Debias

EECS4314 - Advanced Software Engineering

Professor Marin Litoiu

Prepared by - The Agile Alliance

Team members
Siddharth Bhardwaj
Michael Borg
Ruizhe Liu
Arshdeep Saini
Hengchao Xiang

Table of Contents

Abstract

Debias is a news aggregation tool. It aims to solve the difficulty in finding articles with different viewpoints by aggregating daily news articles and displaying their tone to allow users to differentiate articles pertaining to a topic based on their tone. Users can search for news articles in the front end, and articles pertaining to these topics will be selected from the database of processed news articles and returned to the user displaying the name, author, date, of the article as well as scores indicating the tone of the articles. The users can further filter down the articles within the topic by the tone, which can allow them to get a wider range of viewpoints from a topic.

The motivation came from the need to make it easier for individuals to be able to filter their news feed in a way that will allow them to see articles with different tones, which could give them a more balanced understanding of the topic they are searching for.

We developed a conceptual architecture for Debias in the last project phase. In this phase, we derived the concrete architecture, and deployed a functional prototype accessible over the internet.

Concrete System Architecture

Generally speaking, the entire application architecture still maintains a three-tier architecture style, combined with microservice and the use of some cloud services, such as Watson Tone Analyzer to provide the tone score of the article, Firebase Cloud Messaging to assist the implementation of subscription, the subsystem chooses other architecture styles based on its functions, such as pipe and filter style. The front-end adopts SPA mode, which allows us to accelerate the development with the help of the front-end framework , and it interacts with the back-end through REST API. The back-end uses the Flask framework and python for development. On the database, we chose mongoDB to store three types of data: processed article data, user data, and user feedback data.

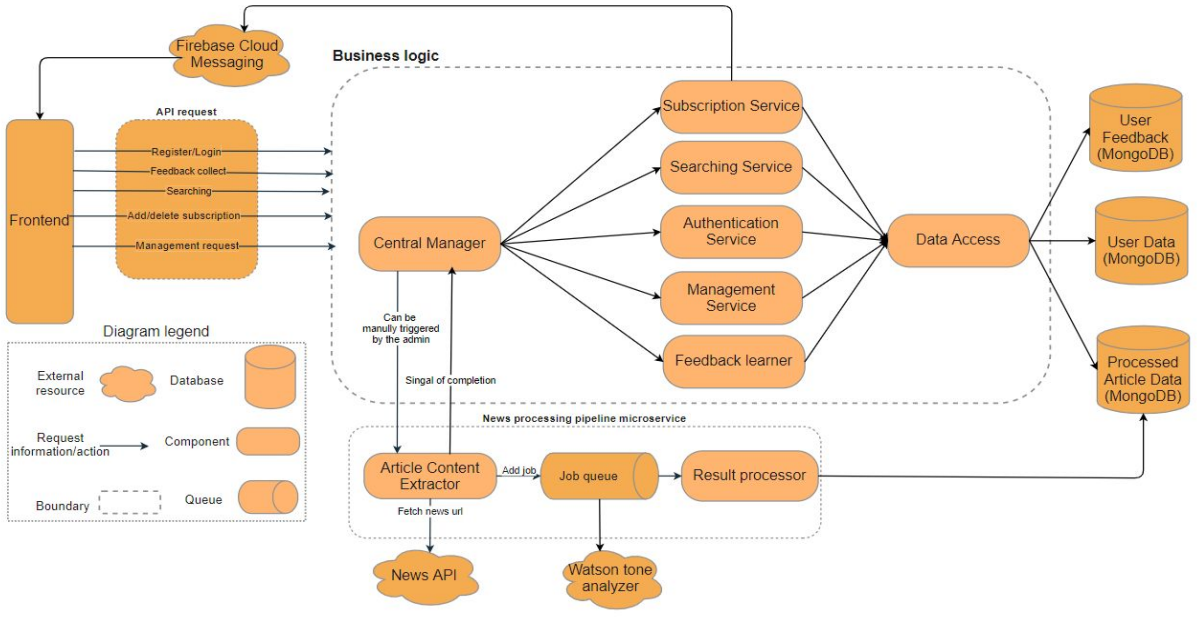

Figure1: Concrete architecture of Debias

The decomposition of the entire architecture is as follows:

- Front-end, since the entire application is a single-page-application, the front-end only interacts with the back-end through REST API, the back-end will expose different API endpoints to respond to different requests, for example the /articles/<tone> to get articles based on the tone.
- Most of the back-end adopts the monolithic mode, combined with a microservice. Since the functions of the subsystem are basically independent,when the development team has sufficient capabilities, the possibility of migrating to microservice is retained. And we selected the corresponding cloud service to achieve some functions:
    - Watson Tone Analyzer
      We communicate by the API provided by the Watson Tone Analyzer to get the tone scores of an article by sending it the article content. The Watson Tone Analyzer is a service on IBM's cloud platform that uses linguistic analysis to detect various tones found in the submitted text, at both the document and sentence level. For our current purposes we strictly use the tone of the overall document. The major benefit of using the Watson Tone Analyzer is that it uses AI to enhance increase the likelihood of determining the correct tone in a sentence, the downside however is that because this service is not open source or modifiable, we cannot make changes to the source code that might help ensure that the result is more suited to our needs. One alternative to using the Watson Tone Analyzer is to use a library such as 'nltk' (Natural Language Toolkit) for python. This library allows us to classify text, or parts of text into a pre-defined sentiment, in our case this would be the tone of the article. We could classify various sentences from news articles according to their tones and use this dataset in order to train a model on these pre-classified sentences and use this model to classify the sentences in the article into their corresponding tone. Using the tone from all the sentences of the article would then allow us to determine the overall tone of the article. We chose the Watson Tone Analyzer over this method due to it's more robust determination of a sentences tone, as well as time constraints as creating a dataset large enough to train a model would take a great deal of time.
    - Firebase Cloud Messaging
      Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that allows us to publish the resource based on the topic and takes the responsibility to push notifications to the subscriber's devices.

      An alternative way of doing this pub/sub subsystem would be building it ourselves using python+redis+websocket or some other tech stacks, but doing this will significantly increase the complexity of the implementation and this will take too much effort probably just to support the web browser notifications, then in the future we have to build it for mobile devices separately. FCM already provides the support for web/Android/IOS and it also provides the support for subscribing to topics which perfectly meets our needs.

- Considering that we do not have complicated operations between tables, and the connection between standards is relatively weak, we chose NoSQL data database according to the database form design. The two candidates for our database are mongoDB and DynamoDB. The main reasons we chose MongoDB instead of DynamoDB are because, first DynamoDB supports key-value queries only, considering we may need to do some complex queries and various aggregations of the data due to the functionality like subscription, DynamoDB will limit the query flexibility. Second, DynamoDB provides limited data type support, types like date and timestamp are not supported by nature which will increase the application complexity. Finally, currently we do not heavily rely on the AWS stack, which also does give a strong reason to choose dynamoDB.

Differences Between Conceptual and Concrete Architectures

We maintained the structure of the conceptual architecture in terms of the overall architecture style, but at the same time made the following adjustments:

- Considering that in the previous conceptual architecture, almost every subsystem needs to interact with the database, we have added the data access component to provide a common database CRUD interface for other subsystems, that is, the data access component will be responsible for the underlying connection with the actual database and exposes common CRUD interfaces to other subsystems, so that the remaining subsystems do not need to consider the low level connection with the underlying database, which also realizes the decoupling of the database and other subsystems, and gives us the flexibility to replace the underlying database, so we only need to modify the database access part of data access and other subsystems are not affected.
- We removed the article content database in the conceptual architecture. Because the API of the Watson Tone Analyzer requires us to provide the content of the article, and news APIs such as The New York Times, only provide us with the URL of the news, the previous approach is extracting the article content and storing it in the database and then sending it to the Watson Tone Analyzer for analysis. But in fact, most news articles are not very large in terms of size and the extracting process doesn't take too long to become a bottleneck here. In order to improve the performance, we then extract the content of the article directly and send it to the tone analyzer to save the time of storing/reading the database.
- We isolate the News processing pipeline as an independent microservice. First of all, it can be seen from the conceptual architecture that the coupling between this subsystem and other subsystems is very low, and it is fully capable of running independently. Secondly, as the news API continues to increase, this part may often need to be changed to adapt the new news API. If it is used as a part of the monolithic backend, frequent changes will lead to large effort of the redeployment. Finally, considering the scaling of the back-end, the redundancy of this subsystem inside the monolithic backend does not lead to better performance and load-balancing. However, leads to increased system fault tolerance with the addition of this independent microservice.

Deployment Architecture

The application is deployed on an AWS EC2 instance which is an ubuntu virtual machine. And it's also containerized and the different parts are running in different containers. From the deployment diagram, we can see that there are 6 main parts. UI is a Svelte web service, which clients can interact with the system through. The Flask web service works as backend REST APIs, which also provides services like searching, authentication, management, etc. News processing service is in charge of article collecting. UI and news processing communicate with the backend using HTTP protocol. The three MongoDB databases are deployed in different containers and they interact with backend and news processing using MongoDB wire protocol which is a simple socket-based, request-response style protocol.

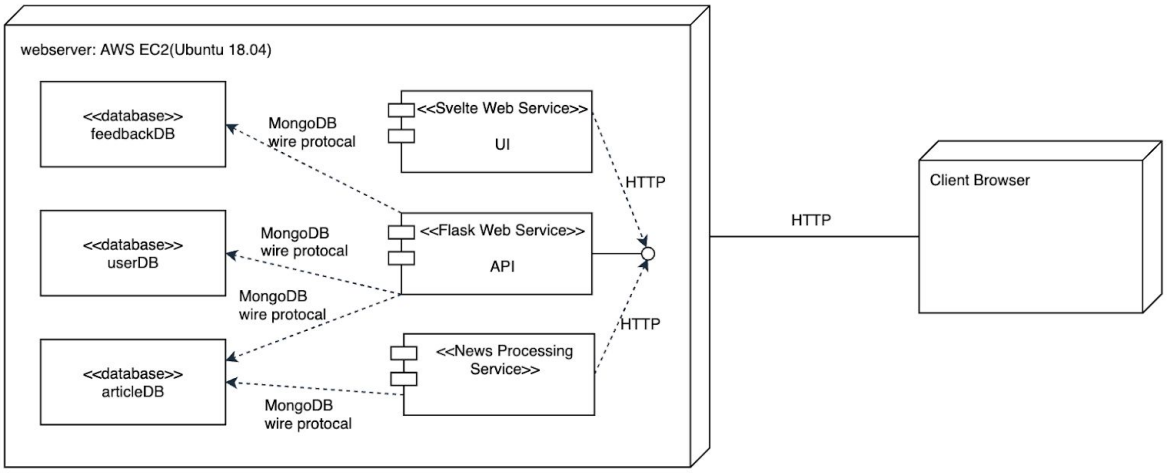

Figure 2. Deployment diagram of Debias

In terms of concurrency, the flask has the option to handle the users' requests in different threads, so that it can handle multiple requests at the same time. For news collecting, the news sources will be periodically querying for new news articles and placing them in a queue where it will await processing via the WATSON Tone Analyzer. Once the response has been retrieved from the WATSON Tone Analyzer, the response will be processed and inserted into a MongoDB database. MongoDB uses reader-writer locks that allow readers shared access to a resource and give exclusive access to a single write operation. So we can have multiple simultaneous readers on the database. For our application, most users will use it to view the articles and search for some articles based on the tone and topic, which involves read operation and can be executed at the same time by the database.

Architecture of news processing pipeline

The news processing pipeline is deployed on our AWS EC2 instance, in the current version of our product it is not containerized and must be started up by using SHH to log into the instance and starting the python application. In a future version this pipeline would be containerized and would feature various API's that would allow administrators to change when the processing will occur. There would be various cron jobs running to constantly fill the database with new articles periodically, however this was not added to the current release due to limitations on the number of free API requests that can be made to the various services being used. At present the news processing pipeline retrieves articles utilizing the API's from various news websites, metadata is retrieved from the API

response and an array is created composed of article objects that contain metadata such as the URL, author, and the articles keywords. Once every article from the response has had its meta data retrieved, we utilize a library ‘Newspaper3k’ that parses an article to get the text body, this text is then submitted to the Watson Tone Analyzer using the python library ‘ibm-watson’, which returns a json object composed of the various tones it detected in the overall article. The tone retrieved from the Watson Tone Analyzer is combined with the metadata from the article and is then stored in the NoSQL database using the articles URL as the key to ensure there are no duplicates, this data can be retrieved from the front end using a GET API request.

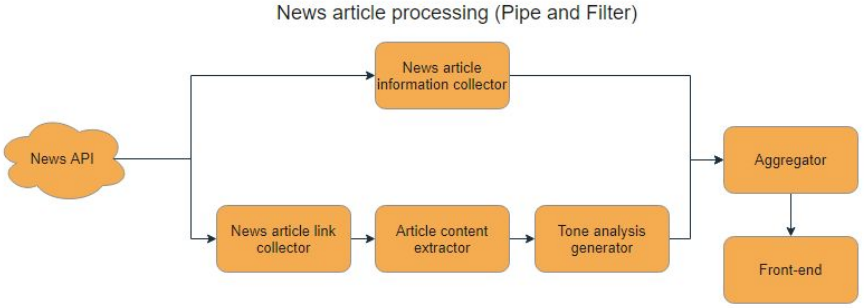

Figure 3. Architecture of News processing pipeline

Subsystems and their Interactions

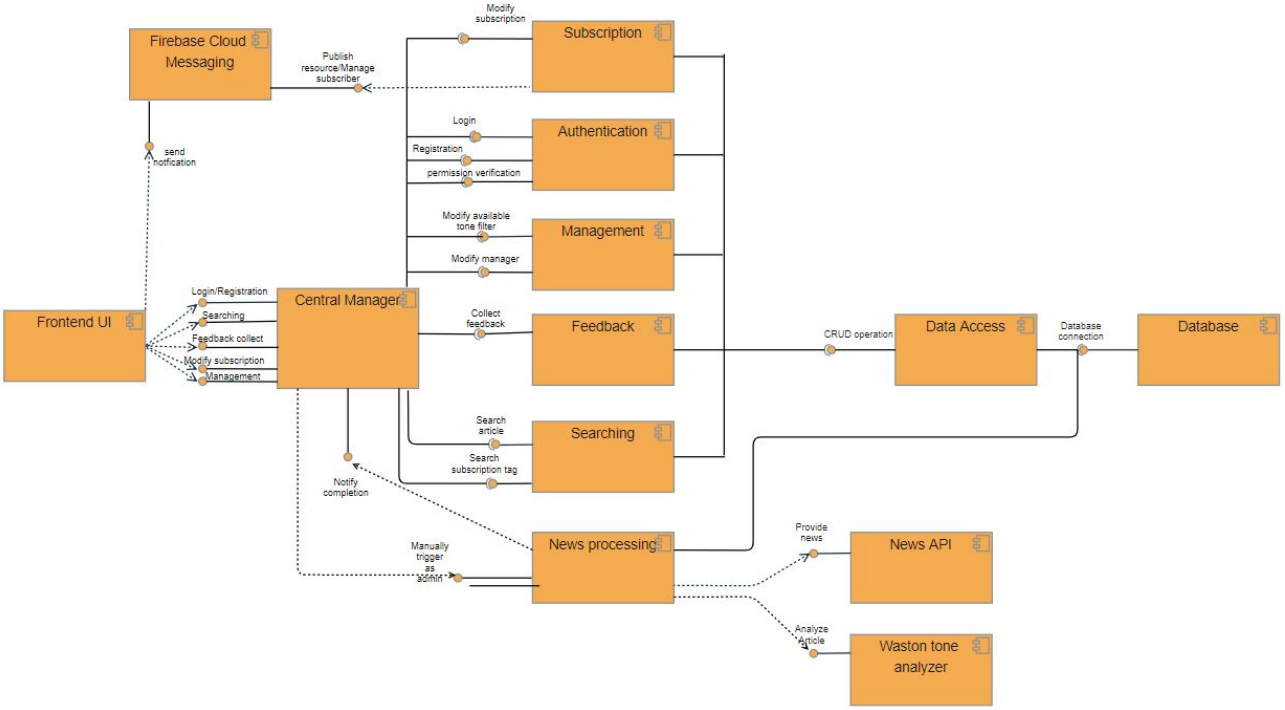

Figure 4. Component diagram of Debias

Central Manager

Responsible for handling the http request, and invokes other components to process corresponding requests. Including log/registration, searching, collecting feedback, modifying subscription and management requests.

News processing pipeline

It will be responsible for obtaining the URLs of corresponding articles one by one from the preset news sources, extracting the content, and then sent to the third-party tone analyzer for analysis. The returned results are processed by the Result processor and stored in the processed article database. This microservice has its own scheduler and should run periodically to update the articles in the database, every time it finishes the update, it should send a http request to notify the central manager so it can detect if this service is still running properly. Also, the admin of the system could directly trigger the update by the apis it provided.

Subscription Service

Connecting to the FCM to manage the user subscriptions and topics, responsible for publishing new messages to the FCM. Expose interfaces to the central manager to fulfil the request of modifying user subscription.

Authentication Service

Providing interface to handle the login and registration request, also provide a method for checking the user permission when a management request is received.

Management Service

Invoked by the central manager to fulfil the management request, exposes the interfaces of managing user permissions, the visibility of the article and the tone filters.

Feedback learner

Providing an interface for collecting user feedback data, storing it in the feedback database and performing learning on it, it will periodically apply the user feedback.

Searching Service

Searching service provides two types of search interfaces: search for news based on keywords with or without tone filters, search for topic tags for subscription.

Data Access

Connecting to the underlying database and providing generic CRUD interface to other subsystems.

Quality attributes, Design Tactics and Trade-off Decision

Quality Attribute	Requirements/design decisions related to the quality attribute	Design Tactic
Security	The system should notify the users to change the password every four months and the previous passwords can never be used again.	

	Encryption in-transit (eg.HTTPS) and at-rest (eg.encryption of DB) will be used for privacy.	Encrypt Data
	The system should verify the user's identity before allowing them to use management functions.	Identify Actors
Tradeoffs: We designed the authentication service as a subsystem to be responsible for authenticating users, sacrificing the usability of the systems to urge users to change passwords regularly, and at the same time, adding permission checks to the APIs that are able to make some major changes to ensure the security of the system by sacrificing certain performance.		
Performance	The 'searching articles' use case should have a response time less than 2 seconds, 90% of times.	Multiple copies of computation
	The system should be able to process 1000 user requests per second in peak load.	Multiple copies of computation
Tradeoffs: We have made some trade-offs between system modifiable, maintainable and performance. We can increase performance by adding the cache of commonly used queries, and more complex search algorithms to improve performance, but those will increase the difficulty of maintenance and development. On the premise that the development team is small, we will postpone the implementation of these plans.		
Availability	The system should not be unavailable more than 1 hour per 1000 hours of operation.	Redundancy
	News processing pipeline should send a signal to the central manager every time it finishes the scheduled update and this signal should be logged.	Heartbeat
Tradeoffs: Through decoupling our application components we improved availability as the development team can perform Canary deployments (detailed in data dictionary) using containers, and AWS. Furthermore, we made a trade-off between security and availability- choosing to ensure security (allow ping request to production server) does not hinder (heatbeat) communication between components.		
Modifiability	Design decision: News processing pipeline requires frequent changes so it gets isolated as an independent service to reduce the overhead of making changes.	Reduce coupling
	Design decision: Data access component is responsible for the low level connection with the database and exposes generic interface of CRUD operation to other subsystems.	Abstract common services
Tradeoffs: Here however we decide to reduce coupling due to frequent code changes required on News Processing pipeline component- tradeoff between availability and modifiability. Again, Canary Deployment is used to mitigate this trade-offs effects.		
Scalability	Database image snapshots can be taken, and then deployed on new application servers to handle many users interfacing with one website (frontend) with multiple backend instances (databases). These details will be abstracted away for user convenience.	Containers running on Amazon ec2

Third-party Libraries, APIs and COTs

Some APIs integrated into Debias are: New York times API for articles, Google, Amazon, facebook API for Federated Identity, IBM Cloud SDK APIs for Tone Analyze. All the aforementioned APIs are RESTful API-- based on HTTPS requests and JSON responses. Thus, Debias since communicates with the above external interfaces using REST, it is a RESTful system. In essence, this allows Debias to support HTTP inputs from the user via request making implemented by the REST server.

Furthermore, we also used Firebase Cloud Messaging. Firebase Cloud Messaging (FCM) is a SaaS by Firebase (developed by Google), which provides pub-sub functionality to developers through the use of the FCM software development kit (SDK). FCM SaaS was used to create pub-sub functionality (in the system) without “reinventing the wheel”.

Conclusion

In conclusion, a functional prototype was implemented in this phase. This prototype is the concrete architecture, which was derived from the conceptual architecture from the last project phase. There were only three modifications from the conceptual architecture: adding the data access component for flexibility and convenience, removing the article content database to remove bottleneck and improve performance, and isolating the News processing pipeline as an independent microservice to increase the flexibility of scaling and fault-tolerance. The functional prototype captures the use case where- a client interacts with the system to obtain aggregated news articles based on tone, using keywords and tone filters in their query/search request. In essence, this report provided details the implemented high level architecture, the quality attributes, design tactics and trade-off decisions made in the derivation of the concrete architecture of Debias, the concrete system’s architecture style(s)- three tier, pub-sub, pipe and filter (which did not change from the conceptual architecture), and microservices architecture (which we derived as we implemented the concrete system of Debias), and how it evolved from the conceptual architecture, the concrete system’s subsystems and their interactions (communication protocols, interfaces, etc), the third-party libraries, application programming interfaces (APIs), and commercially available off-the-shelf (COTS) solutions used in the concrete architecture of Debais, the lessons learned by the team while deriving the concrete architecture from the conceptual architecture, and the architecture/design exploits of new technologies. In the “Functional Prototype” section below, a URL to the implemented Debias system can be found.

Lessons Learned

In consideration of reducing the complexity of development and deployment and maintenance, we start planning the entire architecture from the monolithic model, which really makes it easier for us to plan the subsystems, corresponding deployment strategy and the scaling without having to consider the complex communication issues between various services at the beginning, we basically only need to consider the horizontal scaling of a single layer.

But as we consider more quality attributes and observe the interaction of subsystems, we gradually find that some subsystems are inherently less coupled with other subsystems and are more suitable for independent running. At the same time, monolithic limits the flexibility of scaling, because we must scale the entire backend, not just target on the

certain subsystems that may become performance bottlenecks. But the microservice architecture does reduce the reusability of the code, for example, we must connect to the database multiple times. Therefore, even if you start with a monolithic architecture, it is best to maintain the low coupling of subsystems as much as possible, so as to maintain the possibility of evolution to microservices based on the needs.

Exploits of New Technology

In the implementation of the concrete architecture of Debais, we explored and got hands-on with the Amazon Web Services cloud- specifically, Elastic Compute Cloud (ec2), Firebase Platform- specifically, Firebase cloud messaging, and containerization technology using Docker.

The exploration and usage of FCM is detailed in the “Third-party Libraries, APIs and COTs” section above. Using docs.aws.amazon.com/ and docs.docker.com/ our team utilized Amazon ec2, we deployed a Ubuntu 18.04 virtual machine with a publicly accessible internet protocol (IP) address. Thus, the code for all the system components, and subsystem components was written, compiled and built into executables. Thereby, we imported our frontend and backend code into Github, then cloned it onto the ec2 instance. Thereafter, using docker compose we containerized our backend, frontend and database into different containers to integrate between frontend and backend. In essence, our application/service is hosted on Amazon ec2, and is deployed as containers using docker.

Functional Prototype

Deriving the concrete architecture from conceptual, our team implemented a functional prototype that captures the use case model below.

Use Case Model

Figure5: shows the use case where a client uses keywords to search for articles (which are aggregated and presented to the client). Thereafter, the client can filter the presented articles by tone- combining different tone filters. Furthermore, can view the tone score associated with each article, displayed to the client by the system.

Activity Diagram

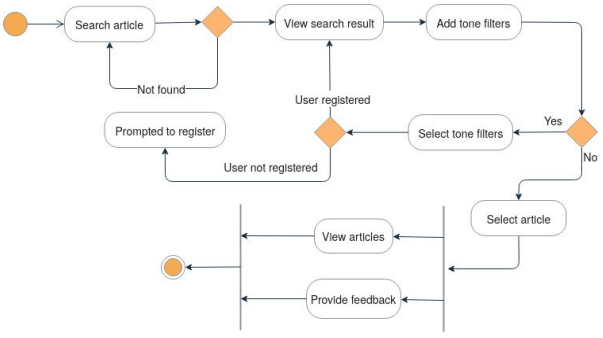

Figure6: shows the activity diagram for the use case diagram shown in FigureX above. The purpose of illustrating this diagram is to show the flow of data and control in the architecture.

Sequence diagram

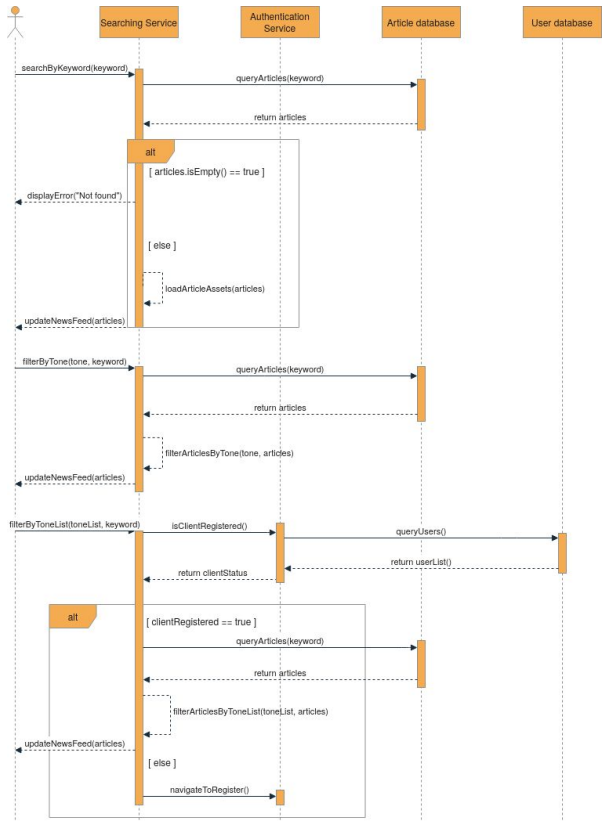

Figure7: shows the sequence diagram for the use case and activity diagrams shown above in FiguresX and X. The purpose of illustrating this diagram is to show the flow of data and control in the system, and interactions between system subcomponents involved in usecase used to implement the functional prototype.

Screenshot

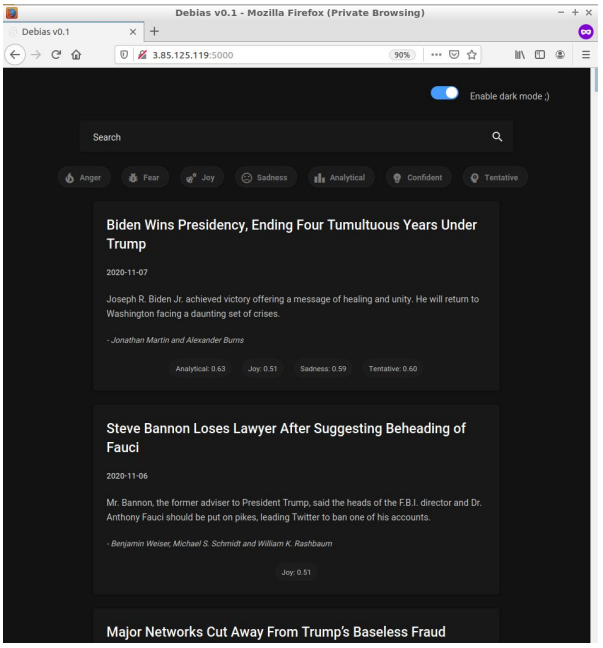

Figure8: shows the deployed functional prototype available as a service, accessible through the internet.

URL to resource

The functional prototype can be accessed through URL: http://ec2-3-85-125-119.compute-1.amazonaws.com:5000/ or TCP/IP Address: http://3.85.125.119:5000

Data Dictionary

Central Manager: subsystem component responsible for routing the api and function as a scheduler-periodically triggering the task of updating article data and coordinating the function of applying user feedback to the article metadata.

News processing pipeline: microservice composed of three separate subcomponents- article content extractor, a job queue, and result processor. It is responsible for processing the article and presenting it to the user.

Feedback learner: subsystem component responsible for collecting user feedback data. Communicates with the Feedback database and Central manager to manage user feedback.

Authentication Service: subsystem component that communicates with the central manager to handle registration and login requests- following NIST guidelines to ensure compliance with security and privacy requirements.

Searching Service: subsystem component that relies on the article metadata database to provide search functionality. Uses keywords, tone filters, and topic tags (for subscription) to search.

Subscription Service: subsystem component that interacts with the central manager to handle requests to adding or deleting a topic tag from the front end, and manage publications and push subscriptions requests.

Management Service: subsystem component responsible for fulfilling the management requests of administrators and managers. Interacts with user data database and article metadata database to handle functions invovinging managing user permissions, article visibility to Clients, and visibility of tone filters available to Clients.

User Data Database: Stores the user information, including user credentials, subscription list information etc.

Processed Article Database: Main database of the system, will store the article information with tone analysis scores, visibility, abstract, url and other necessary information.

User Feedback Database: Stores the user feedback data

Agile: Software development in which requirements are discovered and solutions are discovered through the collaborative effort of self-organizing and cross-functional teams, and the customer/end user.

User: a Client of the System. Extents WebUser.

Canary Deployment: is a pattern for rolling out releases to a subset of users or servers. The idea is to first deploy the change to a small subset of servers, test it, and then roll the change out to the rest of the servers. In essence, through our team's use of containers and AWS we can perform staged deployments of our application (running behind a AWS provided load balancer) on ec2 servers.

Naming Conventions

TCP/IP address: **T**ransmission **C**ontrol **P**rotocol over **I**nternet **P**rotocol address. is a suite of communication protocols used to interconnect network devices on the internet.

SaaS: **S**oftware as a **S**ervice- The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface.

COTS: Commercially available **O**ff-**T**he-**S**helf**.** Software and hardware that already exists and is available from commercial sources. It is also referred to as off-the-shelf.

URL: Uniform **R**esource **L**ocator. Is a short string containing an address which refers to an object in the "web." URLs are a subset of URIs

API: Application **P**rogramming **I**nterface. Is an intermediary software that allows two applications to communicate with each other.

REST: REpresentational **S**tate **T**ransfer. A software architecture style that defines a set of constraints when creating web services.

SOAP: Simple **O**bject **A**ccess **P**rotocol. It defines an RPC mechanism using XML for client-server interaction across a network

MVC: Model-**V**iew-**C**ontroller. Architectural pattern commonly used in developing client-facing applications.

NIST: The **N**ational **I**nstitute of **S**tandards and **T**echnology. Is a government entity that helps organizations to better understand and improve their management of cybersecurity risk.

Scrum: is an agile framework that helps teams work together. Encourages teams to learn through experiences, self-organize while working on a problem, and reflect on their wins and losses to continuously improve.

References

1. “Guide to Secure Web Services”. The National Institute of Standards and Technology. Retrieved from: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-95.pdf
2. “Tone Analyzer”. IBM Cloud Docs. Retrieved from https://cloud.ibm.com/docs/tone-analyzer?topic=tone-analyzer-about
3. Account Plans. IBM Cloud. Retrieved from: https://www.ibm.com/cloud/free
4. “Agile 101”. The Agile Alliance. Retrieved from: https://www.agilealliance.org/agile101/
5. “Software Processes Requirements”. Marin Litiou’s 4314 class slides. Retrieved from:https://eclass.yorku.ca/eclass/pluginfile.php/775191/mod_resource/content/2/Lecture%203-%20Software%20Processes.pdf
6. “Scrum”. Atlassian Agile Coach. Retrieved from https://www.atlassian.com/agile/scrum
7. Send messages to topics on Web/JavaScript. Retrieved from https://firebase.google.com/docs/cloud-messaging/js/topic-messaging
8. “Computer Science Resource Center”. The National Institute of Standards and Technology (NIST). https://www.nist.gov/
9. Amazon EC2. Amazon Web Services. https://docs.aws.amazon.com/ec2/
10. “AWS Comprehend”. Amazon Web Services. https://docs.aws.amazon.com/comprehend/latest/dg/functionality.html
11. “Docker Docs”. Docker Inc. https://docs.docker.com/
12. “Firebase Cloud Messaging (FCM)”. Google Developers. https://firebase.google.com/docs/cloud-messaging/fcm-architecture
13. Comparing MongoDB and DynamoDB. Retrieved from https://www.mongodb.com/compare/mongodb-dynamodb
14. “What is TCP/IP?”. Cloudflare. Retrieved from https://www.cloudflare.com/en-ca/learning/ddos/glossary/tcp-ip/
15. “Canary Deployment”. Octopus Deploy. https://octopus.com/docs/deployment-patterns/canary-deployments