Handling webhook deliveries

Learn how to write code to listen for and respond to webhook deliveries.

In this article

Introduction

When you create a webhook, you specify a URL and subscribe to event types. When an event that your webhook is subscribed to occurs, GitHub will send an HTTP request with data about the event to the URL that you specified. If your server is set up to listen for webhook deliveries at that URL, it can take action when it receives one.

This article describes how to write code to let your server listen for and respond to webhook deliveries. You will test your code by using your computer or codespace as a local server.

Setup

In order to test your webhook locally, you can use a webhook proxy URL to forward webhooks from GitHub to your computer or codespace. This article uses smee.io to provide a webhook proxy URL and forward webhooks.

Get a webhook proxy URL

1. In your browser, navigate to https://smee.io/.
2. Click **Start a new channel**.
3. Copy the full URL under "Webhook Proxy URL". You will use this URL in the following setup steps.

Forward webhooks

1. If you don't already have smee-client installed, run the following command in your terminal:

```shell
```

```
npm install --global smee-client
```

2. To receive forwarded webhooks from smee.io, run the following command in your terminal. Replace `WEBHOOK_PROXY_URL` with your webhook proxy URL from earlier.

   Shell

   ```
   smee --url WEBHOOK_PROXY_URL --path /webhook --port 3000
   ```

   You should see output that looks like this, where `WEBHOOK_PROXY_URL` is your webhook proxy URL:

   Shell

   ```
   Forwarding WEBHOOK_PROXY_URL to http://127.0.0.1:3000/webhook
   Connected WEBHOOK_PROXY_URL
   ```

   Note that the path is `/webhook` and the port is `3000`. You will use these values later when you write code to handle webhook deliveries.

3. Keep this running while you test out your webhook. When you want to stop forwarding webhooks, enter `Ctrl` + `C`.

Create a webhook

1. Create a webhook with the following settings. For more information, see "Creating webhooks."
   - For the URL, use your webhook proxy URL from earlier.
   - If you have an option to choose the content type, use JSON.

Write code to handle webhook deliveries

In order to handle webhook deliveries, you need to write code that will:

- Initialize your server to listen for requests to your webhook URL
- Read the HTTP headers and body from the request
- Take the desired action in response to the request

You can use any programming language that you can run on your server.

The following examples print a message when a webhook delivery is received. However, you can modify the code to take another action, such as making a request to the GitHub API or sending a Slack message.

- Ruby example
- JavaScript example

Ruby example

This example uses the Ruby gem, Sinatra, to define routes and handle HTTP requests. For more information, see the Sinatra README.

Ruby example: Install dependencies

To use this example, you must install the sinatra gem in your Ruby project. For example, you can do this with Bundler:

1. If you don't already have Bundler installed, run the following command in your terminal:

```shell
gem install bundler
```

2. If you don't already have a Gemfile for your app, run the following command in your terminal:

```shell
bundle init
```

3. If you don't already have a Gemfile.lock for your app, run the following command in your terminal:

```shell
bundle install
```

4. Install the Sinatra gem by running the following command in your terminal:

```shell
bundle add sinatra
```

Ruby example: Write the code

Create a Ruby file with the following contents. Modify the code to handle the event types that your webhook is subscribed to, as well as the `ping` event that GitHub sends when you create a webhook. This example handles the `issues` and `ping` events.

```ruby
require 'sinatra'
require 'json'
```

These are the dependencies for this code. You installed the `sinatra` gem earlier. For more information, see "Ruby example: Install dependencies." The `json` library is a standard Ruby library, so you don't need to install it.

```ruby
post '/webhook' do
```

The `/webhook` route matches the path that you specified for the smee.io forwarding. For more information, see "Forward webhooks."

Once you deploy your code to a server and update your webhook URL, you should change this to match the path portion of the URL for your webhook.

```
  status 202
```

Respond to indicate that the delivery was successfully received. Your server should respond with a 2XX response within 10 seconds of receiving a webhook delivery. If your server takes longer than that to respond, then GitHub terminates the connection and considers the delivery a failure.

```
  github_event = request.env['HTTP_X_GITHUB_EVENT']
```

Check the `X-GitHub-Event` header to learn what event type was sent. Sinatra changes `X-GitHub-Event` to `HTTP_X_GITHUB_EVENT`.

```
  if github_event == "issues"
    data = JSON.parse(request.body.read)
    action = data['action']
    if action == "opened"
      puts "An issue was opened with this title: #{data['issue']['title']}"
    elsif action == "closed"
      puts "An issue was closed by #{data['issue']['user']['login']}"
    else
      puts "Unhandled action for the issue event: #{action}"
    end
  elsif github_event == "ping"
    puts "GitHub sent the ping event"
  else
    puts "Unhandled event: #{github_event}"
  end
end
```

You should add logic to handle each event type that your webhook is subscribed to. For example, this code handles the `issues` and `ping` events.

If any events have an `action` field, you should also add logic to handle each action that you are interested in. For example, this code handles the `opened` and `closed` actions for the `issue` event.

For more information about the data that you can expect for each event type, see "Webhook events and payloads."

Ruby example: Test the code

To test your webhook, you can use your computer or codespace to act as a local server. If you have trouble with these steps, see Troubleshooting.

1. Make sure that you are forwarding webhooks. If you are no longer forwarding webhooks, follow the steps in Forward webhooks again.
2. In a separate terminal window, run the following command to start a local server on your computer or codespace. Replace `FILE_PATH` with the path to the file where your code from the previous section is stored. Note that `PORT=3000` matches the port that you specified for the webhook forwarding in the previous step.

```shell
PORT=3000 ruby FILE_NAME
```

You should see output that indicates something like "Sinatra has taken the stage on 3000".

3. Trigger your webhook. For example, if you created a repository webhook that is subscribed to the `issues` event, open an issue in your repository. You can also redeliver a previous webhook delivery. For more information, see "Redelivering webhooks."

4. Navigate to your webhook proxy URL on smee.io. You should see an event that corresponds to the event that you triggered or redelivered. This indicates that GitHub successfully sent a webhook delivery to the payload URL that you specified.

5. In the terminal window where you ran `smee --url WEBHOOK_PROXY_URL --path /webhook --port 3000`, you should see something like `POST http://127.0.0.1:3000/webhook - 202`. This indicates that smee successfully forwarded your webhook to your local server.

6. In the terminal window where you ran `PORT=3000 ruby FILE_NAME`, you should see a message corresponding to the event that was sent. For example, if you use the example code from above and you redelivered the `ping` event, you should see "GitHub sent the ping event". You may also see some other lines that Sinatra automatically prints.

7. In both terminal windows, enter `Ctrl` + `C` to stop your local server and stop listening for forwarded webhooks.

Now that you have tested out your code locally, you can make changes to use your webhook in production. For more information, see "Next steps." If you had trouble testing your code, try the steps in "Troubleshooting."

JavaScript example

This example uses Node.js and the Express library to define routes and handle HTTP requests. For more information, see "expressjs.com."

For an example that uses GitHub's Octokit.js SDK, see "Building a GitHub App that responds to webhook events."

This example requires your computer or codespace to run Node.js version 12 or greater and npm version 6.12.0 or greater. For more information, see Node.js.

JavaScript example: Install dependencies

To use this example, you must install the `express` library in your Node.js project. For example:

```shell
npm install express
```

Javascript example: Write the code

Create a JavaScript file with the following contents. Modify the code to handle the event types that your webhook is subscribed to, as well as the `ping` event that GitHub sends when you create a webhook. This example handles the `issues` and `ping` events.

JavaScript Beside Inline

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

You installed the `express` library earlier. For more information, see "JavaScript example: Install dependencies."

```
const app = express();
```

This initializes a new Express application.

```
app.post('/webhook', express.json({type: 'application/json'}), (request,
response) => {
```

This defines a POST route at the `/webhook` path. This path matches the path that you specified for the smee.io forwarding. For more information, see "Forward webhooks."

Once you deploy your code to a server and update your webhook URL, you should change this to match the path portion of the URL for your webhook.

```
  response.status(202).send('Accepted');
```

Respond to indicate that the delivery was successfully received. Your server should respond with a 2XX response within 10 seconds of receiving a webhook delivery. If your server takes longer than that to respond, then GitHub terminates the connection and considers the delivery a failure.

```
  const githubEvent = request.headers['x-github-event'];
```

Check the `x-github-event` header to learn what event type was sent.

```
  if (githubEvent === 'issues') {
    const data = request.body;
    const action = data.action;
    if (action === 'opened') {
      console.log(`An issue was opened with this title:
${data.issue.title}`);
    } else if (action === 'closed') {
      console.log(`An issue was closed by ${data.issue.user.login}`);
    } else {
      console.log(`Unhandled action for the issue event: ${action}`);
    }
  } else if (githubEvent === 'ping') {
    console.log('GitHub sent the ping event');
  } else {
    console.log(`Unhandled event: ${githubEvent}`);
  }
});
```

You should add logic to handle each event type that your webhook is subscribed to. For example, this code handles the `issues` and `ping` events.

If any events have an `action` field, you should also add logic to handle each action that you are interested in. For example, this code handles the `opened` and `closed` actions for the `issue` event.

For more information about the data that you can expect for each event type, see "Webhook events and payloads."

```
const port = 3000;
```

This defines the port where your server should listen. 3000 matches the port that you specified for webhook forwarding. For more information, see "Forward webhooks."

Once you deploy your code to a server, you should change this to match the port where your server is listening.

```
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

This starts the server and tells it to listen at the specified port.

JavaScript example: Test the code

To test your webhook, you can use your computer or codespace to act as a local server. If you have trouble with these steps, see Troubleshooting.

1. Make sure that you are forwarding webhooks. If you are no longer forwarding webhooks, follow the steps in Forward webhooks again.
2. In a separate terminal window, run the following command to start a local server on your computer or codespace. Replace `FILE_PATH` with the path to the file where your code from the previous section is stored.

   Shell

   ```
   node FILE_NAME
   ```

   You should see output that says `Server is running on port 3000`.
3. Trigger your webhook. For example, if you created a repository webhook that is subscribed to the `issues` event, open an issue in your repository. You can also redeliver a previous webhook delivery. For more information, see "Redelivering webhooks."
4. Navigate to your webhook proxy URL on smee.io. You should see an event that corresponds to the event that you triggered or redelivered. This indicates that GitHub successfully sent a webhook delivery to the payload URL that you specified.
5. In the terminal window where you ran `smee --url WEBHOOK_PROXY_URL --path /webhook --port 3000`, you should see something like `POST http://127.0.0.1:3000/webhook - 202`. This indicates that smee successfully forwarded your webhook to your local server.
6. In the terminal window where you ran `node FILE_NAME`, you should see a

message corresponding to the event that was sent. For example, if you use the example code from above and you redelivered the `ping` event, you should see "GitHub sent the ping event".

7. In both terminal windows, enter `Ctrl` + `C` to stop your local server and stop listening for forwarded webhooks.

Now that you have tested out your code locally, you can make changes to use your webhook in production. For more information, see "Next steps." If you had trouble testing your code, try the steps in "Troubleshooting."

Troubleshooting

If you don't see the expected results described in the testing steps, try the following:

- Make sure that your webhook is using your webhook proxy URL (Smee.io URL). For more information about your webhook proxy URL, see "Get a webhook proxy URL." For more information about your webhook settings, see "Creating webhooks."
- Make sure that your webhook uses the JSON content type, if you have a choice about what content type to use. For more information about your webhook settings, see "Creating webhooks."
- Make sure that both the smee client and your local server are running. You will have these processes running in two separate terminal windows.
- Make sure that your server is listening to the same port where smee.io is forwarding webhooks. All of the examples in this article use port 3000.
- Make sure that the path where smee.io is forwarding webhooks matches a route that is defined in your code. All of the examples in this article use the `/webhooks` path.
- Check for error messages in the terminal windows where you are running the smee client and your local server.
- Check GitHub to verify that a webhook delivery was triggered. For more information, see "Viewing webhook deliveries."
- Check your webhook proxy URL on smee.io. You should see an event that corresponds to the event that you triggered or redelivered. This indicates that GitHub successfully sent a webhook delivery to the payload URL that you specified.

Next steps

This article demonstrated how to write code to handle webhook deliveries. It also demonstrated how to test your code by using your computer or codespace as a local server and by forwarding webhook deliveries from GitHub to your local server via smee.io. Once you are done testing your code, you might want to modify the code and deploy your code to a server.

Modify the code

This article gave basic examples that print a message when a webhook delivery is received. You may want to modify the code to take some other action. For example, you could modify the code to:

- Make a request to the GitHub API
- Send a message on Slack
- Log events

- Update an external project management tool

Verify that the delivery is from GitHub

In your code that handles webhook deliveries, you should validate that the delivery is from GitHub before processing the delivery further. For more information, see "Validating webhook deliveries."

Deploy your code to a server

This article demonstrated how to use your computer or codespace as a server while you develop your code. Once the code is ready for production use, you should deploy your code to a dedicated server.

When you do so, you may need to update your code to reflect the host and port where your server is listening.

Update the webhook URL

Once you have a server that is set up to receive webhook traffic from GitHub, update the URL in your webhook settings. You may need to update the route that your code handles to match the path portion of the new URL. For example, if your new webhook URL is `https://example.com/github-webhooks`, you should change the route in these examples from `/webhooks` to `/github-webhooks`.

You should not use smee.io to forward your webhooks in production.

Follow best practices

You should aim to follow best practices with your webhooks. For more information, see "Best practices for using webhooks."

Further reading

- "Building a GitHub App that responds to webhook events"
- "Best practices for using webhooks"