A SaaS Solution For Streamlining Webhook Integration

Abhigyan Niranjan [1], Aditya Gangwar [1] and Harsh Soni [1]

[1] *Department of Computer Science and Engineering, Indian Institute of Information Technology Dharwad, 580009, India*

ARTICLE INFO

Keywords:

Webhooks, API, Automation,

Developer tools, Data Logging

ABSTRACT

Webhooks are a popular way on the web to send and receive events, they allow seamless cross application communication with minimal setup. In this report, we investigate and present a solution to the problems encountered when integrating webhook based communication into an application. As a solution, we present and analyze a man-in-the-middle service which will receive and forward all webhook data. The receiving application(s) can then receive the forwarded webhook or request the data from this service through WebSockets or Server Sent Events (SSE). We analyze the benefits and limitations of this approach and evaluate potential optimizations.

1. Introduction

Our solution, Webhooked, is an all-in-one platform for seamless webhook integration and efficient data monitoring. It's designed with developers in mind. Webhooked empowers you to effortlessly harness the power of webhooks, streamlining your workflow and enhancing your productivity. With our platform, developers can easily set up, manage, and monitor webhooks endpoints, ensuring reliable data delivery and real-time updates. Whether they're building dynamic applications, automating processes, or enhancing your system's capabilities, Webhooked provides the tools one needs to succeed. It simplifies webhook management and simplifies the integration process.

The prime contributions of this work are:

- Development of a robust and performant service to receive, process and forward webhook data.
- Development of an intuitive user interface for seamless webhook endpoint setup and management.
- Implementation of robust monitoring and alerting mechanisms to ensure reliable data delivery and real-time updates.
- Documentation and support resources to assist developers in leveraging the full potential of Webhooked.

2. Problem Overview

Before jumping to the problem, we will first like to give an outline of how communication using webhooks takes place. In this model, primarily two entities are involved, the webhook sender and the webhook receiver. The receiver provides the sender with a URL beforehand, let us call this the receiving URL. The receiver has access to this URL and can read any requests made to this URL. Whenever a sender wants to send a webhook to a certain receiver, the sender will send a POST request to the receiving URL of that receiver. The receiver can now read the request data and perform some action internally in response to that.

There is one flaw in this flow, which is that the receiver needs to create a web server and listen to requests on a URL. This creates a number of problems:

- Any downtime of the application will lead to permanent loss of webhooks received during that duration.

`22bcs001@iiitdwd.ac.in` (.A.N.); `22bcs003@iiitdwd.ac.in` (.A.G.); `22bcs046@iiitdwd.ac.in` (.H.S.)
ORCID(s):

- It can be hard to test locally as it is not practical to point a domain to your local address.
- Creating a webserver adds unwanted additional complexity on the receiver's end.

3. Proposed Solution

As a solution to this problem, we have built a man-in-the-middle service, which will receive all of the webhook data of the user, and forward it to the user through whatever mode the user wants to receive it. Additionally our service will provide logging, monitoring, fan-in, fan-out, and provisions for resending missed data to ensure the user application can consume all webhooks in a very straightforward manner.

The following figure shows the application architecture:

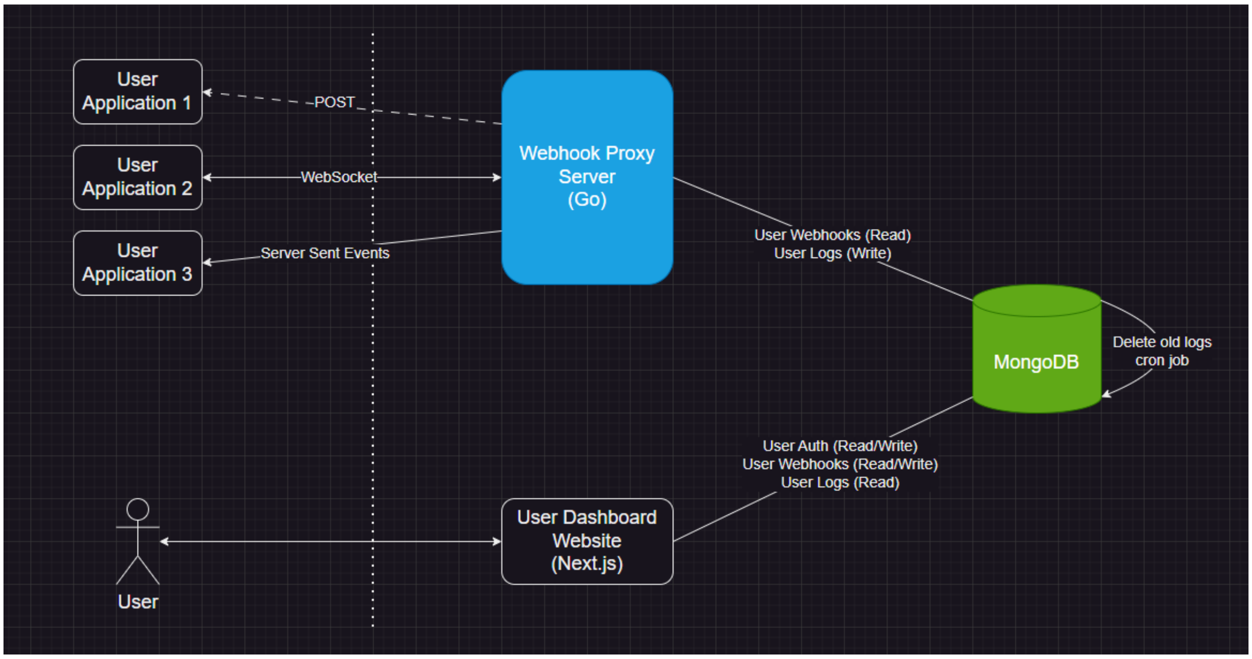

Figure 1: Application Architecture

We will now describe each component in detail.

- **Database:**
  MongoDB was used for the database as our team had past experience using MongoDB. In a single database inside the cluster, there are three collections:
  - **users:**
    This collection stores user info which is provided by GitHub OAuth
  - **webhookInfo:**
    This collection stores data of each webhook endpoint created by a user
  - **webhookLogEntry:**
    This collection stores all log data there is. It is a clustered collection and is clustered by the log's insertion timestamp. This helps drastically improve performance when querying logs. The collection is configured to delete logs older than 7 days. This keeps the size manageable.
- **Webhook Proxy Server:**
  This part of the application is responsible for receiving webhook data for the user, and generating the logs in the

database. It also forwards the received webhooks to the user (if the user has configured it to do so), while also relaying the messages through active WebSocket or SSE connections. It authenticates client applications trying to connect over WebSocket or SSE through a pre-generated API key which is given to the client. The application maintains a watch stream to the database which allows it to dynamically create, delete or update webhooks as the change without the need to restart. It was written in Go due to the nature of the problem it solves. A proxy service should be highly concurrent and go provides an excellent concurrency model while also being fast.

- **User Dashboard Website:**
  This is the user facing component of the application. The user can authenticate using their GitHub account, and then proceed to the dashboard where they can view past logs, configure existing webhook endpoints, and view their account information such as their API key.

4. Analysis

In this section, we analyze benefits and limitations of the application. Starting with the benefits, the most obvious benefit is that users no longer need a public URL to receive webhooks. A user's application can recieve webhook data over a websocket connection or an SSE stream. This makes it possible to consume webhooks in scenarios in which provisioning a public URL is not possible, an example being programs being locally executed on a machine with a dynamic IP address. The second benefit is that applications can now deal with dropped webhook data in the case of downtime. Since logs of recent webhook data is maintained, an application which faces downtime can request past logs of the duration it missed to be replayed. This increases the fault tolerance of applications which utilize the service. The third benefit is that users can now view the webhook traffic their applications receive without having to set up additional monitoring and logging in their code. This also gives them the ability to perform analysis as per their requirements.

The application has a few limitations, which mostly revolve around scaling. We have not yet figured out a way to shard the webhook proxy server into multiple servers to distribute load. Therefore currently, only vertical scaling of the webhook proxy server is possible. The website however can be run serverlessly ensuring high performance. This is a challenge because to forward webhooks and to authenticate client applications, the proxy service needs to read data from the database, and since a client application could listen to multiple webhooks from the same user, it must be ensured that all the webhook data of a user flows through the same shard. We can think of each user being assigned a shard. This is a hard problem to solve as some users will receieve less traffic and some users will receive a lot of traffic, so if sharding is to be made possible, then an algorithm must be devised which can distribute users to shards such that each shard is approximately under the same load. This algorithm must also be able to respond to changes in traffic. A possible solution could be to assign users to shards on the basis of traffic permitted, instead of actual traffic. Meaning that users could sign up for a tier, with a higher tier costing more but allowing higher throughput. Shards could then be designated to only handle users of a specific tier. There will be more users per shard in the lower tiers compared to higher tiers, however this approach can lead to wasteful use of resources if the higher tier users are not utilizing their limits to their full extent.

5. Conclusions

In conclusion, the proposed solution, Webhooked, addresses the challenges of integrating webhook-based communication into applications by offering a robust and efficient platform for seamless webhook integration and data monitoring. By introducing a man-in-the-middle service, Webhooked eliminates the need for application developers to maintain a web server for webhook reception, thereby mitigating the risks associated with downtime and simplifying local testing.
While Webhooked offers significant benefits, there are limitations related to scaling, particularly regarding the webhook proxy server. The current architecture allows for vertical scaling, but challenges remain in implementing horizontal scaling. However, potential solutions, such as user-tier-based sharding, offer promising avenues for addressing these scaling challenges.
Overall, Webhooked simplifies webhook management, enhances fault tolerance, and offers a reliable solution for integrating webhook-based communication into applications, making it a valuable tool for developers seeking to streamline their workflow and enhance productivity.

Conflicts of Interest

The authors report no conflicts of interest.

Competing Interest

The authors report no competing financial interests

References