongodb vs. MySql," 2015 13th International Conference on Engineering of Modern Electric Systems (EMES), 2015. doi:10.1109/emes.2015.7158433.

[4] "MongoDB Atlas: The multi-cloud developer data platform," MongoDB Atlas, https://www.mongodb.com/atlas (accessed Nov. 21, 2023).

[5] "Introduction to Redis," Redis, https://redis.io/about/ (accessed Apr. 26, 2024).

[6] "Redis-py," Redis, https://pypi.org/project/redis/ (accessed Apr. 26, 2024).

[7] "Microsoft identity platform and OAuth 2.0 authorization code flow," Microsoft Learn, https://learn.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-auth-code-flow#applications-that-support-the-auth-code-flow (accessed Oct. 17, 2023).

[8] M. Jones & D. Hardt, The OAuth 2.0 authorization framework: Bearer token usage, 2012. doi:10.17487/rfc6750.

[9] M. Anicas, "An introduction to OAuth 2," Digital Ocean, https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2 (accessed Oct. 17, 2023).

[10] "React single-page application using MSAL React to authenticate users against Azure AD for customers," Microsoft Learn, https://learn.microsoft.com/en-us/samples/azure-samples/ms-identity-ciam-javascript-tutorial/ms-identity-ciam-javascript-tutorial-1-sign-in-react/ (accessed Nov. 21, 2023).

[11] "Microsoft Authentication Library for React (msal-react)," npm, https://www.npmjs.com/package/@azure/msal-react (accessed Oct. 17, 2023).

[12] “MongoDB Atlas Database,” MongoDB, https://www.mongodb.com/atlas/database (accessed Apr. 18, 2024).

[13] T. B. Septian, “Redis – Cache-Aside Pattern,” Blibli.com Tech Blog, https://medium.com/bliblidotcom-techblog/redis-cache-aside-pattern-72fff2e4f927 (accessed Apr. 14, 2024).

[14] M. A. Ayodeji, “Postgres Database Replication in Django,” Earthly, https://earthly.dev/blog/set-up-postgresql-db/ (accessed Apr. 18, 2024).

[15] J. Pomykala, “What is CORS,” SimpleLocalize, https://simplelocalize.io/blog/posts/what-is-cors/ (accessed Apr. 11, 2024).