7Factor Webhooks-as-a-Service: Standard Operating Procedure

Contents

Chapter 1. Overview **3**

Recommendations Overview 3

Chapter 2. Getting Started **4**

Installing and Running the Application 4

Chapter 3. Backend Design **6**

Third Party Providers and Receivers 6

Chapter 4. UI Design **7**

Creating ReactJS Components 10

Rendering HTML Elements 11

Chapter 1. Overview

This standard operating procedure is intended for future computer science MQP teams working with 7Factor on their Webhooks-as-a-Service (WaaS) API that was started in a 2021-2022 Major Qualifying Project (MQP). The tasks that will be covered include procedures recommended by the prior team to continue building the WaaS API. The recommendations are outlined in the *Webhooks-as-a-Service: A Custom API Design report*. This SOP shows WPI student developers how to get started working with the current codebase and materials. It introduces tasks that are to be worked on during the course of their MQP, an extension of the WaaS API.

Recommendations Overview

The following recommendations of the report are covered in this SOP. Each recommendation will include methods that the team suggests for completing each task.

The procedures on how to build off of the existing API framework in place are:

1. Installing and Running the Application
2. Integrating the current UI design with the React.js framework
3. Extending support to more webhook providers and receivers

Chapter 2. Getting Started

To get started with extending the custom WaaS API it is important to get familiar with the most recent project concepts and codebase. You will gain access to the codebase and contribute via a 7Factor provided GitHub Organization.

The codebase contains server and database components for a future working application. Your team should work to complete the connection between the server and database. The current server and database are supported through Docker containers. Using the existing codebase, the past team was able to do manual integration testing with webhooks between Discord and Github. For example, the action of sending a message in Discord when an action occurs in Github is established as a webhook. You will be encouraged to extend the application to other third party APIs.

Using the React modules that have been setup, you will also be expected to work on a user interface based on the standards set in the project mockups.

Installing and Running the Application

You will learn how to install, build, and run the server and database application with Docker.

To install, build, and run the application with the current codebase:

1. Install the codebase from GitHub.
2. Install Docker.
3. Navigate to the installed project `~/webhooks`.
4. Run the following 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 database and server containers through Docker compose:

```
docker compose up
```

6. Access the application on `localhost:666`.
    a. If `localhost:666` does not work on your machine, try looking in the terminal print outs for where the containers are hosted.

In this example, the database appears to be running on port 8000.

```
carleygilmore@autoreg-166496:~/Desktop/webhooks$ docker compose up
[+] Running 3/3
 ⠿ Network webhooks_webhooksnet   Created
 ⠿ Container webhooks-database-1  Created
 ⠿ Container webhooks-server-1    Created
Attaching to webhooks-database-1, webhooks-server-1
webhooks-database-1  |  * Serving Flask app 'database_manager' (lazy loading)
webhooks-database-1  |  * Environment: production
webhooks-database-1  |    WARNING: This is a development server. Do not use it in a production deployment.
webhooks-database-1  |    Use a production WSGI server instead.
webhooks-database-1  |  * Debug mode: on
webhooks-database-1  |  * Running on all addresses (0.0.0.0)
webhooks-database-1  |    WARNING: This is a development server. Do not use it in a production deployment.
webhooks-database-1  |  * Running on http://127.0.0.1:8000
webhooks-database-1  |  * Running on http://172.18.0.2:8000 (Press CTRL+C to quit)
webhooks-database-1  |  * Restarting with stat
webhooks-database-1  |  * Debugger is active!
webhooks-database-1  |  * Debugger PIN: 128-403-806
```

Chapter 3. Backend Design

The backend design consists of a NodeJS server and a SQLite database. Both the server and the database are hosted in Docker containers.

Important prequisite knowledge for working on the backend includes:

- Getting familiar with the current codebase
- Referencing the 7Factor User Guide
- Basic knowledge of JavaScript, Python, and SQLite

Some tasks expected to be completed by the team include:

- Connecting the server and database containers
- Creating a secure login system with multiple users
- Adding functionality for third party providers and receivers (This task will be covered in the next section on *Third Party Providers and Receivers*)

Third Party Providers and Receivers

One of the recommendations provided by the team is to add third party providers and receivers to the API.

To add third party providers and receivers, please refer to the `7Factor Webhooks-as-a-Service User Guide` as you complete this task.

1. Navigate to `server/payload_parsers`.
2. Create a new .js file for the service. For example: `concourse.js`.
3. Research the format of the service's payload you will be sending and receiving.
4. Create methods for handling actions similar to `sendLink`.
5. Require the parser and add a case in `getParser()`in`parser-dict.js` .
6. Consult the User Guide for more information.

Chapter 4. UI Design

The current UI design was mocked up using Figma, a cloud based UI tool for drafting mockups. The foundation for creating UI Designs has been integrated into the current codebase using React, Babel, and Webpack.

Important prequisite knowledge for working on the UI includes:

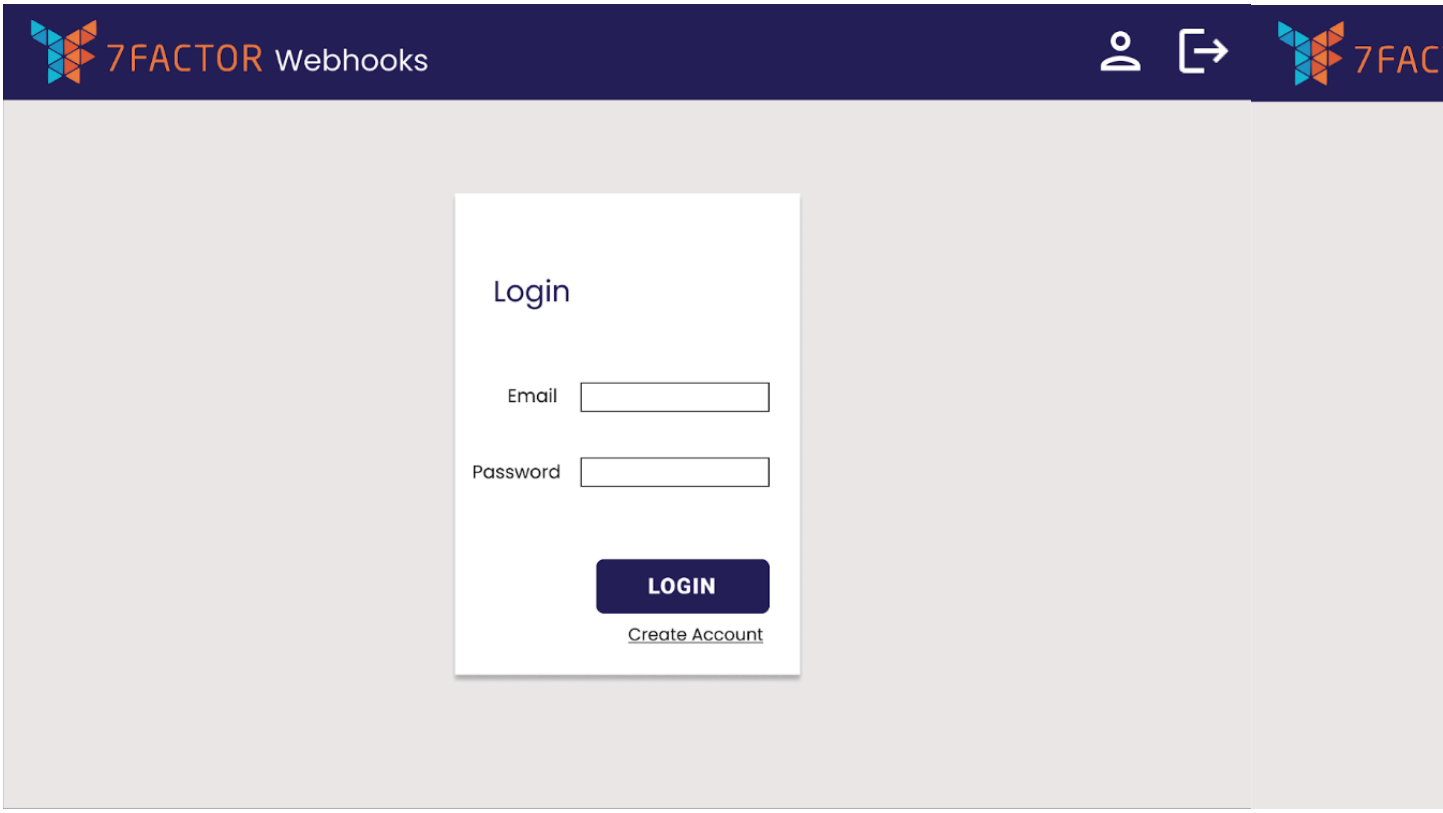
- Getting familiar with the current codebase
- Referencing the 7Factor User Guide
- Strong knowledge of JavaScript and React

Current UI Mockups

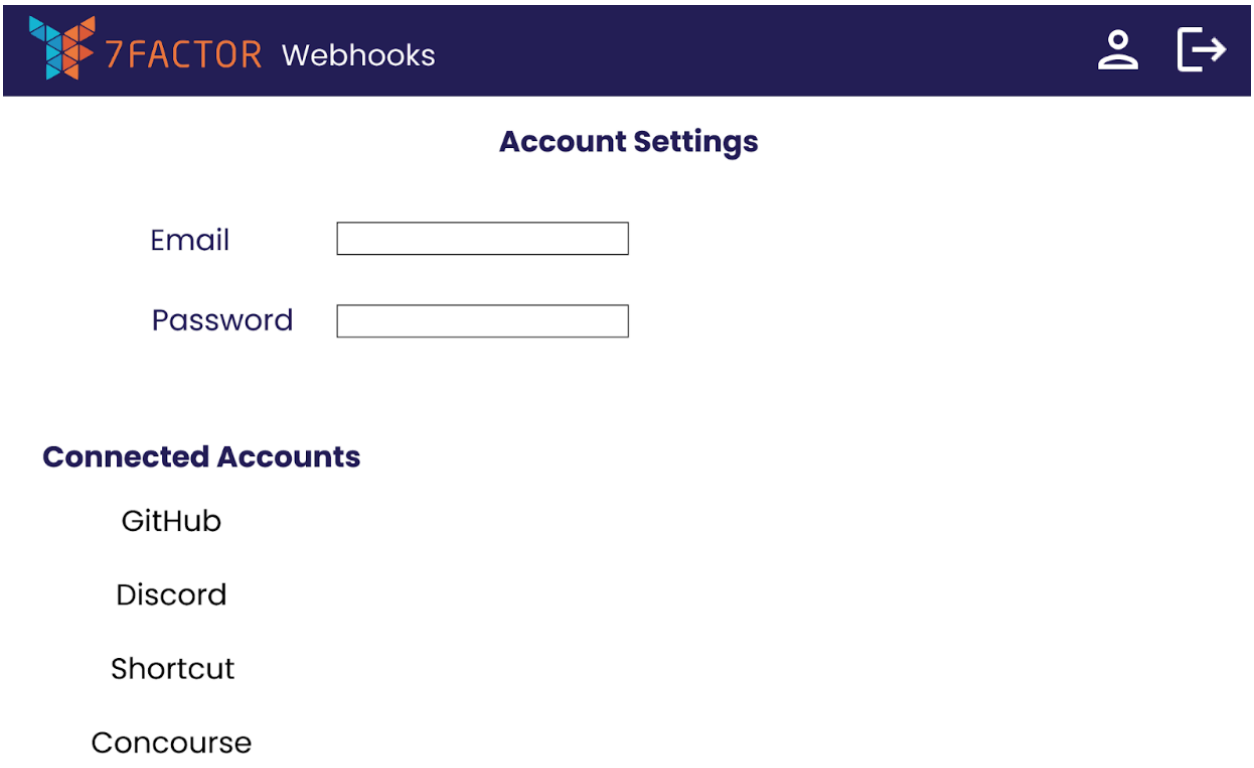
The following pictures represent concepts that should be implemented by future teams using React. You may reference these mockups as a guide before creating your own ReactJS components and rendering HTML elements explained in later topics.

Login Pages

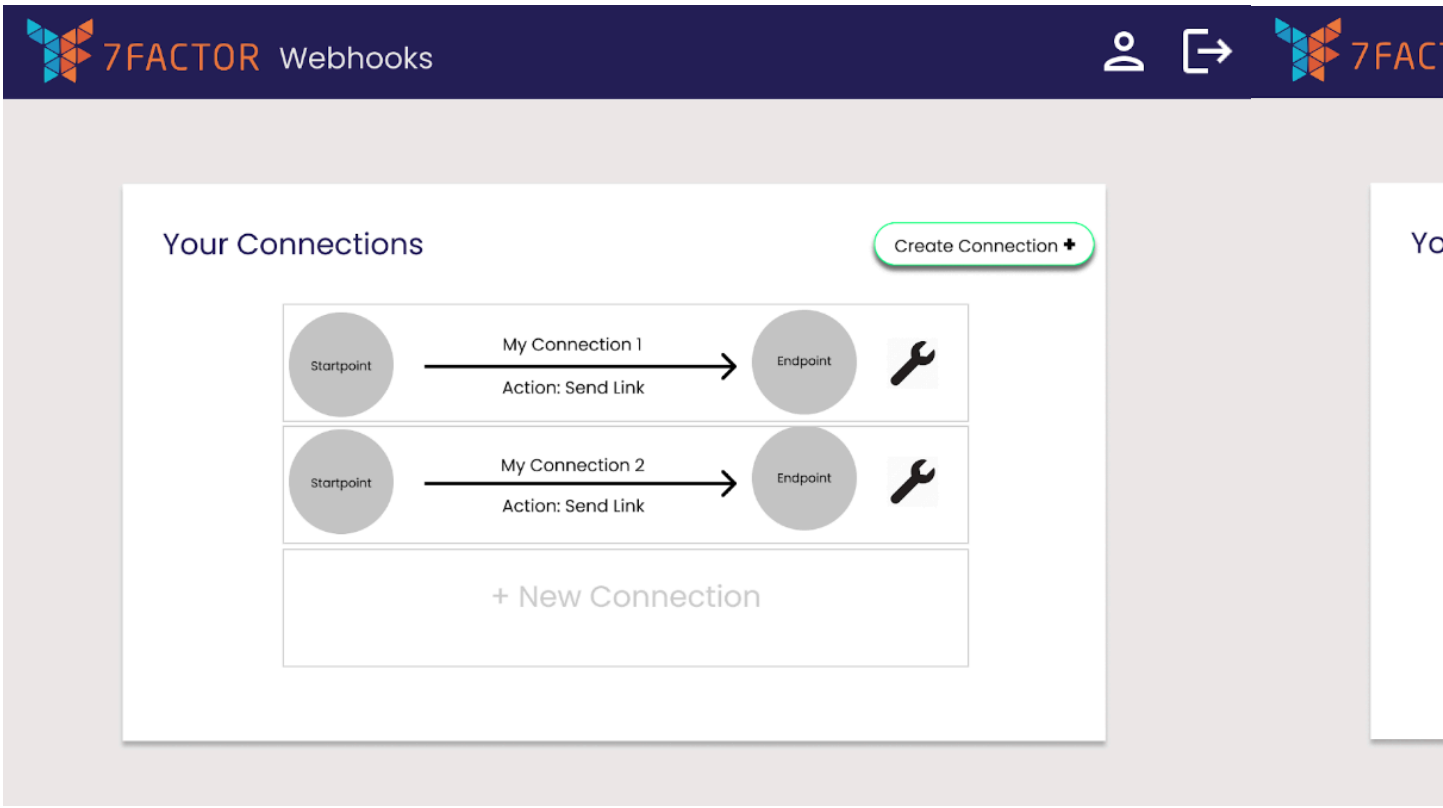
The following pages are for the login and create account screens.



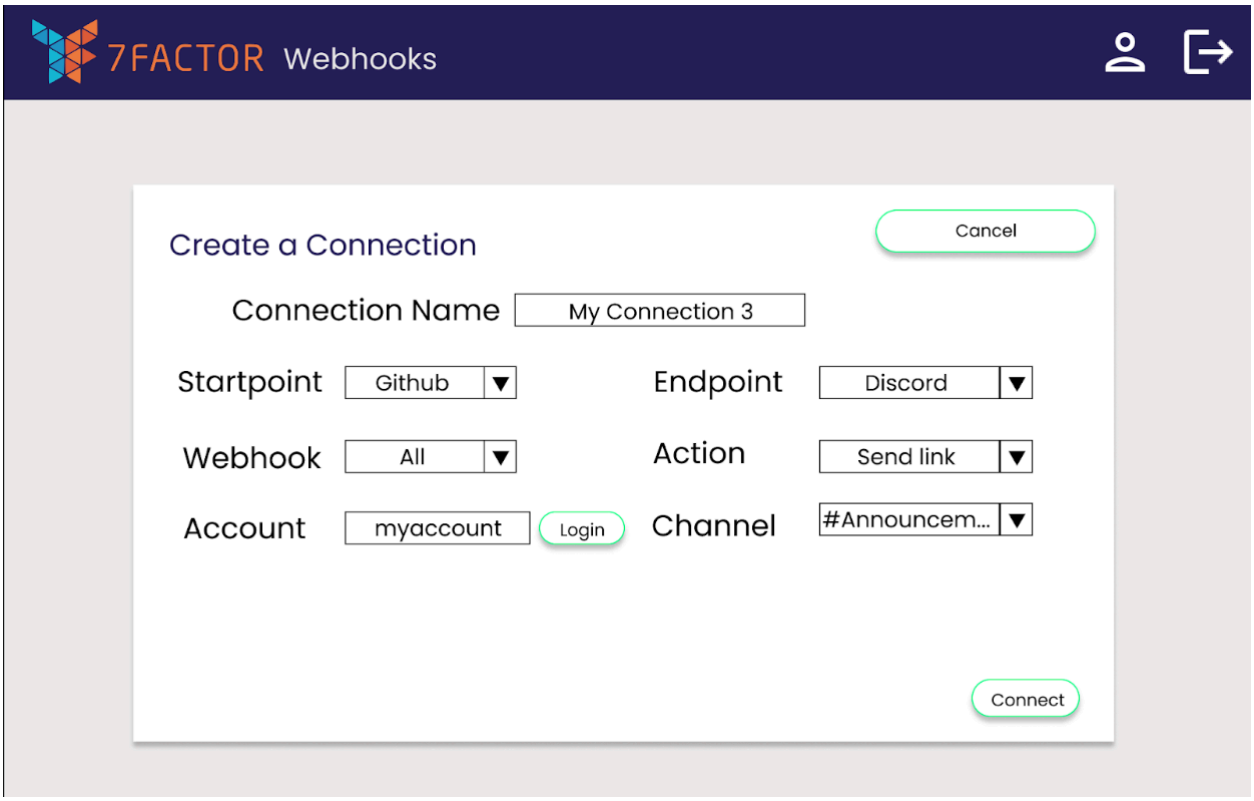
Settings Page



Connections Dashboard



Connection Creation Page



Creating ReactJS Components

You will learn how to create ReactJS components in the application that will be rendered in a webpage.

To create a ReactJS component:

1. Under the `src` folder, right click the folder and create a new .jsx file.

   The resulting file is to be written in plain Javascript

2. Import React with the following code:

```
import React from "react";

import ReactDOM from "react-dom";
```

3. Create a JS class that extends React.Component similar to the example below.

```
class LikeButton extends React.Component {

constructor(props) {

super(props);
```

```
        this.state = { liked: false };
    }
```

4. Create a `render()` method where the structure of the component will be returned.

```
render() {
if (this.state.liked) {
return 'You liked this.' + this.props.value;
    }

return (
<button onClick={() => this.setState({ liked: true })}>
            Like
</button>
    );
}
```

5. Outside of the class, create a DOM container and call render on it with the class created.

```
const domContainer = document.getElementById('react-container');
ReactDOM.render(<LikeButton/>, domContainer);
```

6. Refer to "Rendering HTML Elements" for properly rendering the element in the DOM container.

Rendering HTML Elements

Once you have created a React js file, you should use the Babel and Webpack setup commands for rendering. To make rendering HTML elements in React simpler, Babel is used for transpiling the .jsx files into plain Javascript and Webpack is used to combine all of the Javascript files into one script for efficiency.

These steps will show you what is needed to render React webpages:

1. In the `node_modules`folder, check if Babel and Webpack packages have been installed. Use the following commands to check and/or make updates:
    a. For Babel: `npm install babel`
    b. For Webpack: `npm install webpack`
2. In a terminal window, start Babel for transpiling with the following commands:
    a. For Windows: `npm run start-babel-windows`
    b. For Mac/Linux: `npm run start-babel`
3. Save any changes made to .jsx files.

> **Note:**
> Make sure that after saving changes to .jsx files to check if Babel was stopped. If so, re-run the appropriate Babel command above.

4. In another terminal window, run the following command: `npm run webpack`
5. For Mac/Linux run the following command in a third terminal window: `npm start`

   You should notice .js files of the same name appear under `public/build`

6. Test the React component in a browser

> **Note:**
> It is important to keep these watcher commands running in separate terminals, as they transpile the code while edits are saved.