Webhooks-as-a-Service: A Custom API Design

Major Qualifying Project 2022-2023

Authored by:

Thomas (Cole) Varney

Vishnu Priya Dendukuri

John Higgins

Darian Tavana

Presented to:

Professor Joshua Cuneo

Sponsored by:

Jeremy Duvall, 7Factor Founder

Authorship Page

This report was authored and edited by Thomas (Cole) Varney, Vishnu Priya, John Higgins, and Darian Tavana. Each person contributed equally to the report. Professor Joshua Cuneo provided periodic feedback.

Acknowledgments

The Webhooks MQP team would like to thank the following people for their contributions to this project:

- Professor Joshua Cuneo for advising this project.
- Jeremy Duvall, Lindsay Duvall, and the entirety of 7Factor for sponsoring this project.

Abstract

A key problem for software workflows is automating tasks between unique services. By connecting unique services, an organization can greatly increase its efficiency. This MQP aims to address this issue by researching and developing a Webhooks-as-a-Service (WaaS) platform. To accomplish this, a software design process was conducted alongside our sponsor 7Factor. Through this process, a back-end solution was exclusively built and deployed using Amazon Web Services (AWS) and a front-end solution was set up.

Table Of Contents

Introduction

Our Sponsor

Our sponsor 7Factor is a software contracting company based in Dunwoody, Georgia. They create secure and scalable software solutions for their clients. The 7Factor team creates solutions through the following iterative lifecycle: consulting with the client, developing the desired product, and offering support and maintenance after deployment (7Factor, 2022).

Purpose

We developed a tool which allows 7Factor employees to deliver automated webhook messages from a single producer to several consumers. A Webhook is an HTTP callback function which offers well-structured, automated, and efficient communication between different web applications in real time (Atlassian, 2020). There are existing products such as Zappier and SyncPenguin, but they are rigid and only support a finite number of applications. Additionally, these providers pricing model is inflexible, causing users to pay a substantial overhead for unused resources (Sync Penguin, n.d.; Zapier, n.d.). Hence, 7Factor requested a webhook management tool that can automate their workflows. To meet these needs, the team developed a Webhook-as-a-Service (WaaS) platform which allows the user to manage their personal webhook configurations in a user-driven webhook community. Within the envisioned community, users will be able to create new webhook payload schemas and reuse other payload schemas that have been previously defined. With this approach, the WPI team did not have to build out a large suite of webhook schemas and instead could focus on the back-end architecture and front-end features.

Background

Event-Driven Architectures

Event-driven architecture involves writing programs and applications that use events to communicate and trigger logic between services. An event in this context is a change of state or update on a system. Event-driven architecture typically consists of three main subsystems: the event producer, event router, and event consumers. The producer(s) send(s) events to the event router, which filters the events and sends the results to the consumer(s). The power in this solution is that each part of the system is completely decoupled (Amazon Web Services Inc., 2022). A few examples of events would be a user submitting data via a form, handling a mouse-click or IO event, or a change of state on a machine. From a system perspective, an event may look like a query to the database, a program error, a new addition to the program logs, or even a notification of a program outage. Through the decoupling of services, event-driven architecture brings a few inherent benefits. The first benefit is that they are easy to scale and are more robust since the router and the services work independently. This means that if one service fails, the rest will continue to function since the system is interoperable (Amazon Web Services Inc., 2022). Also, to connect services, developers no longer must write custom code for polling, filtering, and routing events as they normally would. Including a router also removes the need for coordination between producer and consumer services, which in turn speeds up the system development (Amazon Web Services Inc., 2022).

Push Technologies Architectures

The two key mechanisms clients and servers use to interact with events are push and pull (Thomson et al., 2016). An example of a web pull would be a client requesting data from the server via a client facilitated event. An example of a web push would be a client receiving data from the server via a server facilitated event. Our project focuses on Webhooks, a web push implementation. We will explore implementations of various push/pull systems and discuss their tradeoffs which we will leverage to discuss the importance of webhooks.

Long Polling

One common way to realize events is by long polling. Long polling is initiated by a client opening a connection with a server. Once this connection is open, the server can send information to the client. This system exists somewhere in between pull and push technologies, as the client initiates the connection by sending a request, but the server then holds the request until there is an update, and pushes information to the client (Gautam, n.d.). This ensures the user gets all the updated data without performing any action. This method allows server-side caching of general information which helps temporarily store data on the server side so that it can be used (Gautam, n.d.). The disadvantage of this system is that multiple client connections need to be maintained simultaneously, and each connection needs to be left open so that clients can receive the latest information when it is available. This creates the problem that servers will likely have multiple connections open at a time and therefore need to be configured to efficiently use resources (Kilbride-Singh, 2022). Due to these issues, additional latency is introduced (Biehl, 25). Current options such as WebSocket would be more efficient as it does not use the full HTTP request and response content and is not as resource intensive as long polling (Abuhakmeh, 2022).

WebSocket

A WebSocket is a two-way protocol used to communicate in real time between a client and the server. It uses Transmission Control Protocol (TCP) socket to maintain the connection (Pautov, 2021). In this model, to open a connection, both the client and the server both make the handshake after the communication is initiated, if they want to form a connection. Once the connection is established, the data exchange can happen bidirectionally until the connection is terminated (*What Is Web Socket and How It Is Different from the HTTP?*, 2022). The connection would be terminated if either the client or the server decides to break the connection or if either of them shuts down. Real-time applications such as chat applications and gaming applications often use WebSockets as the continuous connection between the server and client. These applications do this to maintain real-time bidirectional transfer to transfer data without making a multitude of requests (*What Is Web Socket and How It Is Different from the HTTP?*, 2022). This would also replace the use of long polling, as it handles the latency issues associated with it. Still, there is a certain amount of demand on the server for keeping these communication channels open, but this is much less when compared to polling (Pautov, 2021). Overall, WebSockets reduce the size of the HTTP payload, as it doesn’t use XMLHttpRequest, which sends headers on every request (Kilbride-Singh, 2022). However, WebSockets requires a full HTML5 supporting browser (Pautov, 2021). Therefore, it is good to use WebSockets when dealing with continuous bidirectional updates.

Webhooks

Another implementation of an event-driven workflow is a webhook based system. It is more frequently used for one-way communication between two servers (Molina, 2022). Webhooks, less frequently referred to as HTTP push APIs or web callbacks, are an event-based communication system over the web (Atlassian, 2020). There are two parts to a webhook system: the webhook provider and the webhook consumer. The function of the provider is to send an HTTP payload when triggered by an event. The payload contains information about the event that occurred. The webhook consumer is then responsible for consuming the data from the producer and performing an action using that information. A system built on webhooks brings multiple benefits in comparison to a solution such as polling. As mentioned above, long polling takes more resources than required in general. According to Zapier, about 99.5% of polling requests don't have an update (*Webhooks vs API Polling*, n.d.). With the use of webhooks, the number of requests sent per second can be reduced dramatically. As the updates happen, the server sends a webhook to a consumer, but in polling there is likely a delay in the update, as the request must be propagated to all the open connections (*Webhooks vs API Polling*, n.d.). This prevents polling from achieving real-time event realization, making it the least economical choice. Webhook systems contain a built-in event data structure, the event payload, for easy information sharing between the server and the client. This means that through webhook architecture, a webhook payload can be the vessel for communicating an event from the server to the client without developers needing to build event sending and receiving logic.

Webhooks use unidirectional communication, meaning they cannot be used for complicated communication that requires multi-direction communication (Sarabyn, n.d.). Additionally, if an event occurs and the other system is down, the receiver would not be able to accept the webhook. In this case, it becomes unclear if the webhook failed to send or if the consumer system is not functioning (Sarabyn, n.d.).

The main benefit of webhooks is that they trigger automatically and immediately. The server handles all logic involved with notification detection and is responsible for finding and notifying the client when events occur. The client does not have to do any work to realize events; it simply must subscribe to a webhook endpoint and wait for the server to send it an event notification. Developers now are solely responsible for building logic for receiving webhook payloads as opposed to the entire pull/push flow needed for a polling system. This improves the developer/user experience associated with the service. Therefore, webhooks work best when used to create notification systems to track changes or to update specific information.

Applications for Webhooks in Workplace Automation

One of the biggest advantages of webhooks for workplace automation is the ability to efficiently integrate unique platforms that otherwise would have no connection to each other. Webhooks allow for instantaneous notification of events to any platform that can be set up to receive the webhook producer's payloads.

Webhooks have many applications in the workplace. Some common applications include employee notification of events, for example, shared document changes or back-end system errors. Events like these can trigger a webhook being sent to a destination like a messaging channel (e.g., Discord, Slack, etc.) where any subscriber to a given webhook producer will be notified instantly. This removes the need for employees to constantly monitor their documents, systems, records. Instead, employees can configure webhooks to notify them of changes like these.

Webhooks are also used to trigger more events, creating multi-step tasks, also known as workflows, which has grand implications in the workplace. Workflows are a set of webhooks/events triggered in series that perform automated tasks for users without the need for manual input. An example of this is automating a build process for an application. Today, such a task can be done trivially using GitHub Actions, which is a system which allows developers to set up a series of steps to perform an action (GitHub, n.d.). However, before GitHub Actions, to perform such automation, a developer would use GitHub's webhooks. By providing such a system, one could leverage a testing framework alongside a containerization application such as Docker to create a new build of an application each time an action is done to a GitHub repository (Jaramillo et al., 2016).

Using webhooks to create workflows increases automation potential. Though developing workflows does take time, effort, and planning, the configuration time is significantly less than manually performing the desired tasks. Additionally, after the initial configuration, the workflow could be reused, shared, or even offered as a paid service.

Multiple Destination Webhook

One problem with the current state of webhooks is that they rely on a 1:1 producer to consumer relationship. This means that for each destination, the user must configure a relationship with the producer webhook.

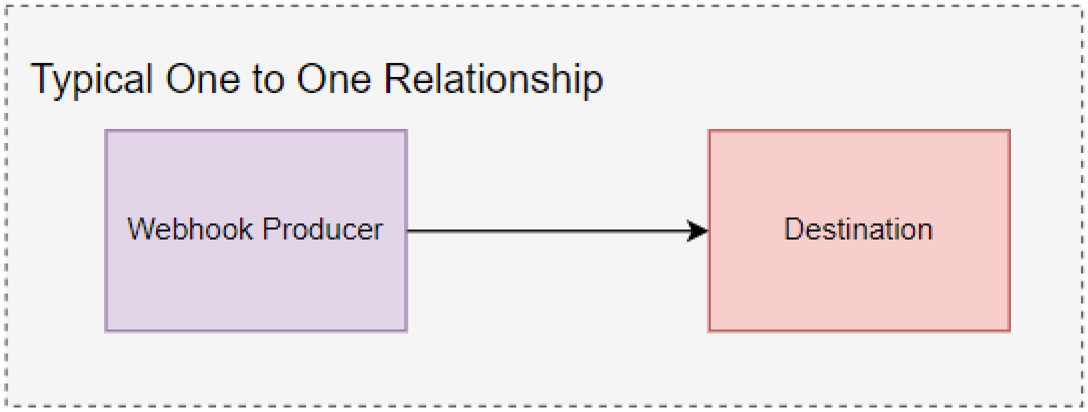


Figure 1: A typical One to One relationship between a webhook producer and destination

Webhooks supporting one-to-many subscription relationships would have a few benefits in workplace automations. Firstly, it would decrease the number of configured webhook relationships. Secondly, it would decrease configuration time and space by requiring only one configuration for a multiple destination webhook instead of creating a webhook configuration for each destination. This one-to-many relationship also centralizes the logic for each webhook provider. The provider must only be configured once and will only need to forward an event payload a single time. This will decrease the necessary use of resources needed for the webhook provider to send the payload, simplifying the whole integration. Lastly, if it is determined that a webhook should be sent to an additional destination, one can configure it to subscribe to the existing webhook configuration instead of reconfiguring an entirely new webhook.

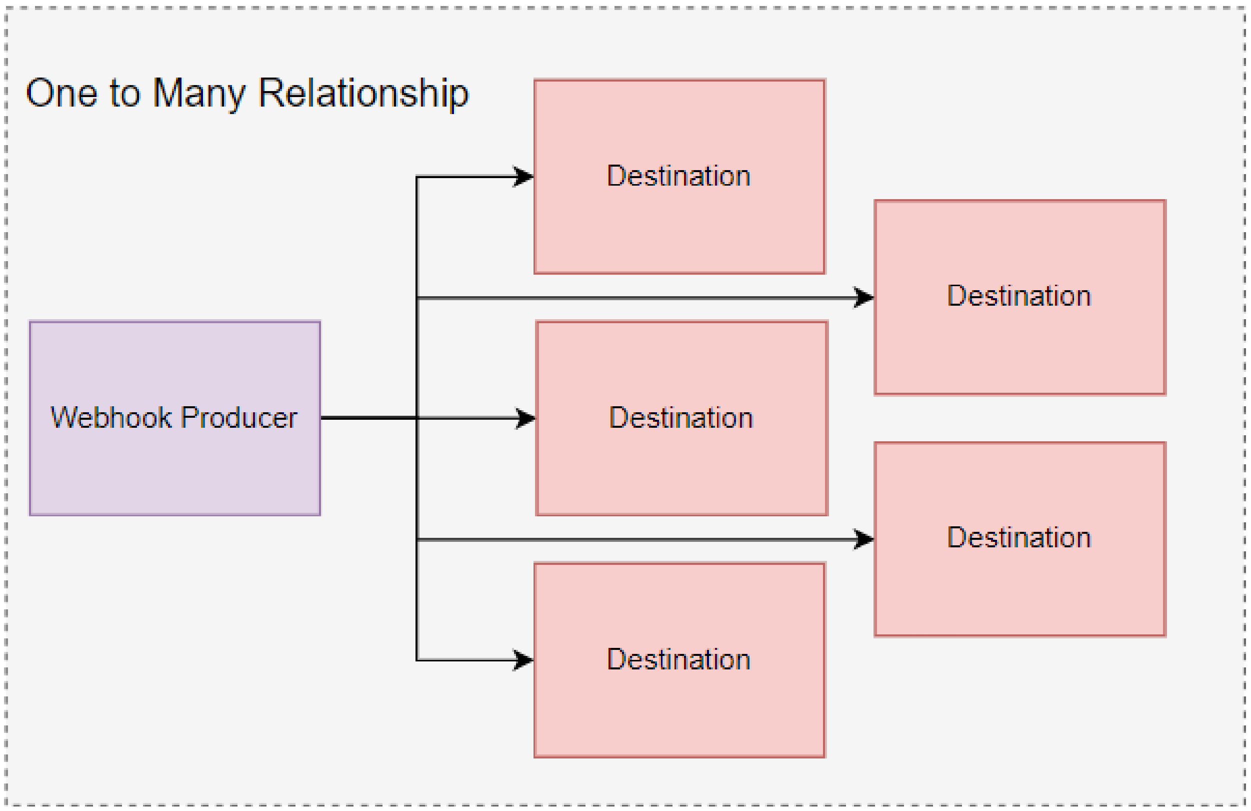


Figure 2: A typical one-to-many relationship between a webhook producer and destination

The main issue with one-to-many relationships for a webhook is the fact that webhooks are a form of pub/sub (publisher and subscriber), meaning that one system publishes events and the other system(s) will listen for them. In the case of webhooks, the publisher needs to push the webhook payload directly to the destination URL. This ultimately means that each webhook can only be sent to one destination. A potential solution to this problem is to use a middleware service to allow a webhook producer to send event payloads to a single endpoint that is then responsible for fanning out the payload to multiple destinations.

One concept that complicates this system is that webhook destinations may be expecting a JSON (JavaScript Object Notation) object with a specific format and will error if the format is not adhered to. A potential solution to this problem is to instruct the middleware to perform data transformation before sending the webhook to each destination. Before sending the webhook payload to each destination, the data will be transformed to adhere to format standards set by the destination's webhook integration APIs. Each subscriber may have different specifications for what a valid payload looks like. In this case, for the system to properly receive the data, a transformation is necessary.

Methodology

At the start of the project, we spent some time researching webhooks, their current state of development, and their current use in industry. This preliminary time was also spent researching possible technologies that could be used to develop a product that tackles the provided problem space.

Previous Work

As mentioned earlier, in the 2020-2021 academic year there was a previous group that worked on this same project, leaving behind their project code and MQP report. Our MQP team came into the project with the notion that we would be continuing from where the previous group left off. After reviewing the materials provided, it was decided that though there was a lot to learn from the previous group’s progress, it made more sense to move in a different direction in developing this Webhook SaaS.

The previous MQP report made it easy to understand the problem space, as well as learn about application specific information on webhooks without having to scour the internet for research. This report also included their platform design, which made it easy to understand how they developed their application.

The previous students on this MQP had built their solution within a containerized Node.JS application run through Docker and Docker compose. They had investigated more permanent hosting options and found that locally hosted Docker images provided the most economically friendly option given the circumstances. For their database, they used SQLite due to its simplicity and ease of integration with their containerized back-end. Their front-end was planned to be built using React, since most of their team members had previous experience.

We felt that the architecture chosen by the previous group retained validity as was, though we saw potential issues with efficiency and cost down the line if the system was required to run under constant load and at scale. It was because of this that we decided to look at the previous group’s MQP as a proof-of-concept and determine a way to build a robust system that could withstand real world use.

Agile Workflow

Our team used the Agile methodology for this project. Agile project management is an iterative approach to create software projects by dividing the whole process into smaller cycles called sprints (*Agile Project Management - The Beginners Guide*, n.d.). The project was completed over 2 terms (14 weeks long). Our team regularly attended standup meetings to discuss progress, setbacks, and tasks to do before the next meeting to make sure we are on track and can seek help if needed.

A-term

During the first three weeks, we looked over the previous team’s project and met with the sponsor Jeremy Duvall. Our team decided not to continue the previous group's project and instead researched different technologies that could be used. Once we formed a clear idea of the project, we made a proposal with a basic architectural diagram and technologies.

After we got an approval from the sponsor, we divided the last four weeks into two 2-week sprints. In the first sprint, we set up API Gateway and DynamoDB and developed a simple middleware to forward webhook payloads to a destination (Child Lambda). During the second sprint we focused on the development of Router Lambda which fans out requests to multiple Child Lambdas. Immediately after, our team connected the Router Lambda to the API Gateway and the Child Lambda. API Gateway was also connected to the database for the registration flow and configured on the Lambda instances. This process also included manual tests to check if the system functioned properly.

B-term

During this term, we decided to have 1-week sprints where we divided the focus to be on both the paper and the project. We set up a plan for the project and had the paper outline ready by the end of the first sprint.

The second sprint included the completion of the Abstract and Introduction draft, and we investigated Remix for the front-end and started mockups for the UI. For the back-end, we finalized the Lambda functions.

During the third sprint, we wrote the background and altered written drafts based on peer reviews. In addition to this, we defined the required front-end API methods, updated the Swagger documentation for the new API endpoints, and finalized the UI mockups.

For the fourth and fifth sprint, the team revised existing drafts based on peer reviews and finished the methodology draft. For the front-end part of the project, we researched and created reusable React components and a custom JSON schema. For the back-end, we implemented the API methods needed for the front-end. The configurations were also tied to a logged-in user.

During the sixth week, we finished the final draft and started editing based on peer review. The team created the final front-end UI to customize the JSON Schema and started creating CRUD functionality. The transformations were also implemented.

Finally for the last sprint, we completed the project report with small, required additions based on the review from Professor Cuneo. The CRUD implementations were completed, the transformations were finalized, and a back-end architectural diagram was created.

JIRA

Jira is a management tool that helps software teams organize and keep track of their work in agile development (Atlassian, n.d.). This tool provides boards where the team can manage the tasks by arranging them based on their status. Each task included the assignee, sprint, description, etc. The team used this tool to assign different tickets for different tasks related to the front-end and back-end. The different statuses include “to do”, “in progress”, “code review”, and “done”. One can move the ticket based on the status through which the other teammates know where they are. The transparency in the workflow helped the team to keep track of the work done and pending work.

Version Control

GitHub

Our team chose GitHub for our version control system of choice. Other version control systems (VCS) we considered were platforms like Atlassian Bitbucket which are often overlooked. For this MQP though, GitHub was the best choice for a couple of reasons. The first being that it is a free service, at least up to a point that would not cost anything within the scope of this MQP. Secondly, each of our team members has had much more experience using GitHub than any of the other version control services.

One way that we used GitHub was to store all our source code for the project. First, we created a GitHub organization, and all the team members were invited to become members. This allowed each team member to see the source code under the organization as well as contribute to and review it. Our source code within the organization was separated into project repositories, only accessible by members of our MQP organization. This included our front-end repositories, our router and child lambda repositories, and our authorizer repositories.

Another great benefit of using GitHub for this project was that it provided a user-friendly way for us to monitor each other's progress and review code that different team members had written. Using webhook integration, we set up GitHub to notify the team's Discord of any changes or activity within a repository for the GitHub organization. This made it easy to see what others were doing between our standup meetings throughout the week without having to go looking in each repository. Lastly, before any code is pushed to the main branch, a pull request is made, which displays the changes made from the previous version to the proposed latest version, and each team member is responsible for reviewing and approving the changes before they are committed.

GitHub Actions

GitHub Actions (GA) is a tool provided by GitHub to automate workflows that are triggered by events such as a commit, issue creation or the creation of a new release (GitHub, n.d.). Typically, GA automates a repetitive task to save time and increase reliability. A few members of our team had experience working with GA and set up an action that built and deployed our code to an AWS Lambda function whenever a developer pushes code to the main branch. We did this to ensure that all our cloud assets were up to date with our development progress.

Researching and Choosing Technologies

Before we set out to build our system, some time was spent looking into possible technologies to use in our implementation. Before development, we first had to explore different architecture options. We also had to determine the technologies that we would use to build each possible design solution. Finally, we set out to determine what architecture and accompanying technologies will best fit our needs. Overall, we looked to find a solution that scales well, is easy to develop rapidly, would be easy to adapt and expand on in the future and is low-cost to develop as an MVP.

Since functionally there would be a lot of communication between parts of the system, we felt that we should use a pub/sub framework or a messaging framework to send information between systems, and instead of using a monolithic architecture, we opted to build it out in distributed systems. We liked the idea of having a microservice architecture or something similar because it brings modularity to functionality. This would make debugging easier and reduce the complexity of the codebase. For the messaging portion we explored options like RabbitMQ and Apache Kafka, or simply building our own way to send messages without using a framework. Ultimately, we determined that using AWS Lambda and having Lambda functions trigger other Lambdas would suffice for our system communication needs.

For persistent storage, we could have gone many directions when building our service. We could have used an embedded and lightweight database like SQLite, or we could have gone with some heavier duty databases like PostgreSQL or MongoDB. We also considered whether we wanted to use a SQL or NoSQL database, since each has their own benefits. Pretty early on, we decided to use a NoSQL database like MongoDB or AWS DynamoDB, since our data structure is highly variable, and having flexible data definitions would make storage significantly easier and less confusing to implement.

Another big question that we had was what language and framework would work best for our use case. Ultimately, this came down to what language most of the team has had previous experience with or enough knowledge that the learning curve would not impede progress while still being fun and applicable to this MQP. The languages that ended up being considered were JavaScript, Java, or Python. Since most of the team has experience with JavaScript but a few didn't like it, we settled on using TypeScript since it largely resembles JavaScript with some added benefits like its type of system. This also made development more interesting because we were able to gain some experience with a new language.

Back-End

After a bit of deliberation, we decided that it made the most sense to build out our back-end infrastructure using NodeJS and TypeScript. This conclusion was made since most of the team had previous exposure to NodeJS, and TypeScript has added value when compared with JavaScript. To host our back-end infrastructure, we decided to use Amazon Web Services (AWS). There were a few factors in this decision that will be discussed in the following sections.

Amazon Web Services (AWS)

Amazon Web Services is a host of online services provided by Amazon that encompasses many corners of cloud computing. Some of the service categories they provide include analytics, cloud financial management, containers, databases, IoT, machine learning, security, identity and compliance, storage, serverless computing, and many more (*Cloud Computing Services - Amazon Web Services (AWS)*, n.d.). The team was very attracted to this platform because of its great reputation and centralization of so many different easily integrated tools. We were also interested in the fact that most of the services AWS provides have a free tier that should remain free within the scope of the MQP. In general, using AWS services would allow for easy scaling to high throughput, if that ever became a necessity in the future after we finish our MQP. We decided that in building our Webhooks-as-a-Service platform, the best direction to take was entirely through cloud computing, and more specifically using AWS. AWS allows us to have all our cloud functionality be through one provider whose services are incredibly well-integrated, making development more rapid and the final product more robust, maintainable, and able to operate at scale in the future.

AWS Lambda

AWS Lambda is a remarkably interesting serverless computing solution that AWS provides. Lambda is a compute service that lets you run code without having to provision or manage any servers. All one must do to use lambda is provide code in one of the various languages that lambda supports. Lambda within AWS is organized by functions, each with their own trigger. Triggering a lambda function can be done through using the AWS API gateway or through other AWS services. Each Lambda allows easy scalable cloud computing that would do well for low to exceedingly high throughput, providing scalability and availability at any amount of usage.

AWS DynamoDB

For our persistent storage, we decided to use AWS DynamoDB. "Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale" (*Cloud Computing Services - Amazon Web Services (AWS)*, n.d.), as declared on the DynamoDB main website. DynamoDB has many qualities and features that drew our attention when deciding what technologies to use. The first was its consistent performance speed. Amazon claims that DynamoDB consistently stays in the single millisecond response range at any throughput. They even claim that this is mostly independent from location, with the ability to automatically configure multi-region replication. Amazon also provides top of the line security for our information, meaning that in moving forward, there would be no need for us or future groups to migrate functionality to a more secure database. Lastly, like other AWS services, DynamoDB is effortlessly scalable to meet our needs at any point in the project and seamlessly connects with the other AWS services that we used to build our WaaS.

AWS API Gateway

Amazon API Gateway is a tool that Amazon provides to help connect applications to different facets of cloud computing through building and maintaining Application Programming Interfaces (APIs) at scale. API gateway provides "traffic management, CORS (cross origin resource sharing) support, authorization and access control, throttling, monitoring, and API version management"[4]. API Gateway supports building mainly RESTful APIs and

WEBSOCKET APIs. The way that API Gateway works is that it sits between applications and systems communicating within the cloud and the actual cloud infrastructure to ensure proper communication and security within the system.

Front-End

To let users interact with our back-end system, we set out to build a user interface or front-end. Our front-end application is a proof of concept that showcases the functionality our back-end service provides.

Front-end Framework: React and Typescript

Since our back-end code was written using NodeJS and TypeScript, we also wanted to write our front-end using similar technologies. The front-end development space is ever changing, and new web frameworks are constantly emerging. To choose a framework, we weighed two factors: team experience and popularity. On our team a few of our members have used React before. To find overall popularity we researched a variety of frameworks. One organization we found called TheStateOfJS performs a yearly review of JavaScript technologies, capturing various statistics from thousands of survey responses (Idera Inc. & Frontend Masters, 2020; Shawn Wang & Mark Erikson, 2021). Based off the survey's results from the past 2 years, React was the most popular framework by a significant margin (Idera Inc. & Frontend Masters, 2020; Shawn Wang & Mark Erikson, 2021). This led us to choose React and Typescript to develop our front-end.

Package Manager: NodeJS Package Manager (npm)

NodeJS package manager or npm is the world's largest software registry for NodeJS (npm, n.d.). By using npm, developers can easily install packages called dependencies to their projects. Npm is the defacto approach across numerous software projects and is the only package manager our team was familiar with. With this we chose npm as our package manager.

Linter: ESLint and husky

Typescript already provides greater clarity to JavaScript code, but another useful tool to help with this is a linter. A linter allows a project to enforce code style and aesthetic choices, often through error messages within a developer's editor. This project-wide unification not only keeps code readable and maintainable but also can reduce errors. An example of this is that a linter will give an error if a variable is unused or if a type is not clearly defined.

For our project, we chose a linter based on our team's experience. In multiple projects a team member had used ESLint, and it proved to be easy to configure. To ensure that we could resolve linter issues, we used Visual Studio Code's ESLint plugin which automatically tags errors/warnings. Regardless of a developer's choice of editor, we ensured that the linter would be enforced by using husky, a library that lets us write scripts that run when GitHub actions occur. For our project we used a script that runs our linter before a developer commits code; if it does not pass the linter checks, the commit is not made.

Mockups: Figma

We began designing our front-end by creating mockups, or example screens the user would interact with. To create these mockups, we used a tool called Figma, a free design platform that lets users collaboratively build the screens of an application (Figma, n.d.). The goal of creating these screens was to facilitate brainstorming and not to display a final design. The reason for this was that we set out to complete our front-end in a single term, and as we develop, the visuals/features may change due to time or scope constraints. We have attached a few of our mockups to Appendix A, and the complete set of mockups can be found by visiting this link: https://www.figma.com/file/wmbnZqtLlZFogAqT9FPLcF/7Factor-Webhook.

Component Library: Ant Design

There are many component libraries for React that exist. The component libraries we explored for our project were Ant Design and Material UI (Ant Design, n.d.; *MUI*, n.d.). We choose our component library based on the library that could support our mockups. Originally, we wanted to use MUI because it is the most popular library according to several sources (Durga Prasad Acharya, 2022; Thomas De Moor, 2022; Vaishnavi Parameswaran, 2021). However, as we investigated the components each library provided, we felt that Ant Design provided both better examples and specific components more suited for our project. Specifically, the Mentions component Ant Design provided allowed us to quickly implement an autofill feature seen in the configuration builder feature discussed in the results section. Therefore, we picked Ant Design as our component library.

Additional Libraries

React Flow

React Flow is a library that contains the components to create an interactive graph of nodes and edges (ReactFlow, n.d.). Our vision for this library was to create a visual representation of a configuration or the connection between a producer and consumer. We felt that when compared to the alternative textual representation, a visual graph would allow connections between producers and consumers to become arbitrarily complex in the future without a complete redesign. The library itself is extremely small and appears well-maintained according to its npm package statistics (webk1d, 2022).

Auth0

Auth0 is a authentication solution that aims to connect applications using varying technologies to external identity providers such as Google, GitHub, and Facebook (Auth0, n.d.-b, n.d.-a). In talking with our client, 7Factor, they recommended Auth0 because they already have a license for the service and use it within their organization.

Results

Back-End

The resulting backend for our project consists of two distinct parts. One aspect of the back-end is a Webhook Dispatch System (WDS) that is responsible for receiving a payload, validating the payload against a user-defined schema, transforming the schema to a format compatible with each destination, and sending out the resulting payload to each destination. We also incorporated a single DynamoDB table that houses all data for our application. The WDS utilizes two Lambda functions known as the Router Lambda and the Child Lambda to perform business logic. We exposed our Router Lambda as the entry point to our WDS via a single API

Gateway endpoint. We also instantiate a Child Lambda for each destination, meaning the relationship between a Router Lambda and its Child Lambdas is one-to-many.

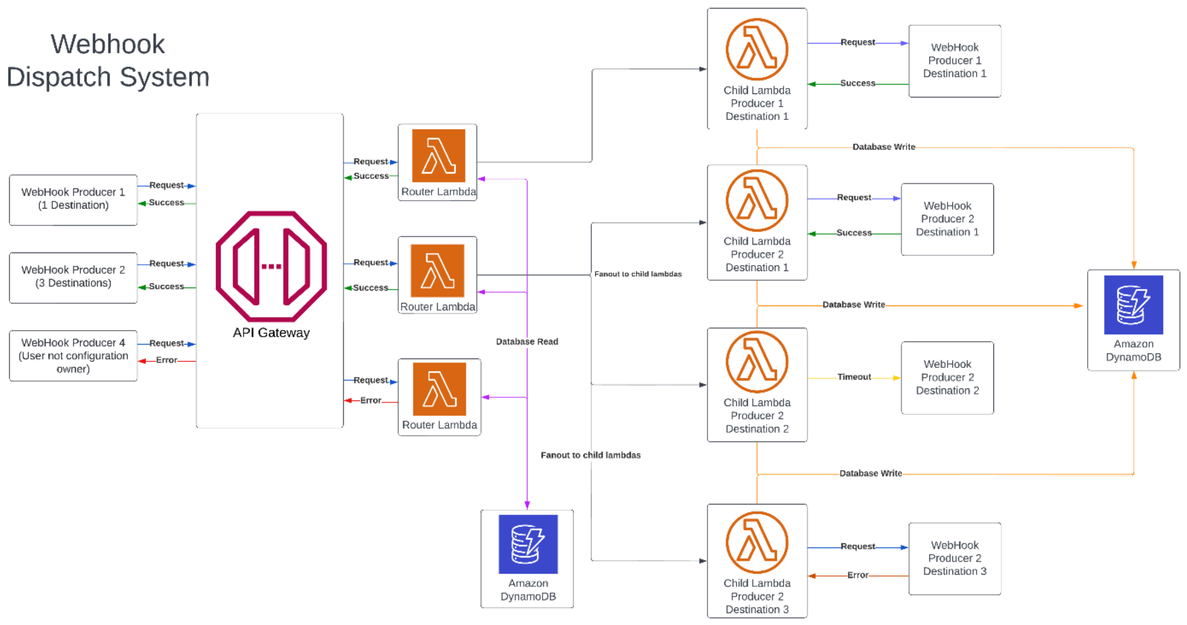

Webhook Dispatch System
WebHook Producer 1 (1 Destination)
WebHook Producer 2 (3 Destinations)
WebHook Producer 4 (User not configuration owner)
Request
Success
Error
API Gateway
Router Lambda
Database Read
Fanout to child lambdas
Amazon DynamoDB
Child Lambda Producer 1 Destination 1
Child Lambda Producer 2 Destination 1
Child Lambda Producer 2 Destination 2
Child Lambda Producer 2 Destination 3
WebHook Producer 1 Destination 1
WebHook Producer 2 Destination 1
WebHook Producer 2 Destination 2
WebHook Producer 2 Destination 3
Timeout
Database Write

Figure 3: An architectural diagram of the Webhook Dispatch System

The second aspect of our back-end was developing a REST API that would allow for our front-end to interact with our DynamoDB table. We used API Gateway to define our endpoints and process all business logic via a single Lambda known as the DB (database) Lambda. Finally, we developed an additional Lambda known as the API Authentication Lambda that is responsible for receiving an Auth0 access token, validating the token, and returning a unique ID that identifies the user.

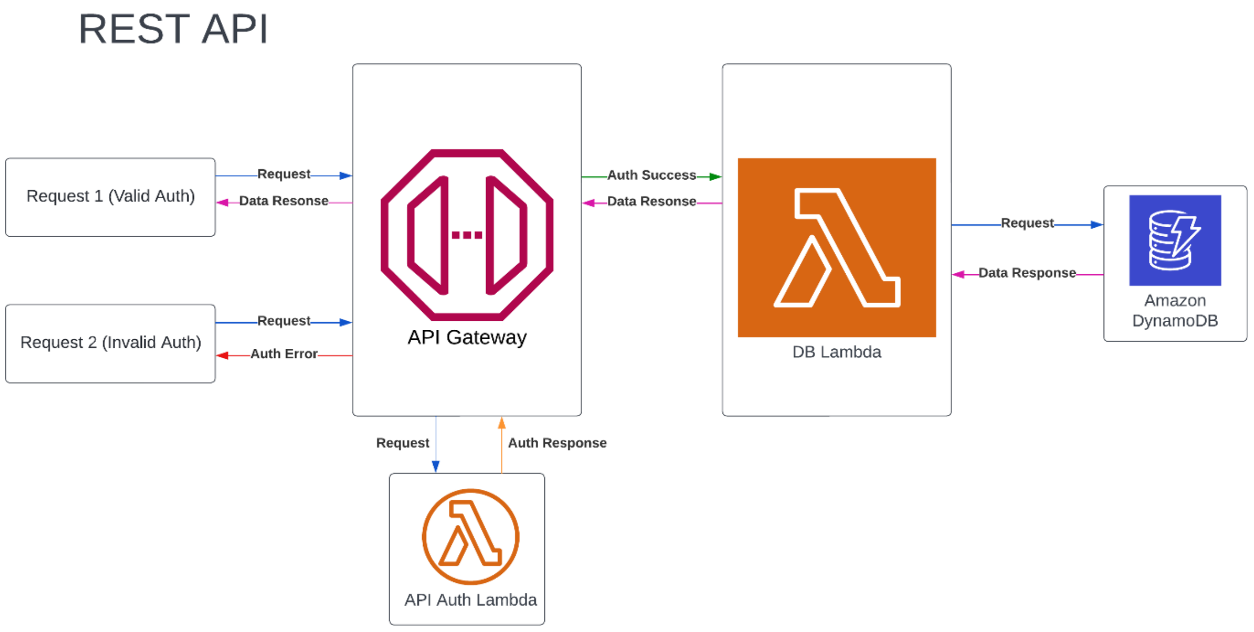


Figure 4: An architectural diagram of the REST API

As previously mentioned, we implemented persistent storage via a single DynamoDB table known as `User_Data`. By using a single table, we eliminated several problems such as the lack of table joins in NoSQL databases (Beswick, "Creating a Single-Table Design with Amazon DynamoDB."). Using a single table required careful planning to ensure that we could perform all CRUD operations necessary. Our product incorporates four different entities that are stored in our database, where one entity, the consumer, is nested within a configuration:

Consumer	
Attribute Name	**Attribute Type**
`consumerID`	UUID
`actionID`	UUID
`targetURL`	URL
`template`	JSON Template Object

Table 1: Consumer entity model

Configuration	
Attribute Name	**Attribute Type**
`userID`	UUID
`entityType`	string
`configurationID`	UUID
`configurationName`	string
`producerActionID`	UUID
`consumers`	Array<Consumer>

Table 2: Configuration entity model

Action	
Attribute Name	**Attribute Type**
`userID`	UUID
`entityType`	string
`actionID`	UUID
`actionName`	string
`payloadSchema`	JSON Schema Object

Table 3: Action entity model

Record	
Attribute Name	**Attribute Type**
`userID`	UUID
`entityType`	string
`recordID`	UUID
`configurationID`	UUID
`consumerID`	UUID
`recordType`	PENDING | SUCCESS | ERROR | TIMEOUT
`responseBody`	object
`statusCode`	number

Table 4: Record entity model

As shown in the tables, each entity contains the following: a `userID` attribute to denote who owns the entity, an attribute that houses a unique ID (UUID) for each entity, and an `entityType` attribute that is a combination of a description of the type of entity as well as the unique ID attribute. Using these three attributes, we were able to formulate unique primary keys for each entity. In specific, we used the notion of a composite key to accomplish this. A composite key is when more than one attribute is used to formulate a key (Balasubramanian, 2017). The following table depicts the format of keys for each of our entities:

Entity Name	**Partition Key**	**Sort Key**
Configuration	`userID`	`CONFIGURATION<configurationID>`
Action	`userID`	`ACTION#<actionID>`
Record	`userId`	`RECORD#<configurationID>:<consumerID>:<recordID>`

Table 5: Entity key structure

By formatting our keys this way, we were able to leverage the `begins_with` DynamoDB function in our conditional expressions when querying. The `begins_with` function takes in a string argument as well as a substring argument and checks if the given string starts with the supplied substring. This function proved to be helpful in our implementation in several ways. For example, assume we were required to query all configurations owned by a given user. By using the `begins_with` function, we could query for all entities with a partition key equal to the supplied user's ID and a sort key that begins with "`CONFIGURATION`". In specific to the record entity, by formatting the sort key with two ID attributes, we open a wide range of access patterns. Firstly, one can query all records owned by a user, as explained previously. Secondly, one can query all records for a given configuration by finding all sort keys that begins with "`RECORD#<configurationID>`". Lastly, one can query for all records for a given consumer by querying for records that have a sort key that begins with "`RECORD#<configurationID>:<consumerID>`". Many of these access patterns are exposed via REST API endpoints, which will be discussed in detail in a later section.

Router Lambda

The Router Lambda begins by receiving data from the "`/webhook`" endpoint on our API Gateway instance. As a result, it acts as the entry point for our WDS. When a user sends a request to this endpoint, they are required to supply a `configurationID` via query parameters. This ID is then used to query for a configuration stored in our DynamoDB table. If a configuration exists, the Router Lambda's next step is to query for the producer action using the `producerActionID` attribute found on a configuration. The producer action informs the Router Lambda of what it should expect the schema of the incoming payload to be. Once the Router Lambda verifies that the incoming payload matches the defined schema, it begins the process of dispatching consumers to their own individual Child Lambdas which in turn are responsible for the actual delivering of messages. These consumers are found on the configuration that was retrieved in a previous step.

The Router Lambda uses the "Event" invocation type when instantiating Child Lambdas to asynchronously invoke them. This is extremely important to our implementation since the Router Lambda can be responsible for spinning up multiple Child Lambdas at any given time. If a Child Lambda were to take a long time to finish executing because of a consumer, we wouldn't want to halt the execution of other consumers. Once the Router Lambda finishes invoking all necessary Child Lambdas, it immediately terminates and sends a response back to the producer.

Child Lambda

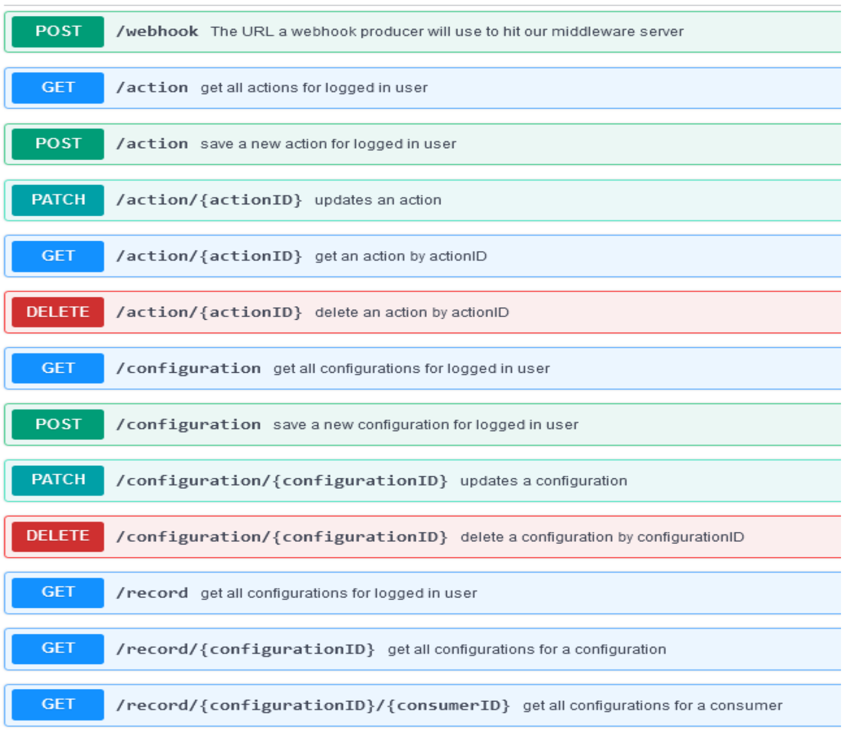
As previously described, the Child Lambda's main purpose is to deliver a transformed payload to a single consumer. To do so, the Child Lambda begins by saving an initial record with the "PENDING" `recordType`. This provides a base record in our database that signifies that the Child Lambda has received the data from the Router Lambda and is about to begin its work. If any record stays as "PENDING", then we know that the Child Lambda did not work properly, as it will always update the `recordType` to one of "SUCCESS", "ERROR", or "TIMEOUT". The next action the Child Lambda performs is transforming the initial payload passed to the Router Lambda to a format friendly to the specific destination it is dealing with. By using the `template` attribute found on a consumer which is supplied to the Child Lambda at instantiation, we can create a new payload object based on the initial payload object. Lastly, the actual request is performed by sending an HTTP POST request to the `targetURL`. There are three outcomes from the request that influence how the initial record that is currently "PENDING" is updated:

1. The request succeeds and `recordType` is updated to "SUCCESS"
2. The request errors and `recordType` is updated to "ERROR"
3. The request times out and `recordType` is updated to "TIMEOUT"

In all the above cases, the resulting status code and response body are also added to the record. At this point, the Child Lambda would terminate and the entire process of the WDS is complete.

API Gateway

API Gateway acts as the middleman between the outside world and our system. It does this by exposing several REST API endpoints. In our API Gateway instance, there are two distinct types of endpoints. The first endpoint, "`/webhook`", on our API Gateway instance allows a webhook producer to initiate the WDS process. This endpoint is directly tied to our Router Lambda. All other endpoints on our API Gateway instance are designed to allow our front-end to perform CRUD operations on our DynamoDB Table.



POST	`/webhook`	The URL a webhook producer will use to hit our middleware server
GET	`/action`	get all actions for logged in user
POST	`/action`	save a new action for logged in user
PATCH	`/action/{actionID}`	updates an action
GET	`/action/{actionID}`	get an action by actionID
DELETE	`/action/{actionID}`	delete an action by actionID
GET	`/configuration`	get all configurations for logged in user
POST	`/configuration`	save a new configuration for logged in user
PATCH	`/configuration/{configurationID}`	updates a configuration
DELETE	`/configuration/{configurationID}`	delete a configuration by configurationID
GET	`/record`	get all configurations for logged in user
GET	`/record/{configurationID}`	get all configurations for a configuration
GET	`/record/{configurationID}/{consumerID}`	get all configurations for a consumer

Figure 5: All endpoints on our API Gateway instance

API Authentication Lambda

All CRUD endpoints also require an Auth0 access key passed via request headers. This key is then validated using our API Authentication Lambda. The API Authentication Lambda is responsible for validating the access key against our Auth0 instance and returning a unique ID that is tied to that access key. This unique ID is what we use for the `userID` attribute on all our entities. Once API Gateway has used our API Authentication Lambda to generate a `userID`, we then forward the request to our DB Lambda which acts as a handler for all CRUD operations. The generated `userID` is also passed as an argument to our DB Lambda.

DB Lambda

The DB Lambda begins by identifying what endpoint it is operating for. It does this by constructing a route key which is of the form "`<HTTP METHOD> <REST RESOURCE>`". An example route key for saving a configuration would look like "`POST /configuration`". Once the DB Lambda identifies the source request, it uses a switch statement to correctly process data for each CRUD request. In general, the DB Lambda can create a new entity, query for entities, update an existing entity, and delete an existing entity. Lastly, the DB Lambda returns either the new entity, the queried entities, the newly updated entity, or the entity that has been deleted.

Front-End

Our front-end system consisted of a variety of displays that showcase the functionality of our back-end. To describe the front-end, we break this section into two sub sections. The first of these sub sections describes the different screens the application contains and their purpose at a high level. The second section elaborates on the specific features each screen contains.

Screens

Login

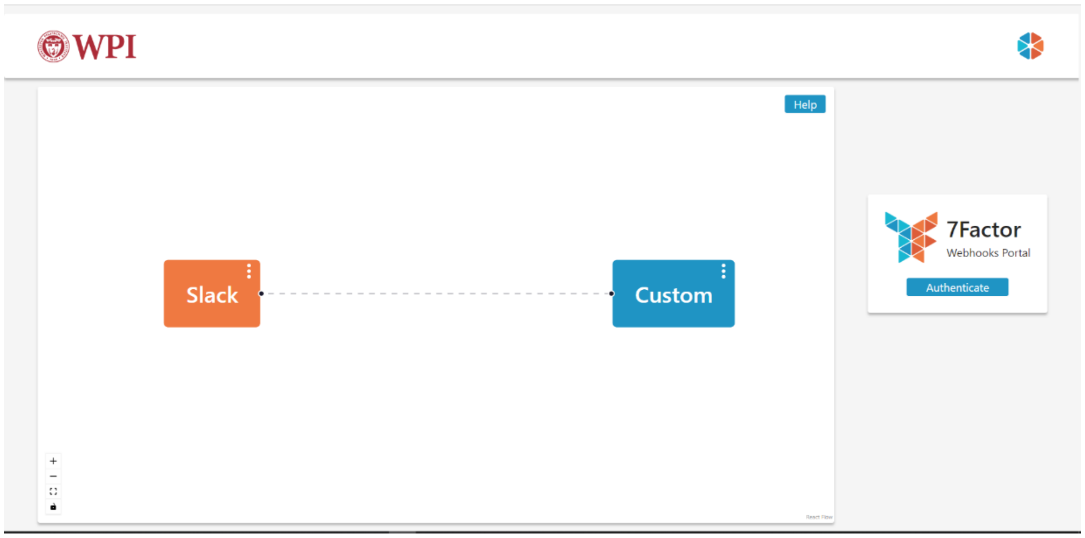
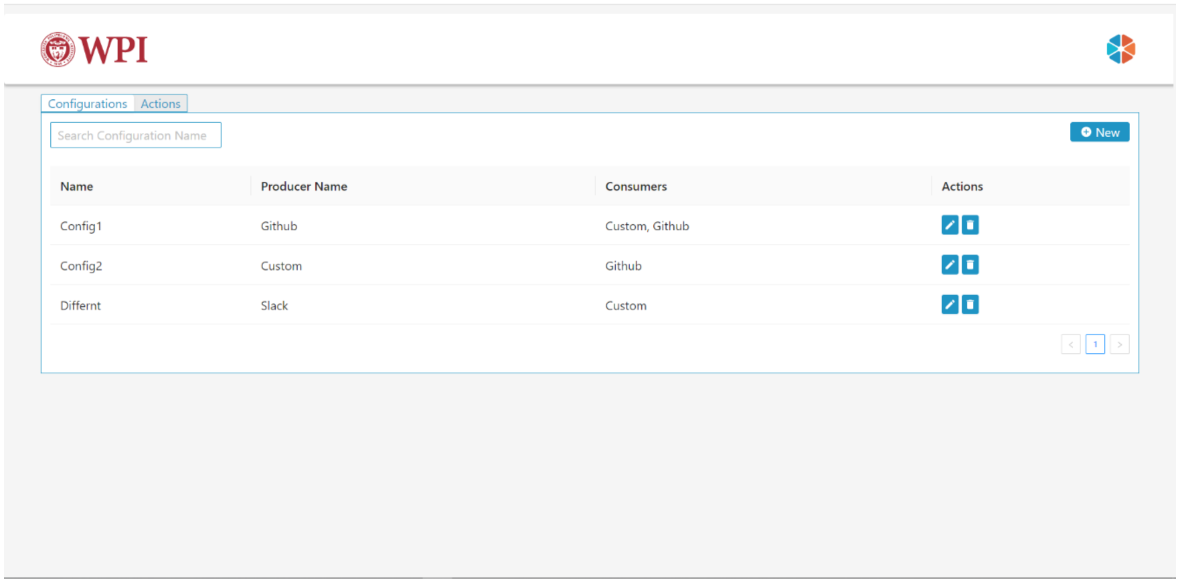


Figure 6: Login Screen

The login screen is made up of two parts: a configuration builder and an authentication tile. The configuration builder is populated with mock data and lets prospective users experiment with our product before signing up. The authentication tile provides a button for the user to authenticate with Auth0.

Home

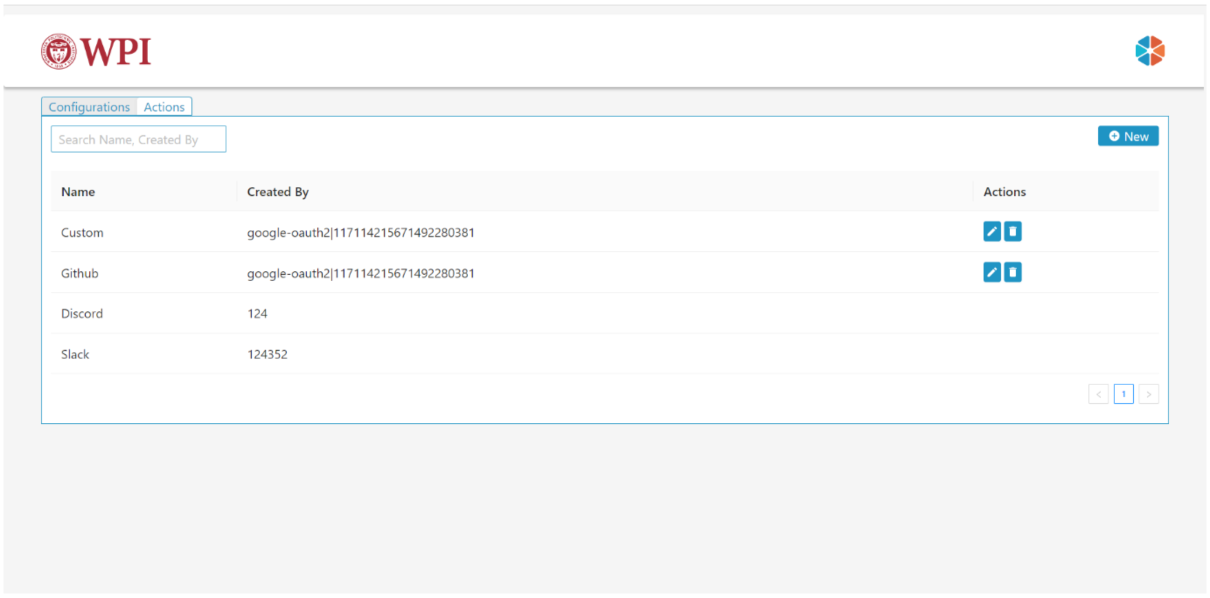
The home screen contains two different sections that display database information: Configurations and Actions. A user can switch between each display by clicking on either of the tabs in the top left corner. Within each of these views a user can perform various modifications to both configurations and actions and search for configurations or actions via the search bar at the top left corner.



Name	Producer Name	Consumers	Actions
Config1	Github	Custom, Github	
Config2	Custom	Github	
Differnt	Slack	Custom	

Figure 7: Home Screen – Configurations Tab

The configurations display shown above lists the configurations for the current logged in user. For now, a user can only view their own configurations. This table captures the most important attributes of a configuration, including the configurations name, the name of the producer, and the names of the consumers. For each existing configuration, a user can perform two different actions: an edit and a deletion. If a user selects the edit action, they will be taken to the configuration screen, and their configuration will be loaded into the configuration builder. If they choose the delete action, the required delete operation will occur in the database, and the page will be refreshed. It is also possible for a user to create a new configuration by clicking on the new button in the top right corner. Like the edit action, this will also take the user to the configuration builder. However, it will load in a default configuration instead of a selected configuration.



Name	Created By	Actions
Custom	google-oauth2|117114215671492280381	
Github	google-oauth2|117114215671492280381	
Discord	124	
Slack	124352	

Figure 8: Home Screen – Actions Tab

The action display shows the actions for all users and not just the authenticated user. The reason we chose to show all actions is that users can use these actions within their own personal configurations. Each action that is owned by the currently authenticated user can have two actions performed on them: an edit and a delete. Editing an action brings up an action builder from within a modal popup in which the user can use to edit the actions schema. Deleting an action performs the delete operation within the database and refreshes the page. A user can also create a new action. Like an edit, an action builder is brought up within a modal popup, but instead is filled with a schema with no attributes.

An issue that stems from allowing users to edit and delete existing actions is handling configurations that reference the edited/deleted actions. In the case of editing or deleting an action, the transformation will not break on the back-end because the template created that maps the producer to the consumer will still exist. However, there may be issues with editing a configuration that refers to a deleted or edited action. This is something we leave to the next MQP group.

Configuration

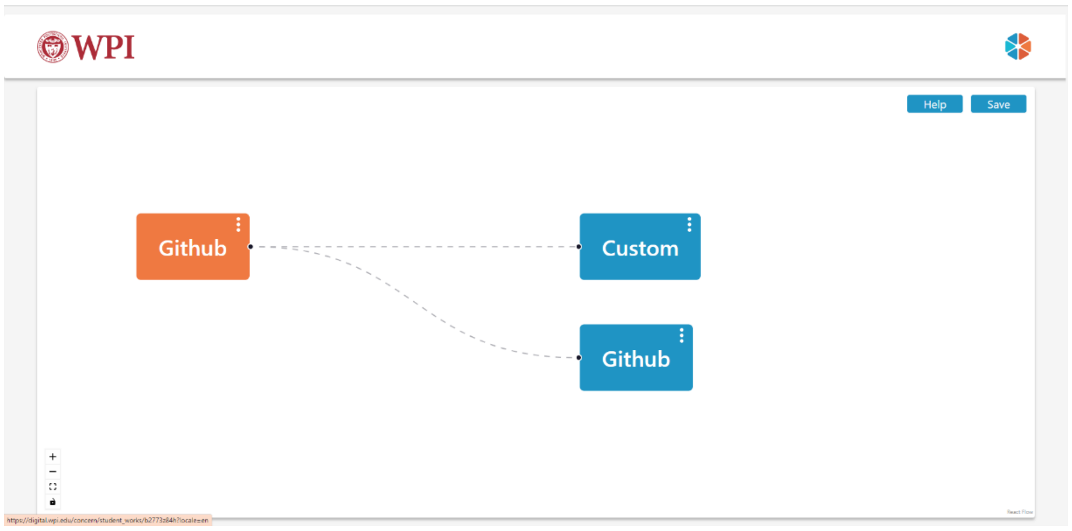


Figure 9: Configuration Screen

The configuration screen uses the configuration builder to allow a user to build a new configuration or edit an existing configuration. These two operations are split into two separate routes: `/configuration/new` and `/configuration/edit/{ID}`. These two routes bring the user to the same screen, but the arguments are parsed so that the different operations are performed.

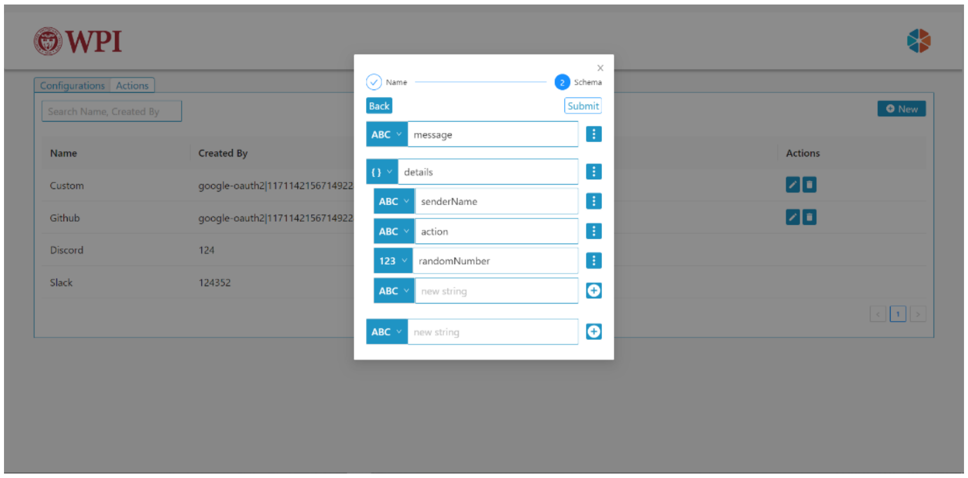
Features

Authentication

We implemented authentication by using page redirection based on the user's authentication status. We use redirection whenever a user navigates to our application. Once they access a route, code runs that checks the existence of a user's session token stored within the browser's memory via the Auth0 React library. If the token does not exist, it has either expired, or the user has never signed in before, so our application redirects them to the authentication screen. If the user has signed in and the token is not expired, then they will be redirected to the home screen. Additionally, this redirection feature is used to prevent unauthenticated users from visiting a part of the application outside of the login screen. If they try to access such a route, they will be redirected to the login screen. Redirection also protects a user's private resources.

For example, if a user visits the route `/configuration/edit/123` when they do not own the configuration with the ID 123 they are redirected to the home page.

Action Builder

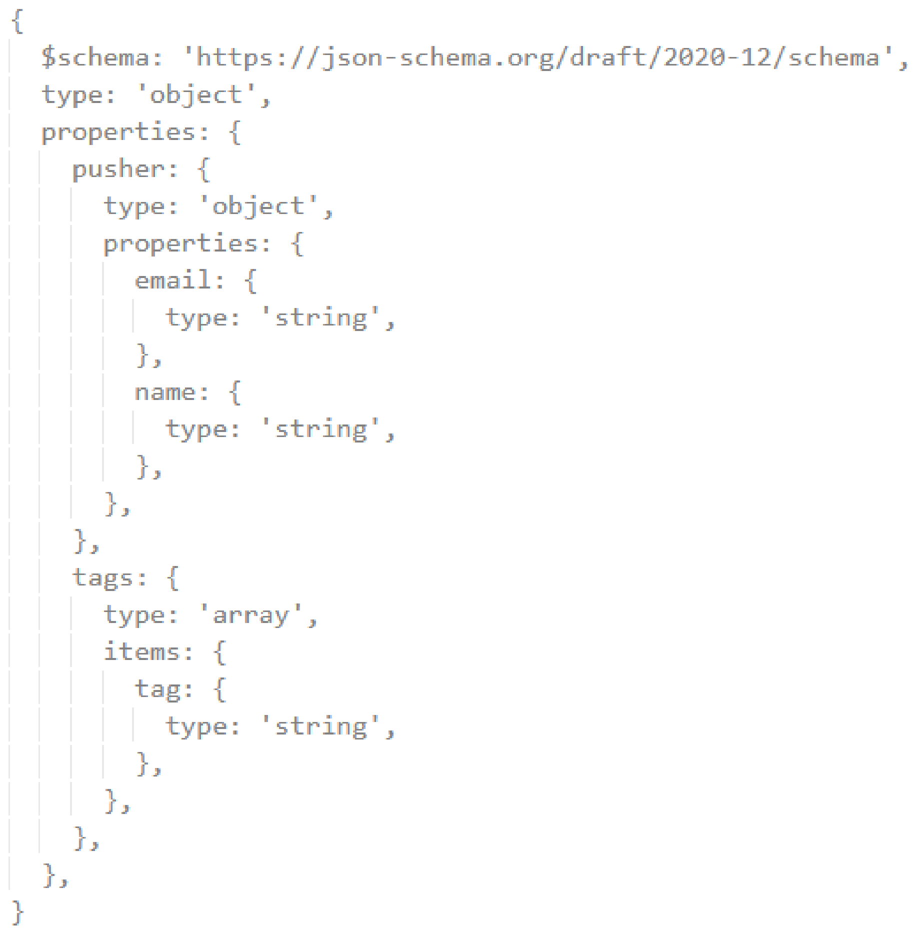


Name	Created By	Actions
Custom	google-oauth2|1171142156714922	
Github	google-oauth2|1171142156714922	
Discord	124	
Slack	124352	

ABC message
{ } details
ABC senderName
ABC action
123 randomNumber
ABC new string
ABC new string

Figure 10: Action Builder

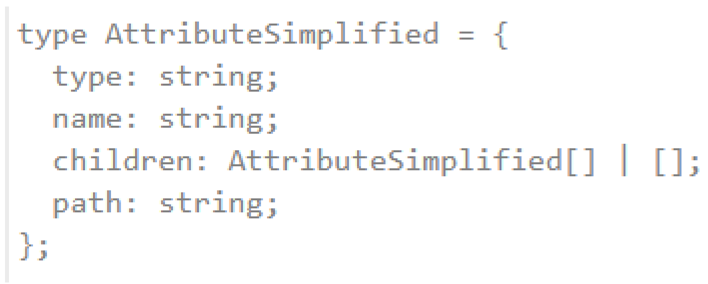
The action builder is a simplistic display that allows users to enter in multiple attributes which are comprised of a type and a name to create a JSON object according to the 2020-12 JSON Schema specification. The types of attributes the JSON Schema supports are string, boolean, number, object, and array. Here is an example of an object that follows the specification.



```
{
  $schema: 'https://json-schema.org/draft/2020-12/schema',
  type: 'object',
  properties: {
    pusher: {
      type: 'object',
      properties: {
        email: {
          type: 'string',
        },
        name: {
          type: 'string',
        },
      },
    },
    tags: {
      type: 'array',
      items: {
        tag: {
          type: 'string',
        },
      },
    },
  },
}
```

Figure 11: JSON Schema example

When the action builder is displayed on the screen, a React context is being used to store the schema in a central place where all components within the action builder have access to it. The fields seen within the UI are dynamically created from this central JSON Schema object by first parsing it into the Typescript type shown below.



```
type AttributeSimplified = {
  type: string;
  name: string;
  children: AttributeSimplified[] | [];
  path: string;
};
```

Figure 12: Attribute Simplified TypeScript type

To perform this parsing, an algorithm was developed. This algorithm was designed to avoid recursion, despite the intuitiveness for using it for this problem. This decision was made to make debugging and the future developer experience easier and better. A limitation that exists within the algorithm was representing an array of an array. Due to the structure of the data, a type of this nature complicated the algorithm and would have required too much time to develop a solution. We decided to leave this to the next project team.

To create an attribute a user can select a type from the drop-down menu on the left side of the field. They can then edit the attributes name via the corresponding input and add the attribute by clicking the plus button. When a user adds an object or array attribute, they can add sub attributes to the created attribute. Objects can have multiple sub attributes of any valid type. However, arrays can only have sub attributes that are of the type string, number, boolean, or object. Another constraint we imposed upon array types is that they can only have a single attribute associated with them. This restriction is not imposed by the algorithm, but rather a semantic decision made by the team. Our justification for this is that it greatly complicated the type casting that would need to be done on the back-end during a transformation. We justified this decision by looking at the providers our client 7Factor planned to use and seeing that none of them included multi-type arrays.

To edit an existing attribute the user can manipulate the type and attribute as they see fit. All changes directly impact the JSON Schema stored within the context. To delete an existing attribute the user must click the vertical ellipses and select the delete icon.

A power user feature we added to the action builder is being able to hide object/array sub attributes. We did this because when we tested deeply nested objects the screen became cluttered and difficult to navigate. To toggle the visibility of the sub attributes, the user can click the hidden and visible eye icons included within the vertical ellipses drop down.

Configuration Builder

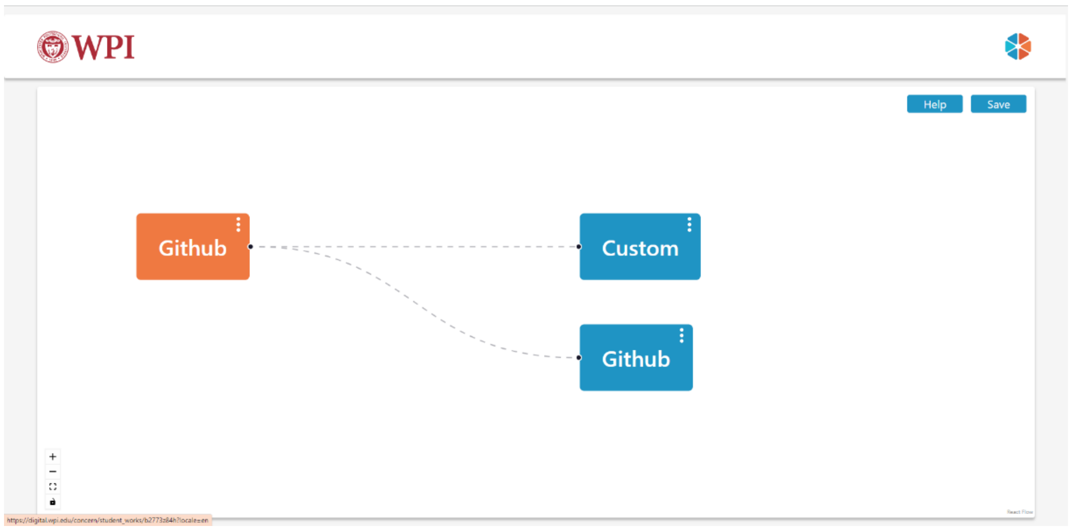
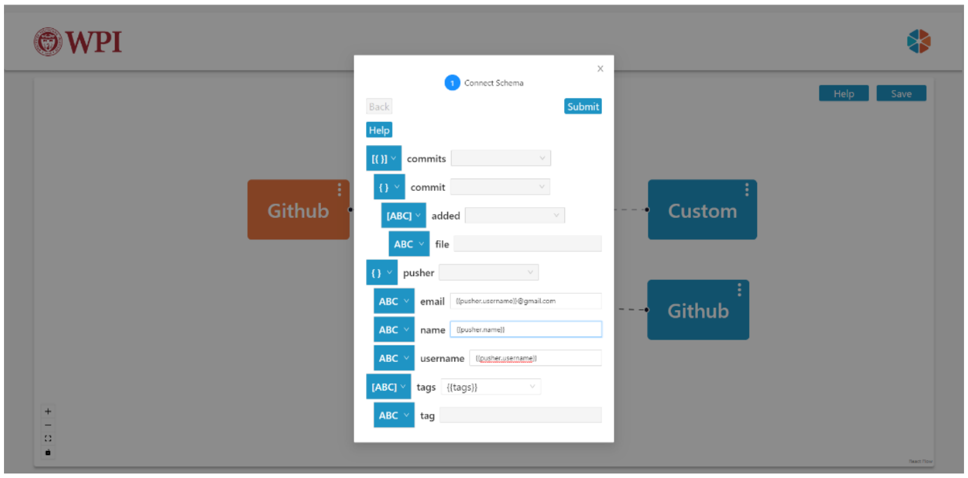


Figure 13: Configuration Builder

The configuration builder provides a visual interface to create connections between a producer (orange node) and consumer(s) (blue node). Given our time constraints in building this application, we only support a single producer node within a configuration. However, numerous consumers may be created. The user can create a consumer node by right clicking anywhere withing the configuration builder and selecting the "Create Producer" option from the drop down that appears.

Users can drag the producer/consumer nodes around the grid and connect them via clicking on the small black dot present on each node and dragging the line to the black dot of the opposite node type. Upon connection, an action connector will appear that lets a user pick the mapping between the producer's payload and the consumer's inputs. Upon completing this mapping, an edge between the two nodes will be created which can be edited by double clicking on it.



[{ }] commits
{ } commit
[ABC] added
ABC file
{ } pusher
ABC email {{pusher.username}}@gmail.com
ABC name {{pusher.name}}
ABC username {{pusher.username}}
[ABC] tags {{tags}}
ABC tag

Figure 14: Configuration Builder – Action Connector

The mapping we create is referred to by the back-end as a template. The template is used to transform a producer's payload into the format the consumer needs. We map the producer's attributes to the consumer's attributes in two ways. The first way is by selecting a producer attribute from a drop-down menu. This can be done only for the following types: array of strings, array of numbers, or array of booleans. The second option is a field where a user can type custom input by referencing attributes from the producer. This can be done for the following types: number, string or boolean. The user can bring up the usable list of attributes by entering {{ into the input box. Doing this results in a drop down where the user can select the attribute they wish to include. When a attribute is selected, the attribute's path is filled into the input followed by }} is added to the input. In the example image above, the email field is created from appending @gmail.com to the producers' `pusher.username` attribute. To create this effect, we leveraged Ant Design's mention component. The mention component works the same way as user tagging on social media. It requires a key string such as the @ symbol to bring up a dropdown and display a list of possible options.

The template that is created for a producer-consumer relationship does not need to have a mapping or text value for every field. Instead, fields can be left blank, and defaults will automatically be inserted. The following are the defaults for each type.

Boolean: False

Number: 0

String: “”

Object: {}

Array: []

Other actions can also be performed on the producer/consumer nodes. These actions can be reached by clicking the vertical ellipses located on the top right of each node. The only action that can be performed on a producer is editing. This is because every configuration must have a producer node. The actions a user can perform on a consumer are editing and deleting. Editing either a producer or consumer node will reset all the edges that exist between the producer and consumers(s). Deleting a consumer will remove the node from the display and remove its edge to the producer.

Each action that is done on a producer or consumer modifies a list of node and edges. These lists are propagated to subcomponents via a React context, allowing them to make changes. Once the user clicks save/create--depending on the action they are performing on a configuration--the nodes and edges are transformed into a configuration that is either updated/inserted into the database.

Recommendations

Throughout the course of this project, we have come up with numerous recommendations for future teams to implement. In this section we describe each of these recommendations.

We recommend that the next group that works on our MQP creates a mechanism that converts an API to a webhook. Many services such as Harvest, the time tracking platform 7Factor uses, do not provide webhooks. Instead, they have a developer API. A developer can create a rudimentary webhook from this by polling the API and sending data when a change occurs. The next team could explore setting up infrastructure that creates scheduled jobs within the AWS cloud that checks an API for a change.

As a team we believe the power in our approach is delegating the creation and maintenance of action schemas. With this, we hope to create a user-driven community that will benefit the masses. A few features we felt a future team could implement to strive toward a user driven community are the ability to save any action for easy access, creating public and private actions, and improving the UI and algorithm used to create an action schema.

Currently our solution is deployed on our own AWS account. This makes it difficult for 7Factor to take advantage of our solution because we would have to manually reconfigure our architecture on their account. Instead of this, a future team should build a AWS SAM/Terraform templates that could automate this process. Additionally, during the creation of this infrastructure as code the front-end should be deployed to the cloud.

Our next recommendation is to develop security measures for our WDS. Currently, any request with a valid `configurationID` query parameter can instantiate our WDS regardless of ownership. As a result, the following team should research and implement an authentication strategy on the "`/webhook`" endpoint.

At the start of our project, we explored implementing a retry queueing system for failed consumer requests such as Amazon SQS. Due to a lack of time, we decided to focus on the core functionality of our application instead of a queueing system.

Another stretch goal we had set for ourselves was to load test our WDS. Given that we designed our infrastructure to be able to handle large one-to-many relationships from a producer to several consumers, we believe it would be wise to verify that our infrastructure performs as expected.

Lastly, our project is currently split into five different repositories. As a result, it is challenging to share type definitions across different components. Therefore, the following team should create an NPM package that holds the definitions for all central types.

Conclusion

The lack of a cost effective and flexible solution to workplace automation using webhooks influenced 7Factor to investigate creating their own webhooks-as-a-service platform. Our team began to solve this problem through the creation of the first version of a community driven webhooks-as-a-service platform. Our service provides a scalable and affordable cloud hosted solution that provides more customization over webhook configuration than other services provide. This platform includes features like multiple destination webhooks and the ability for users to fully customize webhook payloads. In addition to the cloud architecture we developed, we also built a web-application where authenticated users can configure webhook configurations graphically through a user-friendly UI.

References

7Factor. (2022, July 28). *7Factor*. https://www.7factor.io/

About: Polling (computer science). (n.d.). Retrieved December 7, 2022, from https://dbpedia.org/page/Polling_(computer_science)

Abuhakmeh, K. (2022, April 27). *Long-Polling to get recent updates—.NET Guide*. JetBrains. https://www.jetbrains.com/dotnet/guide/tutorials/htmx-aspnetcore/long-polling/

Agile Project Management—The Beginners Guide. (n.d.). Teamwork. Retrieved December 7, 2022, from https://www.teamwork.com/project-management-guide/agile-project-management/

Amazon DynamoDB. (n.d.). https://aws.amazon.com/dynamodb/

Ant Design. (n.d.). *Ant Design*. Retrieved December 4, 2022, from https://ant.design/

Atlassian. (n.d.). *What is Jira Software used for?* Atlassian. Retrieved December 7, 2022, from https://www.atlassian.com/software/jira/guides/use-cases/what-is-jira-used-for

Atlassian. (2020, January 9). *Webhooks*. https://developer.atlassian.com/server/jira/platform/webhooks/

Auth0. (n.d.-a). *Auth0 | Auth0 Introduction*. Auth0. Retrieved December 5, 2022, from https://auth0.com/resources/videos/platform-introduction-video-2020

Auth0. (n.d.-b). *Auth0: Secure access for everyone. But not just anyone.* Auth0. Retrieved December 5, 2022, from https://auth0.com/

Auth0. (n.d.-c). *Social Connections for Auth0*. Auth0 Marketplace. Retrieved December 5, 2022, from https://marketplace.auth0.comundefined

AWS. (n.d.). *Event-Driven Architecture*. Amazon Web Services, Inc. Retrieved November 30, 2022, from https://aws.amazon.com/event-driven-architecture/

Balasubramanian, G. (2017, February 20). *Choosing the Right DynamoDB Partition Key*. https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/

Beswick, J. (2021, July 26). *Creating a single-table design with Amazon DynamoDB*. https://aws.amazon.com/blogs/compute/creating-a-single-table-design-with-amazon-dynamodb/

Cloud Computing Services - Amazon Web Services (AWS). (n.d.). Retrieved December 18, 2022, from https://aws.amazon.com/

Durga Prasad Acharya. (2022, November 3). *React UI Components Libraries: Our Top Picks for 2022*. Kinsta®. https://kinsta.com/blog/react-components-library/

Exosite. (n.d.). *Embedded IoT Protocols: WebSocket's Benefits and Drawbacks*. Retrieved December 7, 2022, from https://blog.exosite.com/embedded-iot-protocols-websocket

Figma. (n.d.). *Free Design Tool for Websites, Graphic Design and More*. Figma. Retrieved December 4, 2022, from https://www.figma.com/design/

From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. (n.d.). Retrieved November 21, 2022, from https://ieeexplore.ieee.org/abstract/document/7958457/

Gautam, S. (n.d.). *Long Polling in System Design*. Enjoy Algorithms. Retrieved December 7, 2022, from https://www.enjoyalgorithms.com/blog/long-polling-in-system-design

GitHub. (n.d.-a). *Features • GitHub Actions*. GitHub. Retrieved December 4, 2022, from https://github.com/features/actions

Github. (n.d.). *GitHub Actions*. GitHub. Retrieved November 21, 2022, from https://github.com/features/actions

GitHub. (n.d.-b). *Understanding GitHub Actions*. GitHub Docs. Retrieved November 21, 2022, from https://ghdocs-prod.azurewebsites.net/en/actions/learn-github-actions/understanding-github-actions

Idera Inc. & Frontend Masters. (2020). *Front-end Frameworks*. https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/

Jaramillo, D., Nguyen, D. V., & Smart, R. (2016). Leveraging microservices architecture by using Docker technology. *SoutheastCon 2016*, 1–5. https://doi.org/10.1109/SECON.2016.7506647

JSONSchema. (n.d.). *JSON Schema*. JSON Schema. Retrieved December 6, 2022, from https://json-schema.org/

Kilbride-Singh, K. (2022, October 17). *WebSockets vs Long Polling*. Ably Realtime. https://ably.com/blog/websockets-vs-long-polling

Molina, J. (2022, June 13). *When to Use Webhooks vs. WebSockets*. Symbl.Ai. https://symbl.ai/blog/when-to-use-webhooks-vs-websockets/

MUI: The React component library you always wanted. (n.d.). Retrieved December 4, 2022, from https://mui.com/

npm. (n.d.). *About npm | npm Docs*. Retrieved December 13, 2022, from https://docs.npmjs.com/about-npm

Pautov, M. (2021, October 15). *Pros and Cons of WebSocket and EventSource*. HackerNoon. https://hackernoon.com/pros-and-cons-of-websocket-and-eventsource

ReactFlow. (n.d.). *Introduction—React Flow*. Retrieved December 5, 2022, from https://reactflow.dev/docs/introduction/

Sarabyn, K. (n.d.). *What are Webhooks?: An Explanation for the Non-Technical*. Pandium. Retrieved December 4, 2022, from https://www.pandium.com//blogs/what-are-webhooks-an-explanation-for-the-non-technical

Shawn Wang & Mark Erikson. (2021). *Front-end Frameworks*. https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/

Sync Penguin. (n.d.). *Pricing & plans | SyncPenguin*. Retrieved November 21, 2022, from https://syncpenguin.com/

SyncPenguin. (n.d.). *Supported apps | SyncPenguin*. Retrieved November 21, 2022, from https://syncpenguin.com/pricing/

Testim. (2021, June 18). *What Is a Linter? Here's a Definition and Quick-Start Guide*. AI-Driven E2E Automation with Code-like Flexibility for Your Most Resilient Tests. https://www.testim.io/blog/what-is-a-linter-heres-a-definition-and-quick-start-guide/

Thomas De Moor. (2022, June 2). *14 Best React Component Libraries in 2022*. X-Team Blog - The Most-Loved Company for Engineers. https://x-team.com/blog/best-react-component-libraries/

Thomson, M., Damaggio, E., & Raymor, B. (2016). *Generic Event Delivery Using HTTP Push* (Request for Comments RFC 8030). Internet Engineering Task Force. https://doi.org/10.17487/RFC8030

Vaishnavi Parameswaran. (2021, July 9). *Top 10 React Component Libraries/Frameworks for 2022*. Atatus Blog - For DevOps Engineers, Web App Developers and Server Admins. https://www.atatus.com/blog/top-10-react-component-libraries-for-2021/

Webhooks vs API Polling. (n.d.). Svix Resoures. Retrieved December 4, 2022, from https://www.svix.com/resources/faq/webhooks-vs-api-polling/

webk1d. (2022, November 28). *Reactflow*. Npm. https://www.npmjs.com/package/reactflow

What is web socket and how it is different from the HTTP? (2022, February 21). GeeksforGeeks. https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/

What is WebSocket and How It Works? ⚙️. (n.d.). A Simple Explanation Of What A WebSocket Is. Retrieved December 7, 2022, from https://www.wallarm.com/what/a-simple-explanation-of-what-a-websocket-is

Zapier. (n.d.-a). *Explore All Apps | Zapier*. Zapier. Retrieved November 21, 2022, from https://zapier.com/apps

Zapier. (n.d.-b). *Plans & Pricing | Zapier*. Retrieved November 21, 2022, from https://zapier.com/pricing

Appendices

Appendix A: Initial Mockups:

Link to the complete set: https://www.figma.com/file/wmbnZqtLlZFogAqT9FPLcF/7Factor-Webhook?node-id=48%3A18838&t=uHP55OkUhnaCkDi4-1

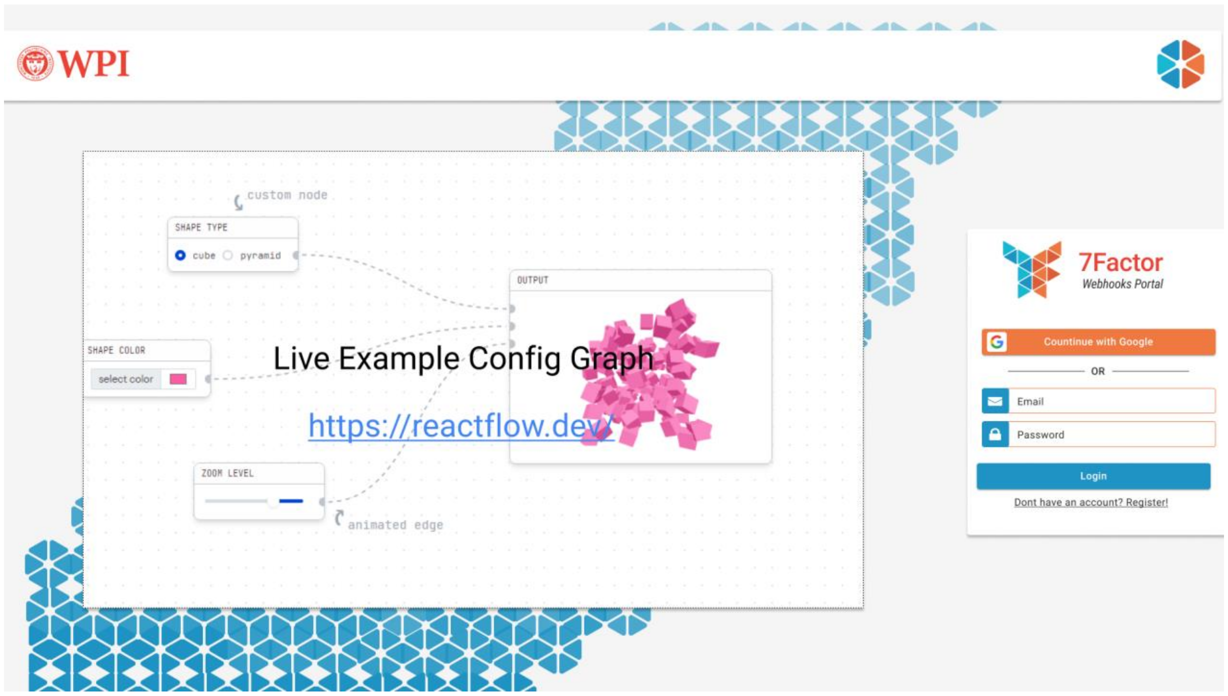
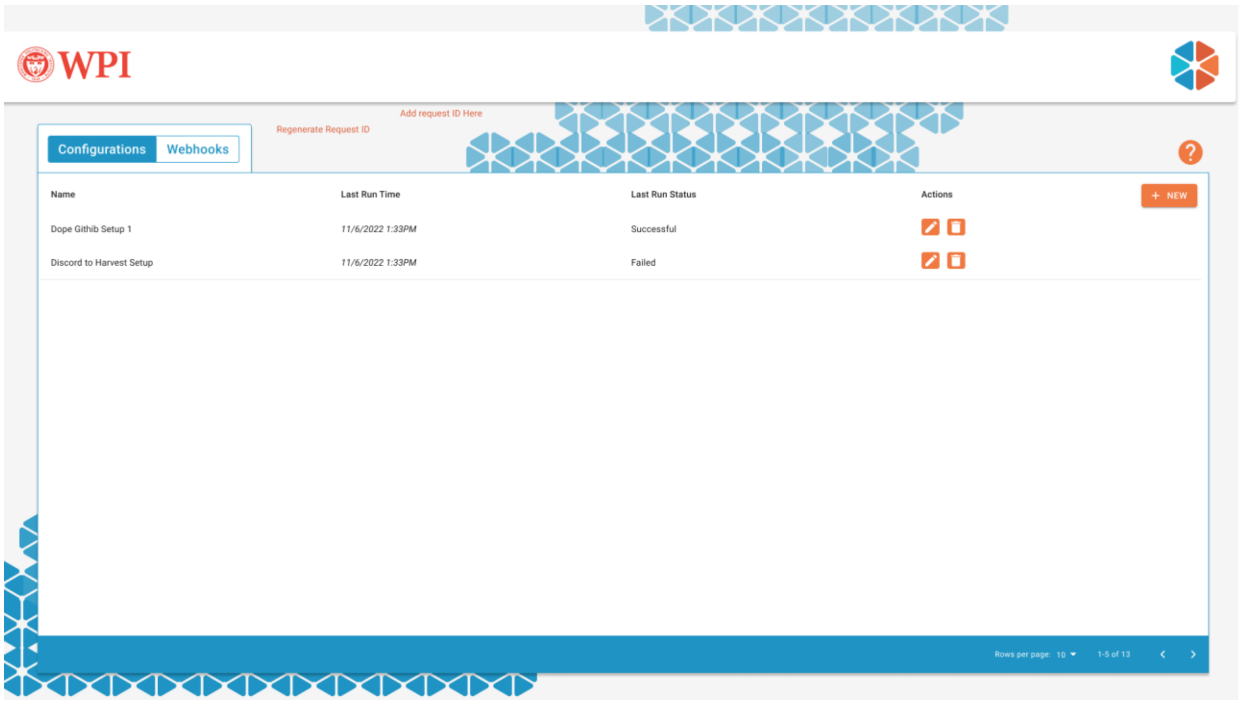


Figure 15: Mockup of login screen.



Name	Last Run Time	Last Run Status	Actions
Dope Githib Setup 1	11/6/2022 1:33PM	Successful	
Discord to Harvest Setup	11/6/2022 1:33PM	Failed	

Figure 16: Mockup of home screen.

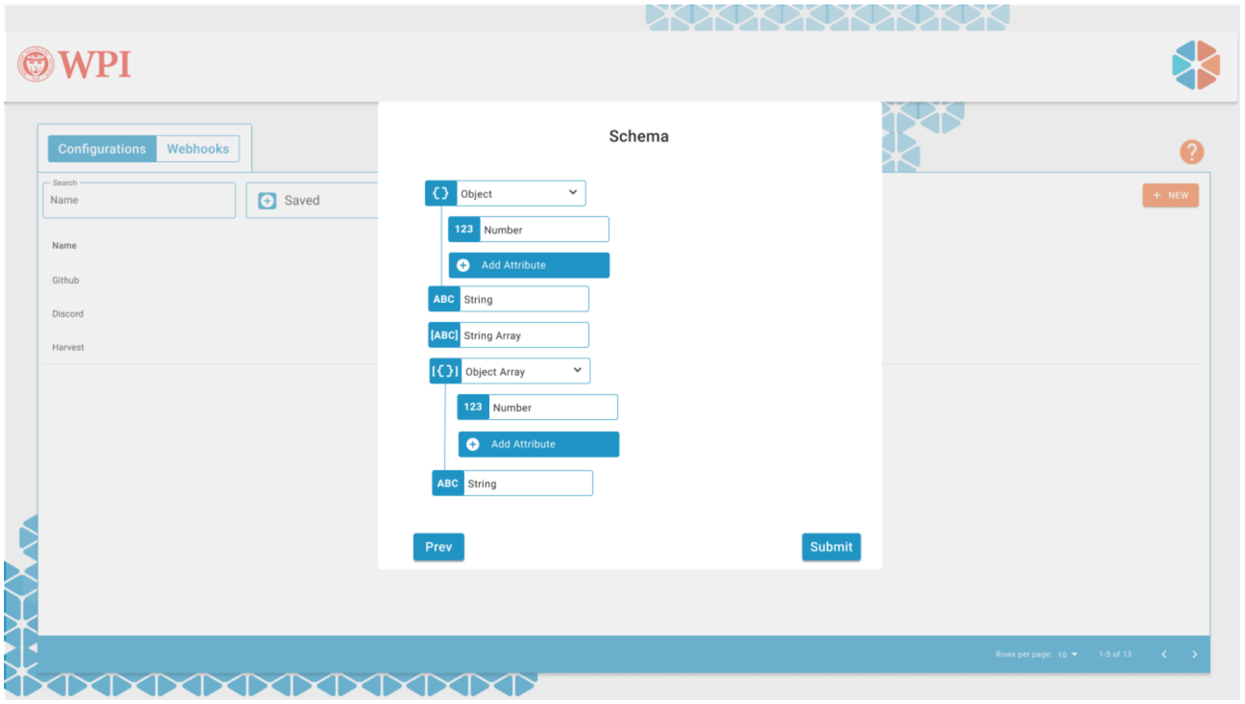


Figure 17: Mockup of action builder.