Data Engineering: A Full Stack Approach to Monitoring Data Freshness

CS 4991 Capstone Report, 2022

Aditya Penmesta
Computer Science
The University of Virginia
School of Engineering and Applied Sciences
Charlottesville, Virginia USA
ap5pp@virginia.edu

Abstract

Flowcode, a data centric offline to online startup at which I interned last summer, found itself ingesting a lot of its data without any centralized, automated place for employees to check the status of these data sources. To address this issue, I built a full stack web application that serves as a data status dashboard, automatically updating every day with the last time data was pulled for every source in the Flowcode ETL pipeline. To design this solution, I created three different web scrapers that would hit Looker, dbt and Snowflake (the three platforms used in Flowcode's ingestion pipeline), and pull the last date in which each source was queried. Then I exposed this data in a custom API created using Django and developed a React front end to render the backend into a clean website. Finally, I deployed the website Docker and Kubernetes into a live, online service which could be accessed via the company's VPN configuration. After we implemented this solution, processes that previously took hours were shortened to minutes, making the job of data engineers at the company much easier. Future work to improve the product would be adding features to automatically detect changes in the status of sources and notifying data engineers, quickening the fetch speed of the website through cache optimizations, and stylizing the website better by using unique front end design concepts.

1. Introduction

Upon arriving at Flowcode for the summer, I was presented with a rather large and meaningful task for an intern. My manager was leading the current data engineering at the company, and recently started an internal, company-wide data migration from Azure to AWS, a tall task that was becoming rather problematic for him.

The biggest problem was that changes in cloud services required a lot of changes to references of downstream data references, and this would affect data using internal stakeholders that could affect vital company functions. As an example, one small migration affected a customer-facing dashboard by the sales team, and it took data engineers most of the working day to diagnose and fix the problem. In light of this mass migration and the problems it was causing for the company, my manager assigned me the task of diagnosing data errors with an easy-to-use full stack solution. After a couple days of deliberation and brainstorming, I decided the best solution was a status dashboard, which would go through all the data services that load and transform data and track whether or not each individual source had been–updated recently. In practice, this solution would simplify data problem diagnostics from an hours-long process to a matter of minutes, making it extremely valuable to the company.

2. Related Works

When coming up with the technical approach and design of the status dashboard, I drew inspiration from a couple of other status dashboard implementations which served as a model for my project. The first one I focused on was the Xbox Server Status Dashboard (2022) [1]. I found the overall layout of the website to be extremely simple yet effective, allowing users to find the information they want as quick as possible without wasting time. Specifically, I found the breakdown of the menu very intuitive, starting with services, then features, and then revealing the updates for each feature. In my implementation, I aimed to have a similar breakdown, first listing project folders, then specific dashboards and finally listing the tiles within each dashboard. This would allow for internal stakeholders to find outages within a matter of minutes and save the monotony of having to comb through all the individual data sources.

Another website I used for inspiration was the Fivetran status dashboard (2022) [2]. Fivetran is a data source connector, allowing for collection of data from specific services, also used internally for Flowcode's data ingestion processes. Since Fivetran's use case for the dashboard was similar to mine, I tried to find features that were useful that I could also implement to provide the most robust experience. A feature that I decided to use for my website was keeping a record of the last 90 days of status updates and displaying it on the main page, which would allow for data engineers specifically to address lingering issues. I also utilized the color scheme of errors, using red for sources that had not been updated in more than 100 days, yellow for sources that were updated in between 1 and 100 days, and green for sources that had been updated on the current day. This allowed for the same usability as Fivetran, providing an easily recognizable color scheme to recognize faults.

3. Process Design

Building this solution required many moving parts that eventually led to a seamless and robust process that addressed the problem description.

3.1 Review of System Architecture

In order to create this solution, I identified the three components I would use in my approach. First, I would develop scrapers that would track data from individual source tables in Snowflake to data models created in dbt and finally to sources as dashboards in Looker. This step would allow me to gather all the data necessary for the backend of the dashboard.

I would then start building out an API which would allow me to expose the data I collected in the backend to the frontend I wish to render. To accomplish this step, I would use Django, a Python framework used to quickly develop and iterate APIs, popular among groups that are trying to prototype a service rapidly. Finally, the data from the Django API endpoint would be fetched and rendered into a clean user interface, which would serve as the main landing page for internal users. This overall approach was refined over several weeks after iterating through an MVP and deciding which technologies would allow for the quickest creation of a complete product.

3.2 Key Components

In order to build a solution, three components had to be created: scrapers to pull in the necessary data, an API endpoint to expose privately, and a front end to render the information scraped.

3.2.1 Data Scrapers

To scrape the appropriate data needed for this project, I had to create scrapers that could gather data from dbt and Looker, two applications used internally that loaded and transformed all the company's data. In order to develop these tools, I had to utilize the API libraries offered by both of these services. For both of them, I first gathered Flowcode's organizational credentials for the API services and used them to explore the metadata offered to users. I then identified key values that would be necessary to store within the organization's database, mainly the last time a certain table or model was updated.

Once I found these timestamps, I consolidated the data needed into a Pandas (A popular Python library used to store and manipulate data) dataframe and utilized an internal connector to Snowflake. This pre-built tool allowed me to take any local Pandas dataframe and dump it into a table within Flowcode's internal Snowflake instance. After building the connection between the results of the API and Snowflake, I had to automate the process using Airflow, an orchestrator which makes a job run at a daily cadence, allowing for data to constantly be refreshed and updated. This final step of the data scrapers component allowed for all the relevant data to be captured and updated on a schedule that will always leave users with the data they need.

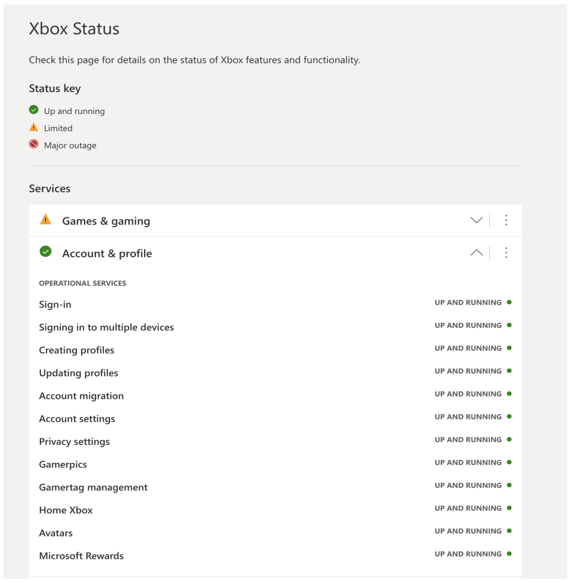
3.2.2 Django API

The next step was to create an API which could take the data in Snowflake and expose it to an endpoint which a front end could use. Within software engineering, this is common practice for connecting a database to a front end. This is for both security reasons and since it is impractical to have a live connection to any data source. I decided on Django since it is a lightweight Python framework which would be easy to configure and develop rapidly. This proved to be the case, as all that was needed to load in the data was the Python-configured Snowflake connector which allowed for a live connection to the Snowflake tables where my data was stored.

Then, I used Django's built in functionality for API development to develop a public view which would allow my front end to interact with. After developing the front end however, I changed the Django endpoint from publicly available to a private connection, only allowing stakeholders with the correct credentials to access it.

3.2.3 React Front End

To render all the data in a visually appealing front end, I settled on React. React is a Javascript framework which allows for front end development with less clutter than most other frameworks come with. First, to gather the data I used Javascript libraries to hit the Django API and get all the data available at the endpoint. I then created a simple HTML page which displayed all the necessary data in a simple table. I then created hyperlinks which would link every data table to a page which listed all the sources and the last time they were accessed. This was the primary functionality and scope necessary for the website.



Xbox Status

Check this page for details on the status of Xbox features and functionality.

Status key

✓ Up and running
⚠ Limited
⊘ Major outage

Services

⚠ Games & gaming

✓ Account & profile

OPERATIONAL SERVICES

Sign-in	UP AND RUNNING ●
Signing in to multiple devices	UP AND RUNNING ●
Creating profiles	UP AND RUNNING ●
Updating profiles	UP AND RUNNING ●
Account migration	UP AND RUNNING ●
Account settings	UP AND RUNNING ●
Privacy settings	UP AND RUNNING ●
Gamerpics	UP AND RUNNING ●
Gamertag management	UP AND RUNNING ●
Home Xbox	UP AND RUNNING ●
Avatars	UP AND RUNNING ●
Microsoft Rewards	UP AND RUNNING ●

Figure 1: XBOX Status Page

For styling, I utilized a similar scheme to the example in Figure 1, highlighting rows with recent data green, somewhat recent data yellow, and outdated sources red. This allowed for users to quickly discern visually what sources could be causing data discrepancies without having to read all the information, leading to a quicker and more convenient user experience.

3.3 Challenges

The challenges around this project were primarily learning all the new languages necessary to complete it. I had never had experience with React, the technology used to create the front end. This meant a huge learning curve that made me take longer than expected to finish that portion of the project.

Additionally, I had to learn how to develop production level code at an actual company. The process for writing code for a company is much different than writing code at school. Learning all the processes took a while and made development quite slow at times. However, the mistakes I made at this internship allowed me to become a much better developer, definitely a net positive in the long run.

4. Results

The current project is still undergoing some final checks before being used in production organization wide. Currently, it is being used by data engineers locally to address data outages that cannot be readily diagnosed. Before leaving the internship, I conducted some tests of the efficiency of the solution.

Before the development of my product, tracing through all the data sources was an hours-long process, and tracing through one dashboard took me approximately 4 hours. However, with my website, the process of finding outdated sources took about 5 minutes, drastically reducing the time required to diagnose problems. This was the largest and most effective outcome of this project, and it allows for data engineers within the organization to drastically reduce their already large workload.

5. Conclusion

Overall, the data monitoring project I did over the summer at Flowcode provided tremendous value to the organization. The website gave internal stakeholders a centralized platform to easily check data freshness. The process of identifying improper data was reduced by a significant factor due to the amount of time saved with the clean front end and robust back end. By reducing the time necessary for data engineers to diagnose data outages, my solution simplified their jobs and allowed them to work on more innovative, forward-thinking solutions that would further push the needle. Thus, at a small company that is looking to accelerate its processes, this solution provides tremendous time value, which can truly create a difference in the productivity of all relevant stakeholders.

6. Future Work

Future work that could be done to improve this solution would be related to the speed of the website. In order to render the front end, the application has to ping the API endpoint and process all the data, which takes an excessive amount of time on a row-by-row basis. One solution that could be implemented would be to batch the data, or process multiple rows simultaneously, saving the time of ingesting a single row at a time. This would reduce the processing time significantly.

Additionally, new features could be added to further serve stakeholders, such as automatic flagging of status changes in order to alert data engineers of when data sources may go stale. This would provide tremendous value as it would allow them to solve problems as soon as they occur, possibly preventing future issues that could have severe downstream consequences.

References

[1] XBOX. 2022. Xbox Status. Retrieved November 28, 2022 from https://support.xbox.com/en-US/xbox-live-status.

[2] Fivetran. 2022. Fivetran Status. Retrieved November 28, 20222 from https://status.fivetran.com/