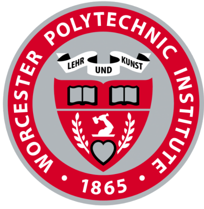


WPI

7FACTOR

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

Major Qualifying Project 2021-2022

Authored by:

Veronica Andrews

Carley Gilmore

Jonathan Rosenbaum

Ryan Stebe

Presented to:

Professor Joshua Cuneo

Sponsored by:

Jeremy Duvall, 7Factor Founder

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its site without editorial or peer review.

Authorship Page

This report was authored and edited by Veronica Andrews, Carley Gilmore, Jonathan Rosenbaum, and Ryan Stebe. Each person contributed equally to the report. Professor Joshua Cuneo provided periodic feedback.

Acknowledgements

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

7Factor, a software company specializing in software delivery and cloud services, was interested in a Webhooks-as-a-Service (WaaS) API that could connect different applications and services in real-time. In this project, the team presents the design and partial development of a WaaS API that allows for the creation of automated webhook connections between third-party APIs. As part of one teammate's Professional Writing MQP, this project also contains a technical writing report and API documentation set.

Table of Contents

Authorship Page 1

Acknowledgements 2

Abstract 3

Table of Contents 4

List of Figures 6

Executive Summary 7

Introduction 8

Background 9
- Webhooks 9
- Applications for Webhooks in Workplace Automation 9
  - Workplace Automation 9
  - Trigger-Action Programming 10
  - Data Synchronization: One-way and Two-way Syncing 10

Methodology 12
- Researching and Choosing Technologies 12
- Database Design 13
- Server back end Design 13
- User Interface Design 13
- Timeline and Project Management 14

Results 15
- Application Design 15
- Prototype Features 18
  - User Interface Technologies: React, Babel, and Webpack 19
  - back end Technologies: Javascript and Node.js 20
  - NodeJS Modules: Express and HTTPS 20
  - Database Technology: SQLite 21
  - Server Hosting Technology: Docker 22

Recommendations 24

References 25

Appendices 27
- Appendix A 27

Appendix B 28
Appendix C 29
Appendix D 30
Appendix E 31
Appendix F 32

List of Figures

Figure 1: Gantt Chart

Figure 2: Account Creation Process Diagram

Figure 3: User Login Process Diagram

Figure 4: Connection Creation and Modification Process Diagram

Figure 5: Incoming Webhook Process Diagram

Figure 6: Database Table Structure

Figure 7: Implemented Database Commands

Executive Summary

This project was sponsored by 7Factor, a software consulting firm based in Atlanta, Georgia. The main point of communication for this project was the founder of 7Factor, Jeremy Duvall. Jeremy tasked the team to create a Webhooks-as-a-Service (WaaS) engine that could be used to set up automated tasks between apps using webhooks such as Concourse and Shortcut, which are two of the services that the company frequently utilizes internally.

The members of the team began the project by researching webhooks, existing tools that provide a similar service as this project such as Zapier, and the individual APIs of the applications that would be used to test the service. With this information, the team created goals and a realistic timeline of what could be completed by the end of the third project term. The team broke the project up into three main modules: the back end server, the database, and the front end.

The top priority was to create a prototype server and database for a future working application, so the team used services they were familiar with–Discord and GitHub–to do manual integration testing throughout the course of the project. The team then designed user interface mockups and began setting up the technological foundation for building a user interface. Lastly, they finalized and tested the server and database components of a foundational prototype and began the process of connecting the server with the database.

The group completed the back end server prototype and a working database supported through Docker containers. They also set up the foundation for React, so a user interface may be easily developed and integrated in the future. Throughout the project, the team communicated regularly with Professor Cuneo–the project advisor–and Jeremy Duvall.

The team then developed a set of recommendations and next steps for 7Factor or other student teams. These steps included developing the user interface, completing the connection between the server and the database, expanding the functionality for the server to more applications, and hosting the server on a domain so it may be accessed by any user.

Introduction

The purpose of this project was to develop a webhooks automation and management tool for the company 7Factor, a software company based in Dunwoody, Georgia, that specializes in delivering high-quality cloud services to their clients. 7Factor works on a contract basis with their clients, delivering the product their client requests, even if it is not cloud-based. According to their website, 7Factor provides their service in three different modes: project work, retainer teams, and advice contracts. The project work and retainer teams are connected, meaning 7Factor will build the application their client requests from end to end and continue to support this application after development is completed. The advice contracts consist of teams that help a client work through a complex issue and debug applications.

The team designed a 7Factor Webhooks management and automation tool that displays current active webhooks and their status and allows the user to add and delete webhooks at will. Webhooks are highly efficient and receptive to consumer needs, offering a practical way to share and manipulate data across web platforms in real-time. This WaaS engine is able to handle payloads from specified version control applications to automate processes in staff management applications. Comparable to products like Zapier and SyncPenguin, 7Factor Webhooks would allow users to create and manage connections between supported applications. For instance, a user can configure a connection so that when a pull request is completed in Github, the 7Factor Webhooks tool can send a message in a Slack channel that the pull request has been made.

The team's design also includes a login and authentication system to prevent unauthorized users from tampering with active webhooks. This authentication system design will allow for one or more administrators to manage the entire system, while also allowing them to add users with specific permissions, such as allowing an employee to create and delete webhooks said employee created.

7Factor requested this product due to their need for a management and automation tool for their multitude of projects for multiple clients. Initially, 7Factor had plans of using Zapier or PieSync, but these tools proved to be too restrictive for their needs. 7Factor determined that the tool they use must have modifiable source code dependent on their current projects. Because of this, they requested the development of a WaaS tool.

Background

Webhooks

Webhooks, also called HTTP push APIs or web callbacks (SendGrid, 2019), are a system for communication over the web that is event-based, meaning that whenever a particular event occurs in an application (the webhook provider), it sends a message containing information about the event to another application (the webhook destination) (Hookdeck, n.d.). Unlike the alternative of polling, this requires a user to subscribe to a webhook so that the source application knows where to send each event (Guay, 2020). This system of communication is usually more efficient than polling because polling requires each receiver to repeatedly send a message containing a request for updates on some kind of information, which results in many extraneous client messages and server responses (Nilsson, 2020). Because webhooks are generally more efficient than polling for both the client and the server , many web applications support the use of webhooks for communicating information about events to clients (SendGrid, 2019). Connections are typically set up by the clients who send subscription requests to the application through the application's specific API (Guay, 2020). Calls to an API can be automated in the client, but the formatting and information required to subscribe to and parse messages from a webhook is typically very different between applications, since each app has its own API and sends its webhook payloads in its own format (SendGrid, 2019).

Applications for Webhooks in Workplace Automation

Workplace Automation

Workplace automation is the implementation of technology to substitute or complement human roles and tasks in the workplace (Autor, 2015). These technologies can include artificial intelligence or any current or future technology that will improve efficiency and productivity in the workplace. It is estimated "that 45 percent of work activities could be automated using already demonstrated technology" (Chui et al., 2015). While there are some ethical concerns about workplace automation because it has potential to eliminate jobs, there are also smaller implementations that can be used to assist employees in mundane or tedious tasks (Brown, 2021). Webhooks are an example of a technology that can be used to automate the communication between commonly used software applications, so all applications can be

updated simultaneously instead of developers or employees needing to manually update each one.

Trigger-Action Programming

There are many uses of webhooks for workplace automation, but this project will compare and contrast two common implementations: trigger-action programming and data synchronization, specifically one-way and two-way syncing. Trigger-action programming is “a programming model enabling users to connect services and devices by writing if-then rules” (Brackenbury et al., 2019). This type of model can be used in applications such as Zapier and IFTTT to allow “non-developers to automate activities across multiple platforms by integrating their functionalities" (Rahmati et al., 2017) by eliminating the need to program. Trigger-action programming has two main parts: the trigger and the corresponding action. This programming framework uses triggers created by the user and corresponding actions to create an event. An example of this would be a push to a GitHub repository (the trigger) that triggers a notification in a specific Discord server about what commits were pushed (the action).

Zapier calls these events “zaps” (Johnson, 2021), and IFTTT calls them “Applets” (IFTTT, n.d.). These services are used for one-way connections that can be used to automate workflows. Using these services, you can automatically connect applications such as Google Calendar and Trello without any knowledge of coding. The downside to these types of services is that they can only be created to be one-way events, and they cannot be continuous. Therefore, the connections end once the payload is sent to the destination application, and these connections have to be reestablished for future events. Zapier and IFTTT allow for more customizable connections than other services–like PieSync–but there are limitations as to what applications can be used. However, both support hundreds of popular applications, so the limitations are not severe.

Data Synchronization: One-way and Two-way Syncing

Data synchronization is the continuous process of synchronizing data between two or more services “to maintain consistency within systems” (*What is data synchronization…,* n.d.). There are many different techniques for data synchronization that are useful in different situations, but this will focus on one-way and two-way syncing.

Mirroring, or one-way syncing, is where data is pushed to your destination from your source (*One-way vs two-way synchronization…,* 2020). This type of synchronization is useful in cases where only the source data is updated and the destination just stores the information, like having a backup of your original data. GitHub communicating with Discord, for example, is an instance of one-way syncing because the data transfer does not go from Discord to GitHub when the webhook is set up on GitHub. Two-way syncing, on the other hand, pulls and pushes data in both directions. This is useful if you have two systems that both need to change a common set of data. One example of two-way syncing is synching data between a phone and a personal computer. Another example is two business tools where the data can be updated and added from either device or tool and continuously retrieved from the other (*What is a two-way sync?,* n.d.).

There are many applications that take advantage of the different types of data synchronization models. Some of the services that utilize this model are Skysync and SyncPenguin. SkySync is marketed as a way to govern unstructured data using automation and two-way syncing. It monitors changes in data from the source and notifies administration or the people listed if any data breaks the previously specified policies (*Automated unstructured data…,* 2022). This uses one-way syncing to continuously monitor data from the source and then push an alert to the designated destination. SkySync allows a user to manage their own data using their packages as a guide, but there is not much flexibility for specific customizations past the packages that they offer (*Enterprise unstructured data…*, 2022).

SyncPenguin is a cloud-based service that allows for real-time two-way syncing and integration across a large array of business tools and applications. It is a highly customizable service with a large array of supported applications and tools and the ability to customize each sync with a visual programming tool, which requires no programming experience (*How it works,* n.d.). The main attribute that distinguishes SyncPenguin from other competitors is that SyncPenguin focuses on the "smaller picture" of syncing individual services or tools, while other competitors focus more on the "bigger picture" of workplace automation. Two-way syncing is a similar concept to webhooks, but the difference is that two-way syncing does not have a specific trigger; instead, it constantly keeps the data on either end of the connection synchronized with the other end. While it is highly customizable and can be used in many cases, there are some cases where webhooks or other methods of trigger-action programming will work better with workplace automation.

Methodology

For the WaaS platform, the team allocated ample time to research the technologies the team planned on including in the designs for the database, server, and user interface. Using the information collected, the team created a plan to design and build the foundation of the WaaS API. The team also created a timeline to manage the project goal and objectives and communicate with the project advisor and sponsor.

Researching and Choosing Technologies

The group researched multiple technologies to use for the project. From the start, the team determined that NodeJS would be used for the main server of the application and that GitHub would be used for version control. These technologies were chosen due to group members already being familiar with them, allowing the team to start development early. To run the server, the team initially looked into and chose to use the program Heroku, a platform capable of hosting web servers for free. However, this technology was quickly abandoned due to a conflict where the team was unable to connect a scalable, permanent database for long term storage to Heroku with a reasonable budget. Upon the recommendation of the team's sponsor, the group looked into and chose to use Docker and Docker Compose for the final product. This allowed the team to host the server application for free, which was a major design consideration. This also allowed for the server to be hosted easily on multiple types of operating systems.

The group determined early on that the server would need a database to serve as long-term storage for the information pertaining to the connections handled by the server. The team then proceeded to research different forms of databases, including MongoDB, MySQL, and SQLite, to determine which one best suited the team's needs. After more research, the group determined that SQLite would work best for the project due to its simplicity and ease of integration with Docker, which was a major design consideration.

For the user interface design, the team decided to use React in order to develop a functional user interface that could easily be updated. This decision was made after much research into different user interface design options, including Angular and Vue. After looking into all of the different options, it was clear that React would fit the best due to the previous experience of the team members.

Database Design

The team determined that the fields the database would need were a unique id, the url of the recipient of the webhooks, any settings for said recipient, the url of the provider of the webhook, any settings for said provider, the user who created the webhook, and any special actions for the webhook. A full overview of the complete design of the database can be found in the Results section of this paper.

Server back end Design

When determining what technologies to use for the back end of the application, the group decided to use Node.js to host a server in Javascript using the Express package for common server functions. This saved time and allowed the team to focus on the database, UI, and the application's specific server logic. Using Javascript within the server also made it far easier to make the application extensible to various third-party APIs, as it is very compatible with the JSON format that most APIs use, and does not have type-checking. The team determined that the server should be hosted on a Docker container for operating system consistency, security, and overhead reduction. The group designed the database and server to be hosted on separate Docker containers linked to each other using Docker Compose so that the database could run separately from the server, while preventing access to the database for anything other than the server. The database was run as a separate program from the public-facing server to keep the data secure. The server and database were run in separate containers due to the standard of only running one program per Docker container.

User Interface Design

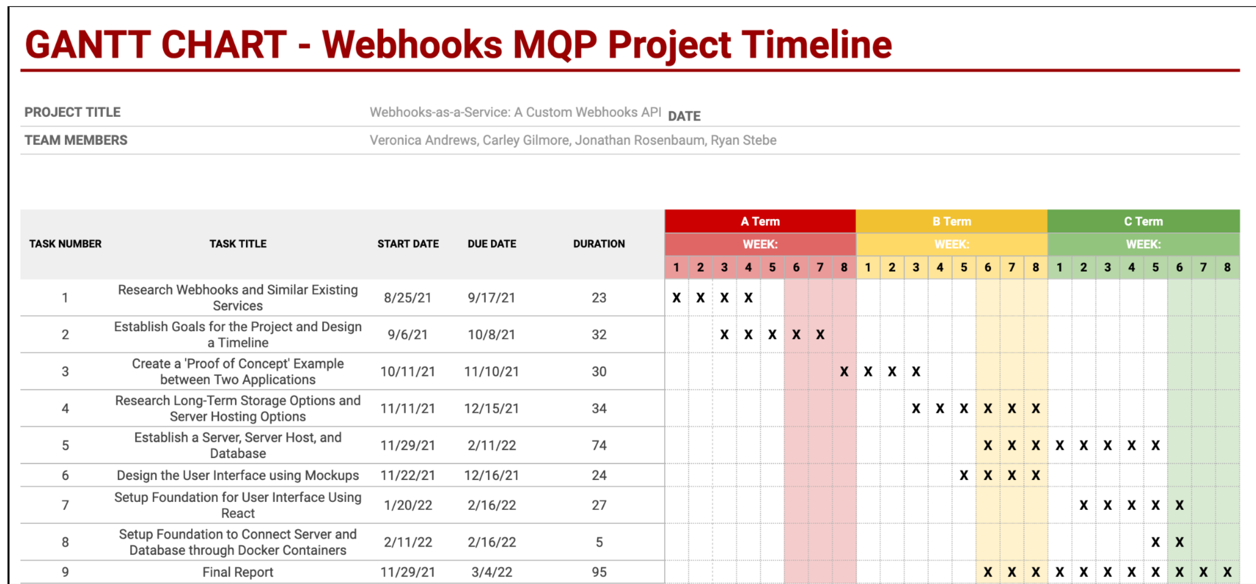
To begin the design of the user interface, the team planned to start with creating user interface mockups. After some research, the team decided to use figma.com user interface mockups, because it was familiar to most of the members of the team. The full user interface mockups are described in the Application Design section of the Results chapter, and images are included in Appendices A-E.

The team decided that after the creation of the user interface mockups, they would use React to build the user interface that would eventually be integrated with the back end of the

project. This was planned as a stretch goal for the project, because the creation of the back end was the main priority.

Timeline and Project Management

When designing the structure of the project, the team outlined a timeline that consisted of objectives and deadlines that could be updated over the course of the project to accommodate milestones and rate of work. The following figure displays the final timeline visually:



GANTT CHART - Webhooks MQP Project Timeline

PROJECT TITLE: Webhooks-as-a-Service: A Custom Webhooks API — DATE

TEAM MEMBERS: Veronica Andrews, Carley Gilmore, Jonathan Rosenbaum, Ryan Stebe

TASK NUMBER	TASK TITLE	START DATE	DUE DATE	DURATION	A Term								B Term								C Term							
					WEEK:								WEEK:								WEEK:							
					1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
1	Research Webhooks and Similar Existing Services	8/25/21	9/17/21	23	X	X	X	X																				
2	Establish Goals for the Project and Design a Timeline	9/6/21	10/8/21	32			X	X	X	X	X																	
3	Create a 'Proof of Concept' Example between Two Applications	10/11/21	11/10/21	30								X	X	X	X													
4	Research Long-Term Storage Options and Server Hosting Options	11/11/21	12/15/21	34											X	X	X	X	X	X								
5	Establish a Server, Server Host, and Database	11/29/21	2/11/22	74														X	X	X	X	X	X	X	X			
6	Design the User Interface using Mockups	11/22/21	12/16/21	24													X	X	X	X								
7	Setup Foundation for User Interface Using React	1/20/22	2/16/22	27																		X	X	X	X	X		
8	Setup Foundation to Connect Server and Database through Docker Containers	2/11/22	2/16/22	5																					X	X		
9	Final Report	11/29/21	3/4/22	95														X	X	X	X	X	X	X	X	X	X	X

Figure 1: A Gantt Chart that outlines the team objectives and timeline for the project over the course of three academic terms, not including scheduled academic breaks.

To complete the project, the team scheduled recurring weekly meetings with the project advisor, Professor Cuneo, so he could be updated on the team's progress and any issues that arose. The team also met with the sponsor, Jeremy Duvall, on an as-needed basis to ask for clarifications on the project's needs and to seek help in overcoming certain obstacles.

Results

By the end of the project's development, the team designed the structure and user interface for a WaaS application that allows users to set up automated connections between supported apps. The group additionally produced components of a partially featured prototype using the technologies that the team decided on during the design phase. The prototype was developed in its own GitHub repository under 7Factor's Github organization.

One of the team members, Carley Gilmore, has completed a Professional Writing MQP that addresses the importance of effective API documentation. The project includes a documentation set with a user guide and standard operating procedure for using the Webhooks API our team created. Future teams will be able to refer to this documentation when using and extending the API.

During the team's development of the prototype of the application, the team needed to test the team's code to ensure that it works correctly and to catch any mistakes. Because most of the application is focused on connecting different technologies together in the proper manner, the group chose to focus on manual integrated testing, since that would allow us to test the communication between the different components of the application.

Application Design

The design of the application is split into two main parts. The first is the front end client that runs on the client side in a web browser. The client allows the user to view their current connections and add, edit, and delete connections under their account. These requests are sent to the second part, the back end. The back end handles the user requests, validates them, and makes the requested changes to the database. As connections are added by users, the server will automatically set up the connection using the connected apps' APIs, pointing the webhook provider's webhook at the back end server so it can handle the incoming webhook as specified by the user.

In order to design the front end of the application prototype, the team first created user interface mockups. The design of the front end begins with a user login page and an account creation page. This design is to allow multiple users to each have their own connections. The login page can be viewed in Appendix A, and the create account page can be viewed in

Appendix B. There is also an account settings page where the user will be able to edit their specific user settings. This page is available to view in Appendix C.

The main page in the design is the dashboard page that displays all of the existing connections and allows a user to create and edit connections in an intuitive way. The dashboard page can be found in Appendix D. The dashboard page displays each connection in a way that can easily be understood without having any prior knowledge about how webhooks work. This was an important aspect because the team wanted to make the design easily accessible for any possible user. The last page, available in Appendix E, is the details page for each connection. This page can be accessed from the dashboard page. The user can either click on an existing connection to see and edit the details, or create a new connection, where each field will be blank for the user to fill in.

The back end is hosted on two separate Docker containers, with one hosting the public-facing server that handles client and third party API requests and the other hosting the database which stores and retrieves data about the various users and their established connections. When a client or third party API sends an http request, it is sent to a different routing path, where the server handles each type of request differently. Requests are made via POST requests, where the server parses the JSON body of the request and looks for the presence of certain fields, which may vary depending on the request . The server then validates and performs the request.

When the server is creating a new connection, it attempts to use the chosen webhook provider's third-party API to set up a webhook pointed at the server. If successful, whenever the third party webhook triggers, it will send a payload to the server with a url corresponding to the unique id of the connection that set up the webhook, and the server will parse the payload according to the connection's settings, and make an API call to the receiver.

A basic interaction diagram was created to model some of the interactions that could occur with the program as a whole, which the team used as a basis for the communication logic that the server handles. The diagrams below show the interactions that can be initiated by a user or by a third party webhook pointed at the server.

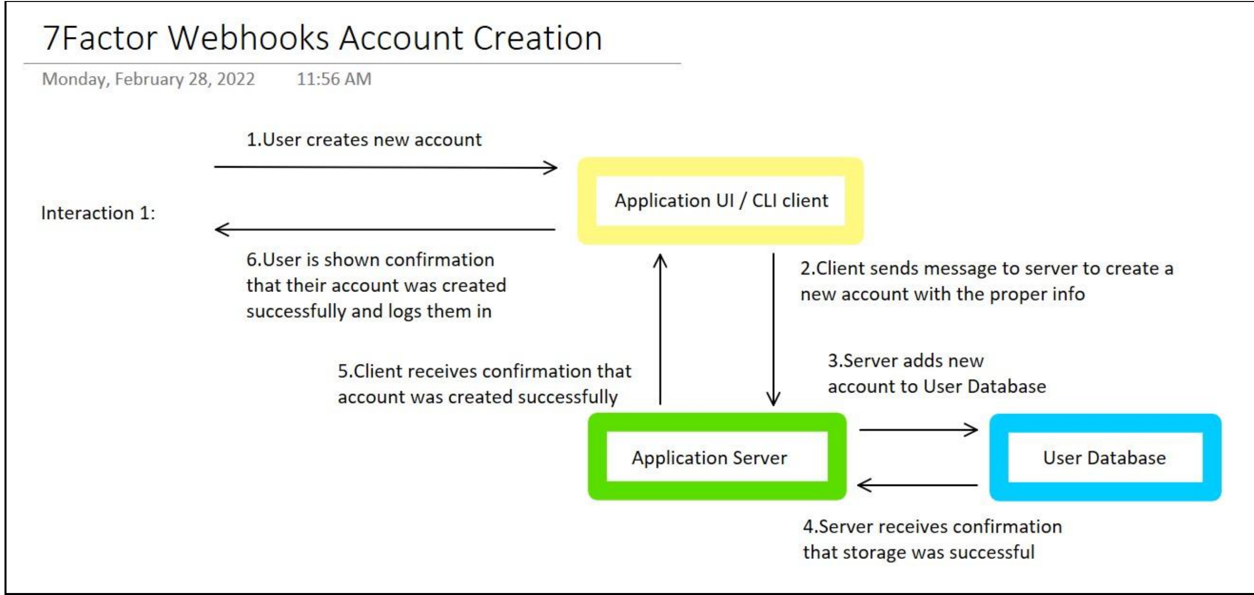


Figure 2: Diagram that details the process of a user creating an account.

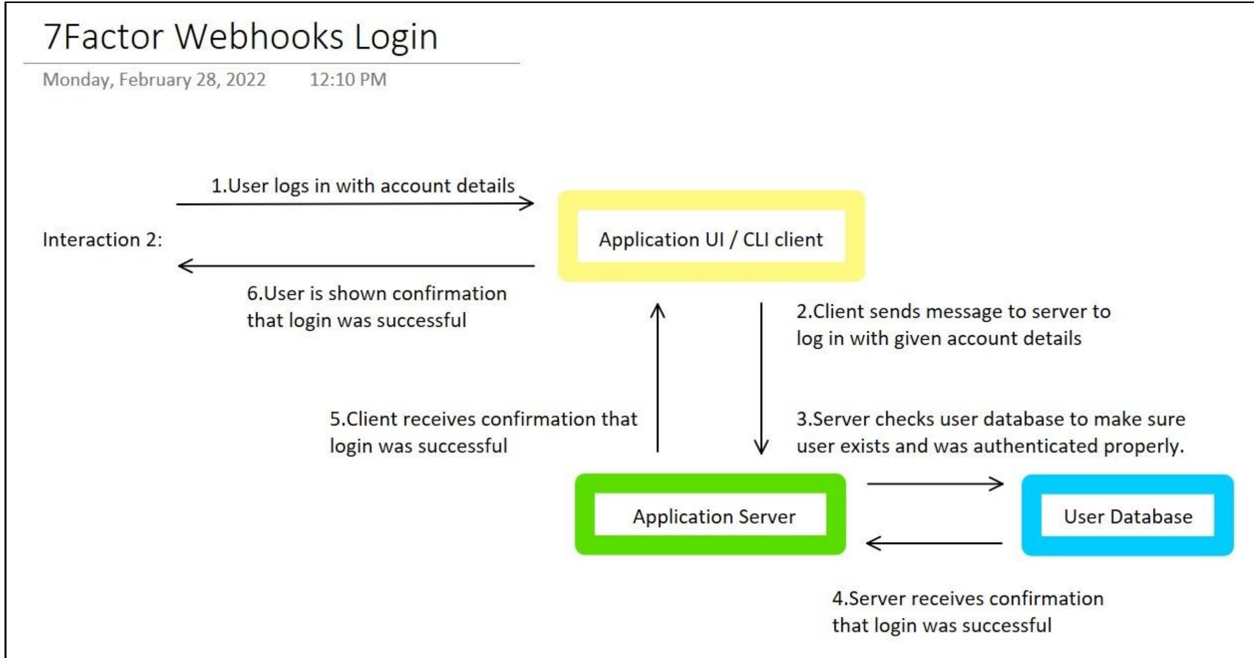


Figure 3: Diagram demonstrating the process of a user logging into their existing account.

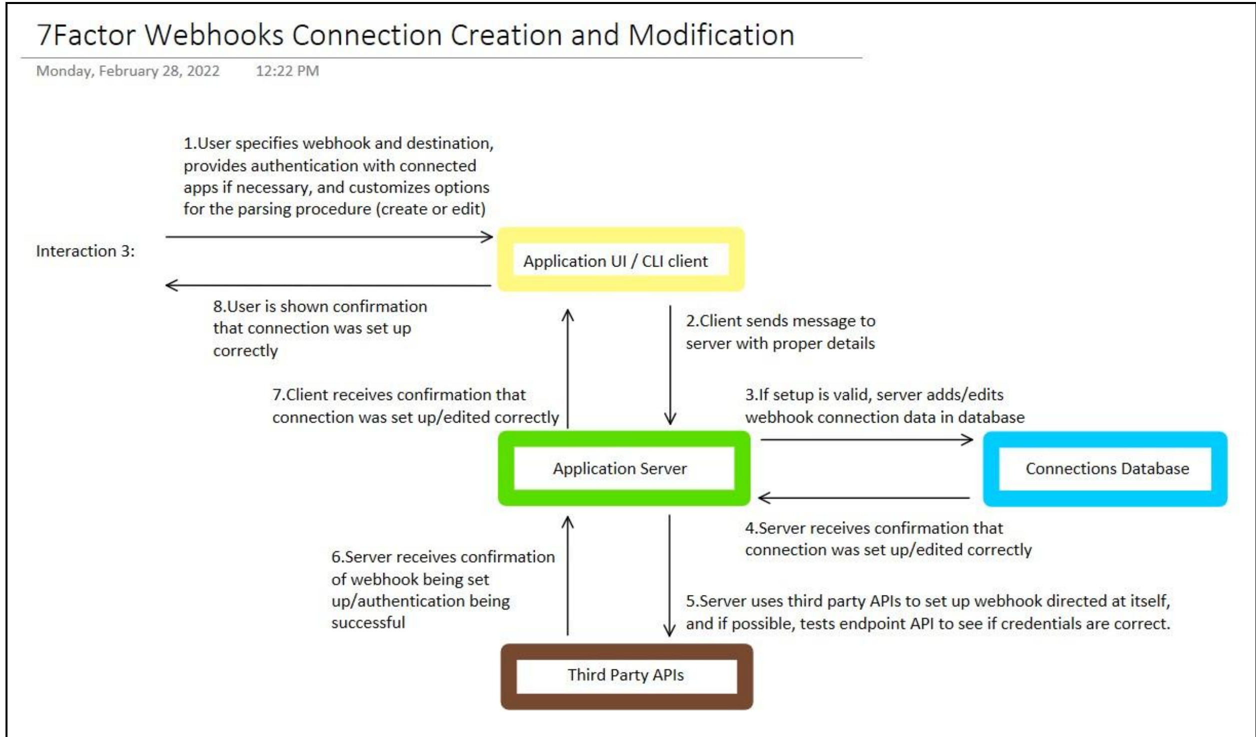


Figure 4: Diagram for the process of a user creating or editing a connection.

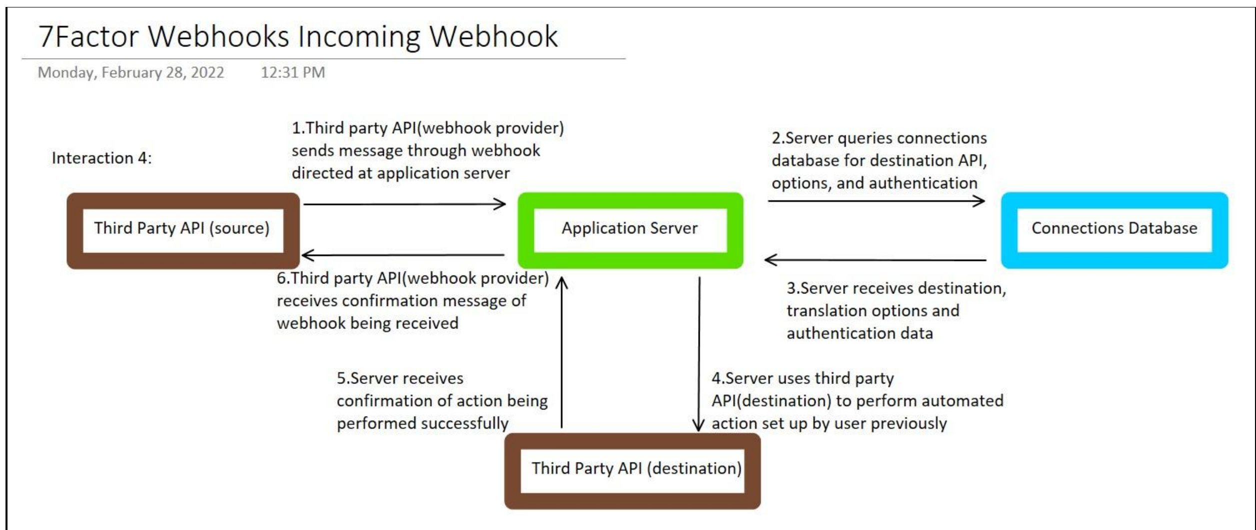


Figure 5: Diagram for the process of the server receiving a webhook from a third party API.

Prototype Features

The prototype implements many–but not all–of the features of the team's design, and it is intended to be easily extendable to implement additional features and integrate additional third-party APIs.

The technologies required for the front end have been set up and tested; however, the UI is not implemented in the prototype and does not yet call the server's API.

The back end server and database are hosted on separate Docker containers that are linked via Docker Compose. The database is hosted on its own docker and runs a Python script that receives HTTP requests from the server requesting various changes and lookups to the database. The database itself uses a SQLite database stored in the Docker container's file system. The database does not currently send a response to the server's requests, however.

The back end server is also hosted on a Docker container and uses HTTP requests to send requests to the database manager, any clients that are making requests to the server, and third-party APIs involved in a connection. The current iteration of the server does not receive responses from the Docker container containing the database. The prototype only supports GitHub as a webhook provider and Discord as a receiver.

In order to make the prototype extendable to additional applications as receivers and/or providers, the incoming webhook data is split up into parser objects for each supported application payload. These "parsers" allow each third-party API that might be integrated into the application to be parsed accordingly, and it allows any combination of provider and receiver in a connection to be parsed effectively without the programmer manually coding each possible connection.

The prototype lets users choose the action a connection's receiver should perform with the data from the provider. The prototype only supported the "send link" action, which gets a url from the provider and sends the link to the receiver. The server also uses "action parsers" to call corresponding methods in the receiver parser for performing the action, and passes the required information to them obtained from methods in the provider parser that find the required information from the payload received by the provider's API.

The server also performs validation of various kinds on each client request to make sure that the user is not creating an invalid connection, while ensuring they have access to any connections they try to edit or delete.

User Interface Technologies: React, Babel, and Webpack

In lieu of building a complete UI according to the design that is listed in Appendices A-E, the group set up a foundation of a user interface using React so that future teams may easily

build the interface and integrate it with the existing back end. The team was able to install React and do the initial integration and testing with the codebase, which also involved integrating Webpack and Babel.

In order to make rendering html elements with React much simpler, the team decided to use the .jsx format for the front end instead of plain Javascript. These .jsx files cannot be read by browsers, so Babel was used to transpile the .jsx files into plain Javascript, which ensured that the front end Javascript was accessible to as many browsers and browser versions as possible. Webpack was also used to combine all of the Javascript files included in the Babel-transpiled code into one script in order to make loading the webpage more efficient.

back end Technologies: Javascript and Node.js

The team used Javascript as the language of choice for both the front end and back end logic. Javascript can be integrated very easily with React, HTML, and CSS, which made it ideal for coding the UI and client-side logic. Node.js is a runtime environment for using Javascript for standalone applications. The group used Node.js to aid in quickly and easily creating a server that runs on Javascript. This allowed us to streamline the communication between the client and server by allowing us to use similar code in both the client and server for communication between the two. Another advantage of using Node.js and Javascript for the server was that it let us easily perform operations on objects with varying fields such as webhook payloads or dynamic connection settings set by a user. This is because Javascript checks if an object has a property or method at runtime, so the group does not need to implement a different internal settings class for each unique combination of webhook provider and receiver.

NodeJS Modules: Express and HTTPS

Express is a Node.js module that handles common features of Node.js web servers that the team used to simplify server routing and communication with clients. The group also used the Node.js module “https” for communicating with the server from the client-side, as well as communicating with the database Docker from the server.

Database Technology: SQLite

For long-term storage of the webhook created by the server, the team stored them inside a SQLite database which was managed inside another Docker container. The structure of the table is as follows:

uuid TEXT	receiver TEXT	receiverSettings JSON	provider TEXT	providerSettings JSON	user TEXT	action TEXT

Figure 6: This table shows the structure of the database table in SQLite.

The `uuid` in the table was used as a unique primary key for when the group needed to reference a specific line of the table, such as removing it. The receiver and receiverSettings columns stored the link to the program that would be receiving the webhook and its settings. The provider and providerSettings columns stored the link to the program providing the webhook and its settings. The user column stored the username of the user who created the webhook. The action column stored what action the webhook should take.

To manage the database, the team created a Python Flask webserver to handle incoming POST and GET requests for the database. The requests sent to the database are a JSON that must contain at least a command field that contains the action that the user desires to take with the database. The possible commands to send to the database and their effects are as follows:

`reset`	Resets the table to be blank. This is done by deleting and recreating the table
`add`	Adds a new line to the table. Must contain data for each field.
`get_all`	Returns a JSON containing all data within the table
`get_uuid`	Returns a JSON containing all data with the specified uuid
`get_user`	Returns a JSON containing all data with the specified user
`remove`	Removes the data from the table with

	the specified uuid
`update`	Updates the data with the specified uuid with the provided data

Figure 7: This table displays all of the implemented ways to manipulate and get information from the database table.

These commands allow for a user of the main server to access and modify all data with the database. If more commands are needed in the future, they can be easily added to the table by following the format of the previous commands.

Server Hosting Technology: Docker

To ensure the program can run smoothly on any system, the group decided to run it using Docker and Docker Compose. For the main server, the team ran it inside a Docker container which then had its port forwarded to allow it to access programs outside of the local area. The image for the server application contained an install command for NodeJS and npm, as well as setting the working directory to `/app`. This image also set the command “npm run start” to be the default command so the server application would automatically run when a container using this image was started. This allowed for a consistent environment to run the server so the group did not have to worry about different operating systems while building the project.

The database and database manager were also run inside their own container. This was done to ensure each container would only run one program. The image for the database container added the Python libraries needed to run the manager and set the working directory to `/db`. When the image is run as a container, a volume is created connecting the `/db` directory with the directory containing the database. This was done to ensure the database would be saved despite us constantly restarting the database.

To link the two containers, the group decided to use Docker Compose. The Docker Compose file contained the names of the containers, which images to use, volumes that the team created, and the ports the containers would use. By linking the two containers over Docker Compose, the server was able to send messages to the database container regardless of where they are hosted.

For step by step instructions on how to install and run this application, see Appendix F.

Recommendations

Based on the results and the prototype of the API and application that the team was able to create this year, the team has developed a set of recommendations for the next MQP team that works on the WaaS project. The recommendations are outlined as follows:

1. Complete the connection between the server and database.
2. Build and integrate a prototype user interface based on the design from this year to create a functional and complete version.
3. Host the server on a public domain so it can be accessed by anyone regardless of the server container's IP.
4. Add the ability for users to create and log in as a specific account.
5. Use OAuth with the login system for security.
6. Add unit tests to the codebase.
7. Generalize the front end to obtain from the server the miscellaneous fields the user needs to provide while creating or editing a connection.
8. Add the availability for users to save their authentication information for supported applications–such as GitHub, Concourse, and Shortcut–to their account, and use them automatically while creating connections.
9. Extend support to more webhook providers and receivers and integrate the function into the user interface.
10. Add functionality for additional actions, such as sending an email notification.

References

Automated unstructured data governance. skysync. (2022, January 20). Retrieved February 5, 2022, from https://www.skysync.com/solutions/data-governance/

Autor, D. H. (2015). Why are there still so many jobs? the history and future of workplace automation. Journal of Economic Perspectives, 29(3), 3–30. https://doi.org/10.1257/jep.29.3.3

Brackenbury, W., Deora, A., Ritchey, J., Vallee, J., He, W., Wang, G., Littman, M. L., & Ur, B. (2019). How users interpret bugs in trigger-action programming. Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. https://doi.org/10.1145/3290605.3300782

Brown, A. (2021, June 2). Automation and the future of work - how engineered systems are improving the workplace. Forbes. Retrieved December 7, 2021, from https://www.forbes.com/sites/anniebrown/2021/06/02/automation-and-the-future-of-work how-engineered-systems-are-improving-the-workplace/?sh=65de483a7372.

Chui, M., Manyika, J., & Miremadi, M. (2015). Four fundamentals of workplace automation . McKinsey Quarterly.

Enterprise unstructured data management. skysync. (2022, January 24). Retrieved February 5, 2022, from https://www.skysync.com/solutions/

Guay, M. (2020, September 12). What are webhooks? Zapier. Retrieved November 18, 2021, from https://zapier.com/blog/what-are-webhooks/.

How it works? SyncPenguin. (n.d.). Retrieved February 13, 2022, from https://syncpenguin.com/docs/

Nilsson, M., & Dunér, D. (2020). (tech.). Scalability of push and pull based event notification. Stockholm: KTH Royal Institute of Technology.

One-way vs two-way synchronization: Comparison & explanation. skysync. (2020, December 9). Retrieved February 5, 2022, from https://www.skysync.com/one-way-vs-two-way-data-sync/

Rahmati, A., Fernandes, E., Jung, J., & Prakash, A. (2017). IFTTT vs. Zapier: A Comparative Study of Trigger-Action Programming Frameworks. Arxiv.org. Retrieved from https://arxiv.org/pdf/1709.02788.pdf.

SendGrid Team. (2019, October 17). What's a Webhook? SendGrid. Retrieved November 18, 2021, from https://sendgrid.com/blog/whats-webhook/.

What are webhooks and how they work. Hookdeck. (n.d.). Retrieved November 18, 2021, from https://hookdeck.com/guides/webhooks/what-are-webhooks-how-they-work#what-is-a-webhook.

What is a two-way sync? SyncPenguin. (n.d.). Retrieved February 5, 2022, from https://syncpenguin.com/knowledge-base/what-is-a-two-way-sync/

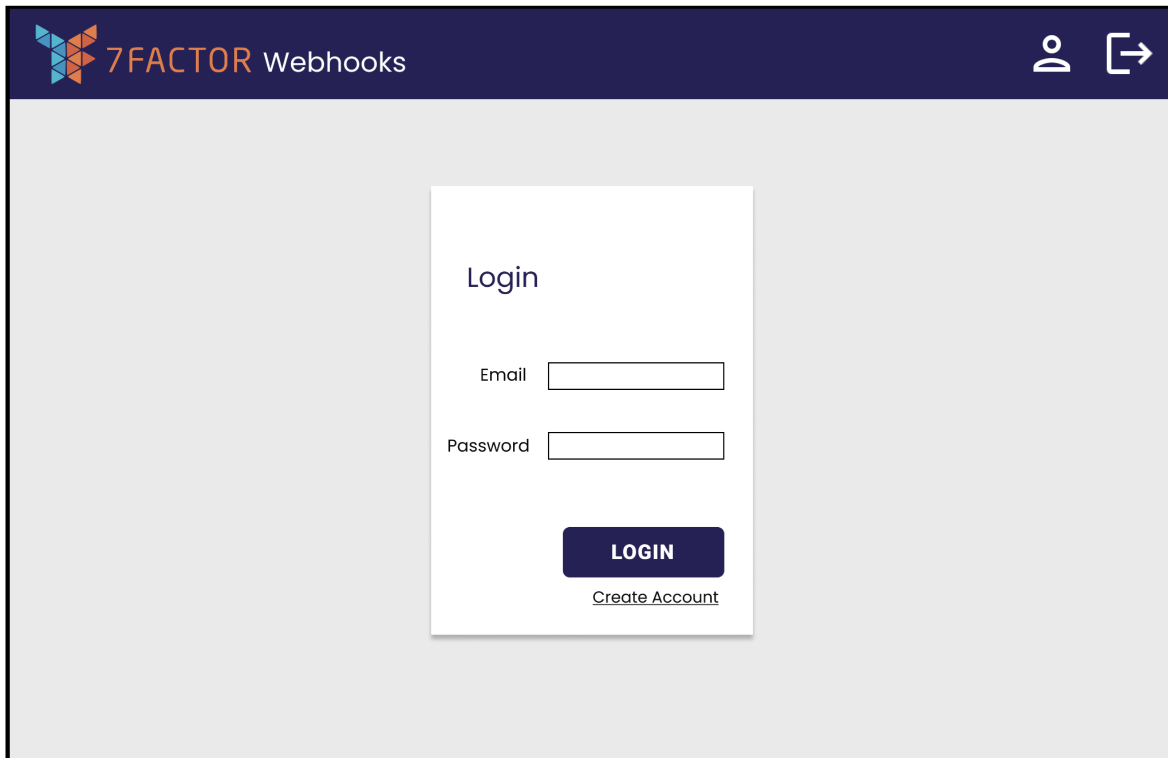
What is data synchronization and why is it important? Talend. (n.d.). Retrieved February 5, 2022, from https://www.talend.com/resources/what-is-data-synchronization/

What is the difference between PieSync and Zapier? Piesync. (n.d.). Retrieved December 7, 2021, from https://www.piesync.com/help/faq/what-s-the-difference-between-piesync-and-zapier/.

Appendices

Appendix A

The following image is the user interface mockup for the user login page of the application.



Appendix B

The following image is the user interface mockup for the application page that will allow a user to create an account.

7FACTOR Webhooks

Create Account

Email

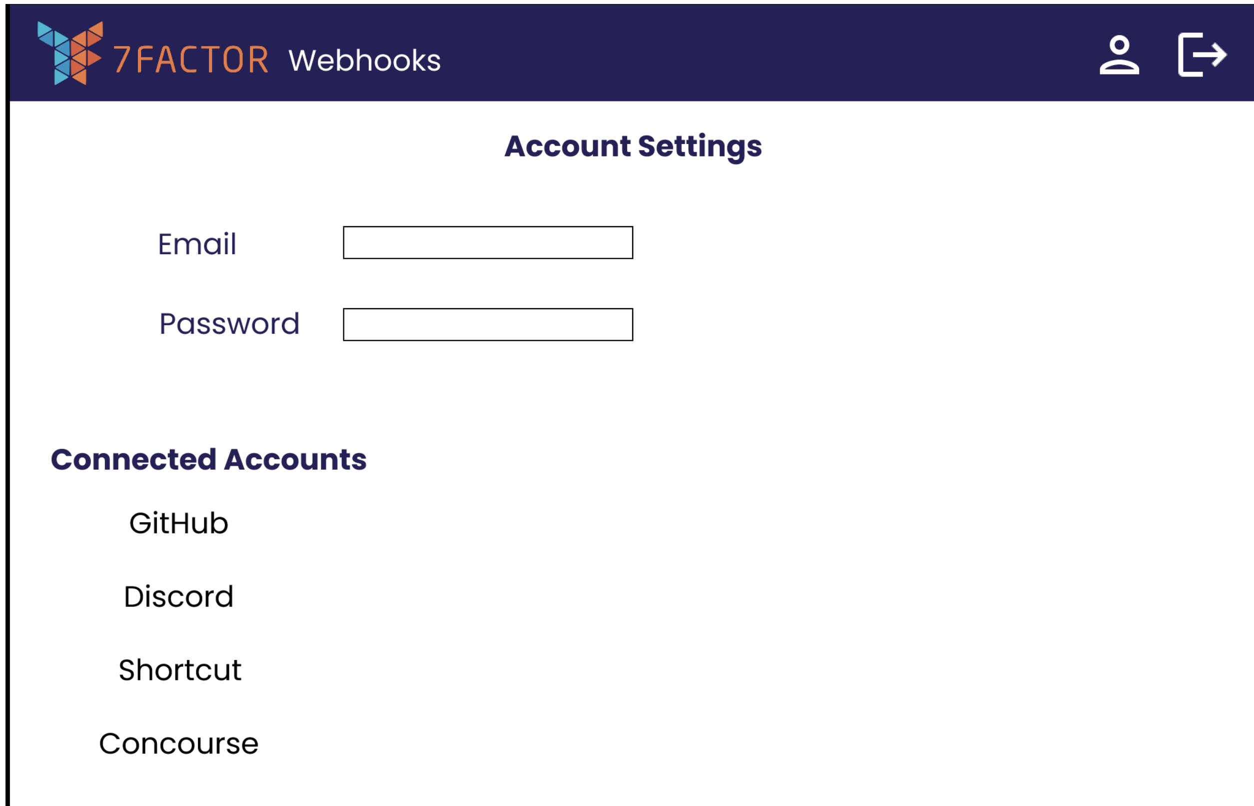
Password

Confirm Password

CREATE

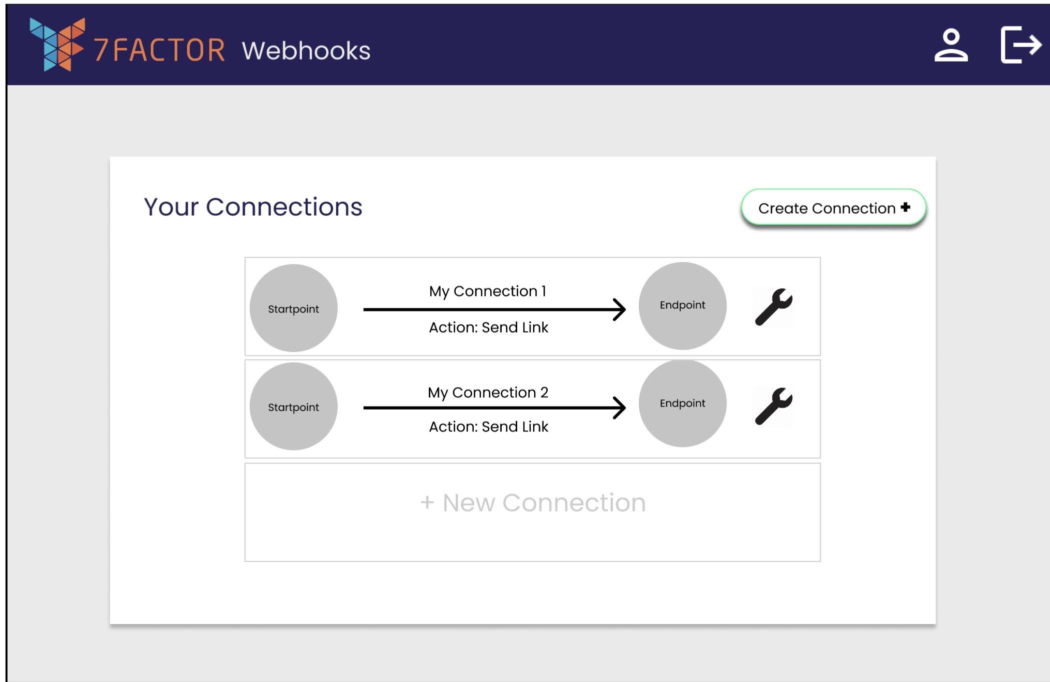
Appendix C

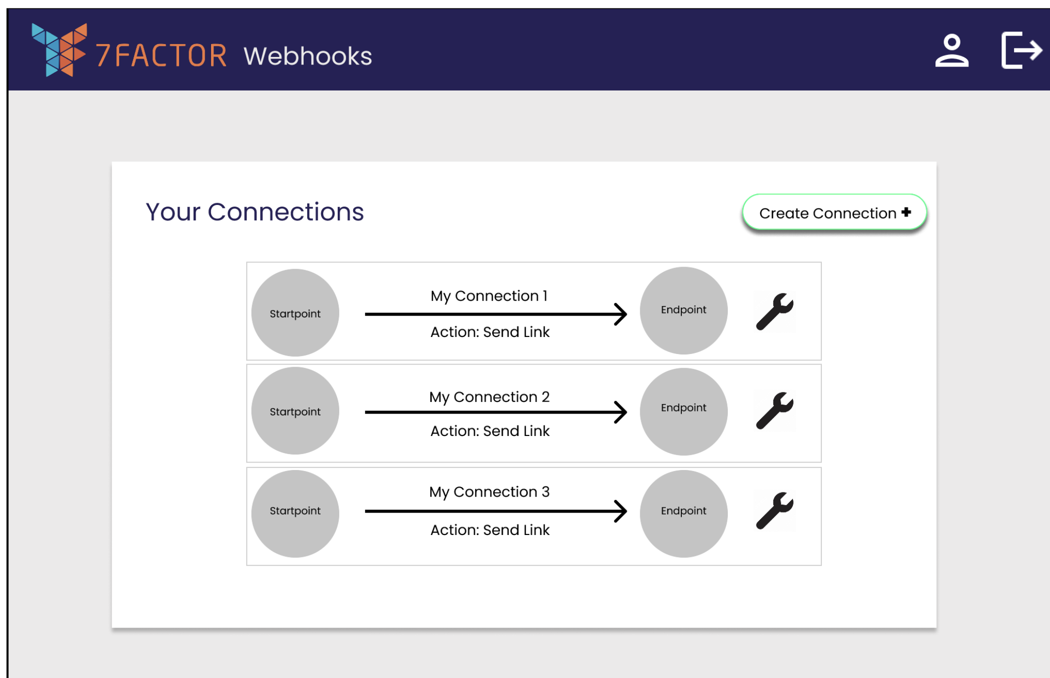
The image below is the user interface mockup for the specific user settings based on the account that is logged in.



Appendix D

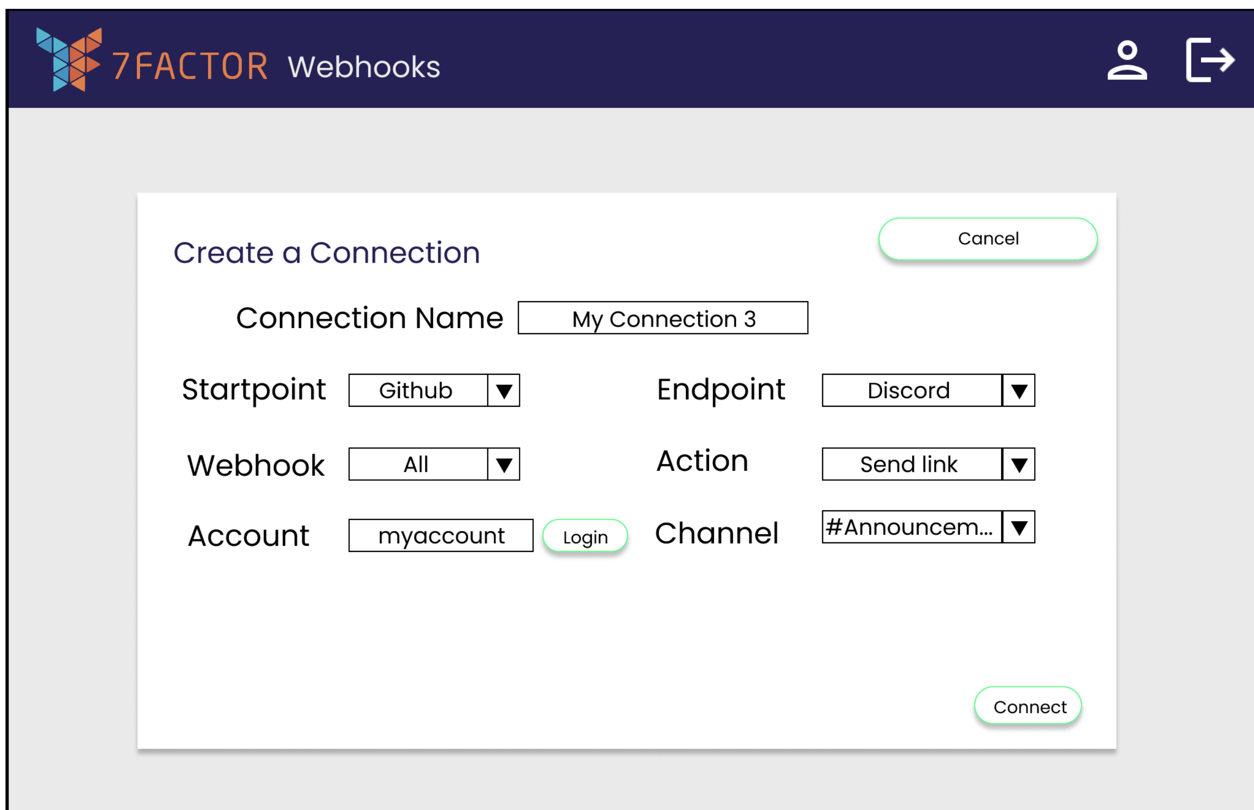
The images below are a mockup of the main dashboard of the application that will display all of a user's connections and allow a user to move onto the details page to create a new connection. The second image is a representation of what the dashboard will look like once a user adds another connection.





Appendix E

The following image is the detail page of a connection. The user can access this page from the dashboard by either clicking an existing connection or clicking the ‘create connection’ button. If an existing connection is clicked, then this page will be filled with the existing details of that connection. If ‘create connection’ is pressed, then the fields on this page will be blank by default.



Appendix F

To install and run the application, follow these steps:

1. Install the codebase from github.
2. Install Docker, follow these steps from Docker's website: https://docs.docker.com/get-docker/
3. Once Docker is installed and running correctly, navigate to the installed project `~/webhooks`
4. Run the following two commands to build the two required images:
```
docker build -t jlrosenbaum/webhooks_mqp .
docker build -t jlrosenbaum/webhooks_db Docker/Database
```
5. Run the following command to run the two containers through Docker compose:
```
docker compose up
```
6. Once both containers are running, the application can be accessed on `localhost:666`