Software Automation: Overview of the Process of Creating Tools to Streamline Business Operations

CS4991 Capstone Report, 2023

Fardeen Khan
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
ffk9uu@virginia.edu

ABSTRACT
A consumer and business credit card company with millions of customers needed to find a more streamlined way to track records of each communication while maintaining customer privacy. Internal access required consulting engineers, who had to spend time querying and troubleshooting data. To streamline this operation, I used Vue.js, Docker, Java Spring Boot, and Amazon Web Services to create an automated tool enabling customer-facing employees to bypass the engineers, saving valuable time. The tool consisted of a "message id" lookup which returned message delivery status and method, etc. Future expansions may include broadening the tool's function to serve as a means for customers to track their messages.

1. INTRODUCTION

When a credit card customer makes a purchase, receives a promotional alert, or completes a payment to their account, a system is triggered to send them a notification confirming this action. However, upon triggering this notification, background subsystems are also initiated to store metadata about the notification process. For example, if the customer completed a purchase and is now receiving the message confirming payment on their mobile application, the metadata includes the following—send timestamp, message language, message id, delivery status, delivery method (mail, push notification, call, etc.), message type, etc.

Prior to development of this tool, clients asked engineers about the status of a message by giving them a message id. Then the engineers would query data using the message id and look up the additional related metadata. This process could take up to one hour a day per developer. To streamline this operation, the tool was created for the clients to bypass the process of reaching out to an engineer, saving valuable time. In broad technical terms, the tool consisted of a "message id" lookup which would return the desired information about a message such as delivery status and method. In the background the Vue.js frontend calls an API querying the necessary data. To serve this in a production environment, these components were deployed onto an AWS ECS instance in a Docker container. Upon deployment, the process to access this metadata is now reduced to searching a message id in the tool, which will return the necessary data.

2. RELATED WORKS

A similar solution was proposed by Gunklach, et al. (2023). When dealing with large raw data sources like those in large enterprise applications, cataloging the data for human use and interpretation becomes tedious, especially for non-technical users. Navigating through the complex datasets

often requires specific queries and in turn extensive knowledge of the architecture. The solution proposed is a three-tier architecture comprised of a logging layer, an extraction layer and a presentation layer. The end user interacts with the presentation layer, which is the front-end of the application. The logging layer is used to log all queries, which are then executed in the extraction layer.

Because an integral part of this tool is the search interface for looking up message information, it functions much like a search engine, using queries to obtain the desired data. However, from Silverstein and Helzinger's (1999) study, search engines such as the Altavista Search Engine or Google may need to perform additional computation to query data that may have been implied but was not explicitly defined by the user. This is one aspect where the self-service tool I worked on differs from a search engine, as the inputs for the tool must match an existing message id exactly, and no intention by the user can be inferred.

3. PROJECT DESIGN

The self-service tool was a part of a larger suite of tools within the bank's messaging services. As such, prior to the start of the project, there needed to be planning in order to ensure the tool would properly use the existing services used by the other tools.

3.1 Client Specification

A key component of this tool is its user interface, as it is the main form of interaction a user will have with the service. As a result, it was imperative the design of the interface complied with the "User Interface Guidelines" set for all the company's software to maintain a streamlined experience for all users. My development team was assigned a design team to collaborate with on the user interface and ensure these guidelines were followed with the app. This meant certain icons, error messages, fonts, symbols, etc. were to be used when designing it.

The design team also specified the necessary information to be displayed on the page when returning a result: the send timestamp, message language, message id, delivery status, delivery method (mail, push notification, call, etc.), message type. These specific data fields were chosen because they were deemed the minimal necessary fields but would still provide ample information to debug a scenario while keeping the visual state clean.

3.2 Overview of Design

Implementing a tool to track message statuses requires a three-tiered approach like most apps: a front-end page developed using Vue.js framework, a backend logic controller configured using Java Spring Boot, and the data layer hosted on Amazon's AWS DynamoDB service.

3.2.1 Front-end

Due to the structure of our summer internship, the front-end component was the first step towards creating the app. The goal for the user interface was to have a search bar upon the page loading for the first time. The search bar only accepts a valid message status ID, which will then be used to look up a matching entry in the database and return its associated data fields. The fields are then displayed in a 3x2 grid. If the input is invalid an error message will be shown to the user saying an 18-digit message ID must be entered. There will also be errors communicating application issues, such as HTTP 1xx, 3xx, 4xx, and 5xx statuses. The exact message was drafted with the design team to adhere to the company's UX guidelines.

3.2.2 Back-end

The majority of this app's functionality was spent on the back-end services, using Java

Spring Boot. During the process of sending a message to the users, the message is sent to a message vendor, such as Apple or Google for push notifications or SparkPost for other forms of communication. These vendors return status codes showing the status of the message. Additionally, as the message is sent, the messaging system also stores data of its own, such as timestamps, message types, message size, transfer rate, language codes, etc. The API would be used to access this data from the database upon request from the front end.

Specifically, the main function to be implemented is the one retrieving the necessary data fields. The format of the returned data is required to be a JSON format map, where the fields, send timestamp, message language, message id, delivery status, delivery method (mail, push notification, call, etc.), message type, would map to their values. This JSON payload would then be sent to the Vue app to be parsed into the HTML page.

3.2.3 Architecture

The reasoning for this design is mainly for the purpose of resiliency and speed. Using Java allows for a broad range of contributors and Spring Boot is available as a very scalable framework, allowing for over 66 million daily messages to be sent while avoiding overloading the infrastructure. Vue.js is a modern front-end framework that can combine the advantages of both Angular.js and React.js allowing for simplicity while developing an app.

Throughout the development pipeline, there are three stages used to deploy versions of the application to. Deployments must move from development environments to QA environments to the production environment, allowing each to test the state of the app as new features are added. Jenkins CI is used to automate this process, along with the company's internal managed pipeline, which reads a YAML file containing configurations for the infrastructure using AWS and Docker. Specifically, the YAML file contains the necessary IAM roles for the app's AWS services to access the data. It also contains information for starting up the ECS Fargate service and initializing a Docker container with it using the provided Dockerfile. Once the ECS stage is complete, the app is deployed into multiple regions for resiliency, along with traffic routing for load balancing. As a result, a sudden increase in the traffic would not immediately exhaust one region, but would route traffic to another to ease the load on it. Also, failover capabilities are implemented to allow for the resilience, so the app does not lose its functionality in case there is a failure on one of the regions.

4. RESULTS

The original purpose of this tool was to alleviate engineer stress while allowing clients to easily view and debug problems related to sent messages. Upon deployment of this application into the production environment, it is available to all internal employees with a valid message id, allowing them to quickly see the status of their desired messages.

Prior to this app, engineers would spend on average one hour a day searching through a database to present the clients with potential solutions to their issues. Because the app is a self-service tool, it can operate autonomously, relieving engineers of this duty while also presenting the clients with immediate results, saving time for both parties.

5. CONCLUSION

The culmination of the internship was marked by the deployment of this tool to the company's live servers for employees to use on a daily basis. When visiting the site, users

will be able to enter a given message tracking ID and see results for it within seconds. They will be presented with information pertaining to delivery statuses, dates and times relevant to the transportation of the message, non-personal identifiable information about the contents of the message, such as language and method of communication, and type of message. In the case of an error, they will also be given an error code and a team responsible for troubleshooting. This will save valuable time for both the company's customer-facing employees and engineers.

6. FUTURE WORK

Currently, the status tool has been functional as the same release candidate at the end of my internship. However, there has been new proposed additions to the tool to allow it to provide information of not only account-level but also email-level messages, which has been prioritized as the next step of the lifecycle of the tool. Further potential additions of the app can be searching multiple messages at once, for example, looking up statuses of all push notifications sent to customers containing promotional information within the last three days, as this would remove the need to do the same tasks such as repeatedly searching related messages.

REFERENCES

Gunklach, J., Michalczyk, S., Nadj, M. Metadata Extraction from User Queries for Self-Service Data Lake Exploration. *Datenbank Spektrum* **23**, 97–105 (2023). https://doi.org/10.1007/s13222-023-00448-z

Silverstein, C., Marais, H., Henzinger, M., & Moricz, M. (1999). (rep.). *Analysis of a very large web search engine query log*.