7Factor Webhooks-as-a-Service: User Guide

Contents

Chapter 1. Overview

This user guide outlines the key features implemented by 7Factor's Webhooks-as-a-Service (WaaS) API, started in a 2021-2022 Major Qualifying Project (MQP). A summary of how the custom API was designed is outlined in the *Webhooks-as-a-Service: A Custom API Design* report. The user guide serves to introduce API features to future teams as they work on their MQPs as extensions of the WaaS API. The guide covers the basic functionality and set up of the API's database and server, which handle webhook payloads, connection actions, and connection requests.

Chapter 2. Getting Started

The WaaS API is intended to allow for a customized automation of tasks among third party applications. This guide is meant to walk through the code processes of creating webhooks, storing webhooks, and relevant information in connections. The current version of the API contains a working server and database for managing and storing webhook data. At the end of this guide, you will have an understanding of the framework of the application and its logic.

Chapter 3. Database Structure

For long-term storage of the webhooks created by the server, the webhook is stored inside of a SQLite database.

The structure of the table is as follows:

Table 1. Database Table for Webhook Storage

`uuid`	`receiver`	`receiver-Settings`	`provider`	`provider-Settings`	`user`	`action`
TEXT	TEXT	JSON	TEXT	JSON	TEXT	TEXT

- `uuid`: a unique primary key for referencing a specific line of the table
- `receiver` and `receiverSettings`: store the link to the program receiving the webhook and its settings
- `provider` and `providerSettings` store the link to the program providing the webhook and its settings
- `user` represents the user who created the webhook
- `action` stores what action the webhook should take

Database Management

The database has a Python script called `database_manager.py`

The script utilizes Flask, a Python framework useful for database management. The methods currently implemented include:

- `def reset_table(connection):`
- `def add_data(connection, rec, recSet, prov, provSet, user, action):`
- `def get_data(connection):`
- `def get_data_by_uuid(connection, uid):`
- `def get_data_by_user(connection, user):`
- `def remove_connection(connection, uid):`
- `def update_data(connection, better_jason, uid):`

These methods execute SQLite commands which can be found in the codebase for creating, removing, resetting, and updating data in the database.

Chapter 4. Server Structure

This section describes the implementation of the current NodeJS server.

The server is hosted in the `app.js` file located in the server folder. It utilizes connection actions, payload parsers, a webhook payload parser, and a connection request handler, which are described in later topics

The following code sample show the progression of methods that build the infrastructure of the server, handling webhooks and payloads

Creating an Express app for the Server

```
// create express app, initialize miscellaneous middleware
const app = express();
app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(function(req, res, next) {
console.log(req.body);
    next();
});
app.use(cookieParser());
```

Sending JS, HTML, and CSS to Browser

```
// send frontend javascript, html, css to browser when requested
app.use(express.static(path.join(__dirname, '../public')));
```

Handling Frontend Request

```
// handle connection creation request from frontend
app.post('/connections/add', (req, res) => {
    res.json(connectionRequestHandler.add(req.body));
});


// handle connection edit request from frontend
app.post('/connections/edit', (req, res) => {
    res.json(connectionRequestHandler.edit(req.body));
});
```

```
// handle connection delete request from frontend
app.post('/connections/remove', (req, res) => {
    res.json(connectionRequestHandler.remove(req.body));
});

// handle get user's connections request from frontend
app.post('/connections/update', (req, res) => {
    res.json(connectionRequestHandler.update(req.body));
})
```

Handle Received Webhook

```
app.post('/webhooks/:id', (req, res) => {
webhookPayloadHandler(req.body, req.params.id);
});
```

Error Catching/Handling

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
    next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
// set locals, only providing error in development
    res.locals.message = err.message;
    res.locals.error = req.app.get('env') === 'development' ? err : {};

    // render the error page
    res.status(err.status || 500);
    res.json({
message: err.message,
        error: err
    });
});
```

Connection Actions

Connection actions represent the tasks that automate among connected third-party APIs.

There is a folder named `connection_actions`, which contains Javascript files for specific connection actions.

An example of a connection action is the `send-link.js` method shown below:

```
const sendLinkActionHandler = {
// all actions must implement this function
    apply: function(payload, provider, receiver, providerSettings, receiverSettings){
return {data, options} = receiver.sendLink(receiverSettings, provider.getURL(providerSettings,
 payload));
    },

    // all actions must implement this function
    isValidProvider: function(provider){
return (typeof provider.isValidProviderSettings === 'function' &&
typeof provider.getURL === 'function');
    },

    // all actions must implement this function
    isValidReceiver: function(receiver){
return (typeof receiver.isValidReceiverSettings === 'function' &&
typeof receiver.sendLink === 'function');
    }
}

module.exports = sendLinkActionHandler;
```

Note:
See Payload Parsers for code samples which aid in the integration of webhook actions for the third party APIs, Discord and GitHub.

Payload Parsers

The current payload parsers handled by the WaaS API are for Discord and GitHub. The parsers are used in handling connections, getting information on which API is the receiver and which is the provider to manage webhooks accordingly.

The following code snippets represent the establishment of creating webhook data with specific options configured as receiver settings. It can be found in the `discord.js` file under `payload_parsers`.

```
const discordParser = {


    sendLink: function(receiverSettings, url) {
        data = JSON.stringify({
            username: 'Gompei',
            content: 'Hooked on a feeling',
            embeds: [{"title": "Test", "url": url}]
        });
```

First, the parser is initialized as a `const` value. `sendLink` represents the action. The JSON data is then set with properties specific to Discord's JSON payload formatting. The `username`, `content`, and `embeds` values can be initialized here and changed to any value depending on user input.

```
options = {
            hostname: 'discordapp.com',
            path:
 '/api/webhooks/902593973997146135/5WW8GsSejvLam1o1JpF_wu5EIdyT0Dqx24-2h0kqtexVkwWjYMxfwM1K1AMZ0RGfEMLw',
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                'Content-Length': data.length
            }
        };
```

Next, the options for the data is set with the appropriate path. This path is specific for webhooks established with the Discord API.

```
return {data, options};
    },
```

The parser returns the set data and options, and connects the information to the receiverSettings, making Discord the receiver of a payload from Github.

Webhook Payload Handler

The webhook payload handler pulls a webhook from the database and the information associated with its provider and receiver parsers.

The code sample below shows how the webhook data is retrieved from the database and the receiver and provider values are established:

```
const https = require('https');

const getAction = require('./connection_actions/action-dict');

const getParser = require('./payload_parsers/parser-dict');

const validator = require('./connection-validator');

const database = require('./database-wrapper');


function handleWebhookPayload(payload, id){

    // id is from url payload was sent to


    // get webhook from database

    let webhook = database.getConnection(id);


    // get action, providerParser, and receiverParser

    let action = getAction(webhook.action);

    let provider = getParser(webhook.prov);

    let receiver = getParser(webhook.rec);

    let providerSettings = webhook.provSet;

    let receiverSettings = webhook.recSet;


    // check if connection is valid

    if (validator(action, provider, receiver, providerSettings, receiverSettings)) {

        const {data, options} = action.apply(payload, provider, receiver, providerSettings, receiverSettings);

        sendPayload(data, options);

    }

}
```

sendPayload

`sendPayload` is a method that sends a request to send the JSON payload

Parameters

data: stringified JSON with data to be sent in the payload

options: JSON of with information on the host and path to send webhook

```
function sendPayload(data, options) {

    const req = https.request(options, res => {

        console.log(`statusCode: ${res.statusCode}`);


        res.on('data', d => {

            process.stdout.write(d)

        });

    })


    req.on('error', error => {

        console.error(error)

    });


    req.write(data);

}
```

Connection Request Handler

The connection request handler contains authentication and specific information on the provider and receiver for establishing a connection

The code samples below shows how the connection request handler retrieves values from the request and adds, edits, or removes them via the database which stores these connections.

add

The `add` method grabs the action, parser, and validator for a payload. It sets the user, action, provider, providerSettings, receiver, and receiverSettings which are the attributes of a webhook connection. It will only add a connection if the validator returns `true` as shown below.

```
if (validation === true) {

            // put variables above into database as a connection

            database.addConnection(user, action, provider, providerSettings, receiver, receiverSettings);

            let mesg = "Added " + action + " connection as user: " + user + " between " +

            provider + " and " + receiver + ". Provider options: " + providerSettings +
```

```
            ", Receiver options: " + receiverSettings;
            console.log(mesg);
            return { message: mesg };
        }
```

edit

The `edit` method grabs information in the same way as *add*. It checks if the user is the same user that owns the connection that it is to be edited, and sets values accordingly.

```
database.editConnection(id, action, provider, providerSettings, receiver, receiverSettings);
        let mesg = "Edited connection " + id + " (" + action + ") as user: " + user +
        " between " + provider + " and " + receiver + ". Provider options: " + providerSettings
        + ", Receiver options: " + receiverSettings;
        console.log(mesg);
        return { message: mesg };
```

remove

The `remove` method does the same validation and ownership check as stated above. It makes a call to `database.removeConnection`.

update

The `update` method calls the `database.updateConnections`.