project Documentation

Release 1.0

Sviatoslav Sydorenko

Feb 20, 2020

Contents

GitHub provides a great platform for collaborating. You can take it to the next level by creating custom GitHub bots. By delegating some of the chores to a bot, you get to spend more time developing your project and collaborating with others.

Learn how to automate your workflow by building a personal GitHub assistant for your own project. We'll use a framework called octomachinery to write a GitHub bot that does the following:

- *Greet the person who created an issue in your project*.
- *Say thanks when a pull request has been closed*.
- *Apply a label to issues or pull requests*.
- *Gives a thumbs up reaction to comments you made*. (becoming your own personal cheer squad).

The best part is, you get to do all of the above using Python 3.7 as the framework heavily relies on some of its features!

This tutorial is heavily based on Build-a-GitHub-Bot Workshop by Mariatta Wijaya which she first presented at PyCon US 2018. Yet, current version uses a bit more higher level approaches to abstract some implementation details away from programmers and help them focus on the business logic part.

CHAPTER 1

Code of Conduct

Be open, considerate, and respectful.

CHAPTER 2

License

CHAPTER 3

Agenda

3.1 Preparation

Before coming to the tutorial, please do the following:

1. Have Python 3.7 installed in your laptop. **Note:** It's important that you're running Python 3.7 (or above) because we will use some of the new features added in 3.7 in this tutorial.
    - Here's a tutorial that will help you get set up
2. Create a GitHub account, if you don't already have one. If your account has two-factor-authentication enabled, bring that device. (Yubikey, your phone).
3. Create a Heroku account. Your bot will be deployed to Heroku.
4. Install Heroku Toolbelt. (Optional, it can be useful for debugging later).

3.2 GitHub Bots: What and Why

Welcome to the Hands-on: Creating GitHub Bots to deal with boring routines!

3.2.1 What are GitHub bots?

Web applications that run automation on GitHub, using the ecosystem of GitHub Apps and its APIs.

3.2.2 What can bots do?

Many things: it can automatically respond to users, apply labels, close issues, create new issues, and even merge pull requests. Use the extensive GitHub APIs and automate your workflow!

3.2.3 Why use bots?

By automating your workflow, you can focus on real collaboration, instead of getting stuck doing boring housekeeping things.

3.2.4 Example GitHub bots

Chronographer

Source code: https://github.com/sanitizers/chronographer-github-app

Waits for Pull Request related events, as well as `check_suite` and `check_run` with `rerequested` action. Every time PR is updated, it:

- retrieves and parses the PR diff
- checks whether it contains additions to changelog fragment files
- reports intermediate statuses and the final outcome back to GitHub using Checks API
- uses nice visual indication and highlights the details on Checks page
- can be used to block PRs where the author forgot to add a change fragment
- is a GitHub App, unlocking the ability to use certain APIs
- can be used as a GitHub Action

the-knight-who-says-ni

Source code: https://github.com/python/the-knights-who-say-ni

Waits for incoming CPython's pull requests. Each time a pull request is opened, it does the following:

- find out the author's info
- find out if the author has signed the CLA
- if the author has not signed the CLA, notify the author
- if the author has signed the CLA, apply the CLA signed Label

bedevere-bot

Source code: https://github.com/python/bedevere

Performs status checks, identify issues and stages of the pull request. Some tasks that bedevere-bot does:

- identify the stage of the pull request, one of: awaiting review, awaiting merge, awaiting change request, awaiting core dev review.
- apply labels to pull requests
- checks if the PR contains reference to an issue
- automatically provide link to the issue in the bug tracker

miss-islington

Source code: https://github.com/python/miss-islington

Automatically create backport pull requests and reminds core devs that status checks are completed.

In addition, miss-islington can also automatically merge the pull request, and delete merged branch.

3.3 Resources

Tools and documentations that we'll use throughout this tutorial.

3.3.1 venv

See also: Python venv tutorial documentation.

It is recommended that you install the Python packages inside a virtual environment. For this tutorial, we'll use `venv` (but feel free to use any other wrapper you are comfortable with).

Create a new virtual environment using venv:

```
python3.7 -m venv tutorial-env
```

Activate the virtual environment. On Unix, Mac OS:

```
source tutorial-env/bin/activate
```

On Windows:

```
tutorial-env\Scripts\activate.bat
```

3.3.2 GitHub API v3 documentation

- Issues API
- Pull requests API
- Reactions API
- Event Types & Payloads

3.3.3 octomachinery

- Installation:

  ```
  python3.7 -m pip install octomachinery==0.2.1
  ```

- octomachinery documentation
- octomachinery source code
- Owner: Sviatoslav Sydorenko

3.3.4 gidgethub

- Installation: `pip install gidgethub`.
- gidgethub documentation
- gidgethub source code
- Owner: Brett Cannon

3.3.5 f-strings

We will use some f-strings during this tutorial.

Check out Mariatta's talk about f-strings.

Example:

```
first_name = "bart"
last_name = "simpson"

# old style %-formatting
print("Hello %s %s" % (first_name, last_name))

# str.format
print("Hello {first_name} {last_name}".format(first_name=first_name, last_name=last_
↪name))

# f-string
print(f"Hello {first_name} {last_name}")
```

3.3.6 asyncio

Both octomachinery and gidgethub are both async libraries. Read up the quick intro to asyncio.

3.3.7 Heroku

Python on Heroku documentation.

3.4 Using octomachinery client on the command line

Let's do some simple exercises of using GitHub API to create an issue. We'll be doing this locally using the command line, instead of actually creating the issue on GitHub website.

3.4.1 Install octomachinery

Install `octomachinery` if you have not already. Using a virtual environment is recommended.

```
python3.7 -m pip install octomachinery==0.2.1
```

3.4.2 Create GitHub Personal Access Token

In order to use GitHub API, you'll need to create a personal access token that will be used to authenticate you in GitHub.

1. Go to https://github.com/settings/tokens.

   Or, from GitHub, go to your profile Settings > Developer Settings > Personal access tokens.

2. Click `Generate new token`.

3. Under `Token description`, enter a short description, to identify the purpose of this token. I recommend something like: `pycon bot tutorial`.

4. Under select scopes, check the `repo` scope. You can read all about the available scopes here. In this tutorial, we'll only be using the token to work with repositories, and nothing else. But this can be edited later. What the `repo` scope allows your bot to do is explained in GitHub's scope documentation.

5. Press generate. You will see a really long string (40 characters). Copy that, and paste it locally in a text file for now.

   This is the only time you'll see this token in GitHub. If you lost it, you'll need to create another one.

3.4.3 Store the Personal Access Token as an environment variable

In Unix / Mac OS:

```
export GITHUB_TOKEN=your token
```

In Windows:

```
set GITHUB_TOKEN=your token
```

Note that these will only set the token for the current process. If you want this value stored permanently, you have to use another way of doing this, like using `dotenv`, `direnv` or similar (we'll omit this as it is out of scope of this tutorial).

3.4.4 Let's get coding!

Create a new Python file, for example: `create_issue.py`, and open it in your favorite text editor.

> **Note:** You can optionally create a working directory where you'll put this file but for the purposes of this tutorial page it's unnecessary.

Copy the following into `create_issue.py`:

```
import asyncio


async def main():
    print('Hello world.')


asyncio.run(main())
```

Save and run it in the command line:

```
python3.7 -m create_issue
```

You should see "Hello world." printed. That was "Hello world" with asyncio!

3.4.5 Create an issue

Ok now we want to actually work with GitHub and `octomachinery`.

Add the following imports:

```
import os

from aiohttp.client import ClientSession
from octomachinery.github.api.tokens import GitHubOAuthToken
from octomachinery.github.api.raw_client import RawGitHubAPI
```

And replace `print("Hello world.")` with:

```
access_token = GitHubOAuthToken(os.environ['GITHUB_TOKEN'])
async with ClientSession() as http_session:
    gh = RawGitHubAPI(
        access_token,
        session=http_session,
        user_agent='webknjaz',
    )
```

Instead of "webknjaz" however, use your own GitHub username.

The full code now looks like the following:

```
import asyncio
import os

from aiohttp.client import ClientSession
from octomachinery.github.api.tokens import GitHubOAuthToken
from octomachinery.github.api.raw_client import RawGitHubAPI


async def main():
    access_token = GitHubOAuthToken(os.environ['GITHUB_TOKEN'])
    async with ClientSession() as http_session:
        gh = RawGitHubAPI(
            access_token,
            session=http_session,
            user_agent='webknjaz',
        )


asyncio.run(main())
```

So instead of printing out hello world, we're now instantiating a GitHub API client from `octomachinery`, we're telling it who we are ("webknjaz" in this example), and we're giving it the GitHub personal access token, which were stored as the `GITHUB_TOKEN` environment variable.

Now, let's create an issue in my personal repo.

Take a look at GitHub's documentation for creating a new issue.

It says, you can create the issue by making a `POST` request to the url `/repos/:owner/:repo/issues` and supply the parameters like `title` (required) and `body`.

With octomachinery's GitHub API client, this looks like the following:

```
await gh.post(
    '/repos/mariatta/strange-relationship/issues',
    data={
        'title': 'We got a problem',
        'body': 'Use more emoji!',
    },
)
```

Go ahead and add the above code right after you instantiate `RawGitHubAPI`.

Your file should now look like the following:

```
import asyncio
import os

from aiohttp.client import ClientSession
from octomachinery.github.api.tokens import GitHubOAuthToken
from octomachinery.github.api.raw_client import RawGitHubAPI


async def main():
    access_token = GitHubOAuthToken(os.environ['GITHUB_TOKEN'])
    async with ClientSession() as http_session:
        gh = RawGitHubAPI(
            access_token,
            session=http_session,
            user_agent='webknjaz',
        )
        await gh.post(
            '/repos/mariatta/strange-relationship/issues',
            data={
                'title': 'We got a problem',
                'body': 'Use more emoji!',
            },
        )


asyncio.run(main())
```

Feel free to change the title and the body of the message.

Save and run that. There should be a new issue created in the test repo. Check it out: https://github.com/mariatta/strange-relationship/issues

3.4.6 Comment on issue

Let's try a different exercise, to get ourselves more familiar with GitHub APIs.

Take a look at GitHub's create a comment documentation.

Try this yourself, and leave a comment in the issue you just created.

3.4.7 Close the issue

Let's now close the issue that you've just created.

Take a look at the documentation to edit an issue.

The method for closing an issue is `PATCH` instead of `POST`, which we've seen in the previous two examples. In addition, to delete an issue, you're basically editing an issue, and setting the `state` to `closed`.

Use GitHub API client to patch the issue:

```
await gh.patch(
    '/repos/mariatta/strange-relationship/issues{/number}',
    data={'state': 'closed'},
    url_vars={'number': '28'},
)
```

Replace `28` with the issue number you created.

3.4.8 Bonus exercise

Add reaction to an issue.

> **Attention:** Pay attention at the blue `Note:` box in the docs. Using this API endpoint requires setting a special marker with `squirrel-girl` codename in order to flag GitHub that you *really* want to access this *preview* api version. If you miss that, attempting to use this API will result in an error response from the GitHub platform.
>
> Please, pass it as `preview_api_version='squirrel-girl'` argument each time you make this API call.

3.5 Using octomachinery to respond to GitHub events

In the previous example, we've been interacting with GitHub by making requests to GitHub. And we've been doing that locally on our own machine.

In this section we'll use what we know so far and start building an actual bot. We'll create a webserver that responds to GitHub webhook events.

3.5.1 Webhook events

When an event is triggered in GitHub, GitHub can notify you about the event by sending you a POST HTTP request along with the payload.

Some example `events` are:

- issues: any time an issue is assigned, unassigned, labeled, unlabeled, opened, edited, closed, reopened, etc.
- pull_request: any time a pull request is opened, edited, closed, reopened, review requested, etc.
- status: any time there's status update.
- checks: any time there's check requested or its status update posted.

- installations: any time GitHub App is installed or removed from a repo or new update permissions are accepted by the user.

The complete list of events is listed here.

Since GitHub needs to send you POST requests for the webhook, it can't send them to your personal laptop. So we need to create a webservice that's open on the Internet.

To the cloud!

3.5.2 Create a new Heroku App

Login to your account on Heroku. You should land at https://dashboard.heroku.com/apps.

Click "New" > "Create a new app". Type in the app name and click "Create app" button. If you leave it empty, Heroku will assign a name for you.

On the app page, right-click on the "Open app" button and click "Copy link address". We'll need this a bit later, in the next step.

3.5.3 Create a new GitHub App

1. Go to profile Settings > Developer Settings > GitHub Apps > New GitHub App
2. Click "New GitHub App".
3. Fill in 3 required fields:
   - Webhook URL (*paste the URL of the Heroku app you copied earlier*, it should look smth like `https://your-heroku-app-name.herokuapp.com/`)
   - GitHub App name (this will be turned into an app slug)
   - Homepage URL (put some URL here, you might use the same as for webhooks and change it later)
   - **(New!)** Enable the checkbox called **Active**
4. Select the following permissions:
   - `Checks`: `Read & Write`
   - `Issues`: `Read & Write`
   - `Pull requests`: `Read & Write`
5. Go to "Subscribe to events" section and select:
   - `Check run`
   - `Check suite`
   - `Issue comment`
   - `Issues`
   - `Label`
   - `Pull request`
6. Keep `Only on this account` radio selected and hit "Create GitHub App"
7. Click "Generate a private key" in the bottom of the app page and download it onto your computer
8. Keep the app page open, you'll need it soon

3.5.4 Create a new repository on GitHub

You'll need a GitHub repository. We'll store the bot source code there which is essentially a webservice application. Further in this tutorial, it'll be referred to as `github-bot`.

Later we will also use this same repo to test our bot — we'll install a GitHub App there and it'll be able to interact with this repo.

You can create the repository on GitHub, and then clone it to your local machine. Otherwise, ensure that you know how to add a remote and push a pre-existing Git repo to GitHub.

Note: **Pro tip:** Use dashes (kebab-case) for your repo name like `github-bot` or `pycon-2020-github-app` so that this name is not a valid Python importable. This'll save you from confusion when you're running commands from a wrong directory (accidentally).

3.5.5 Create a github-bot

Let's get ready to write your own GitHub bot. To start, use your favorite text editor or IDE. Go to the directory where your github-bot is at (the root of the repository you've created earlier).

Inside that directory, create a `requirements.txt` file. Add `octomachinery` to it.

`requirements.txt`:

```
octomachinery
```

Now, let's create a `.env` file in the directory next to `requirements.txt`. Add fill it in with the development env vars. For simplicity, set `GITHUB_PRIVATE_KEY_PATH` var and run the bash one-liner as shown in the example below. It turns a multiline private key file contents into a properly escaped single-line string.

```
touch .env
GITHUB_PRIVATE_KEY_PATH=~/Downloads/your-app-slug.2019-03-24.private-key.pem
cat $GITHUB_PRIVATE_KEY_PATH | python3.7 -c 'import sys; inline_private_key=r"\n".
↪join(map(str.strip, sys.stdin.readlines())); print(f"GITHUB_PRIVATE_KEY='"'"'
↪{inline_private_key}'"'"'", end="")' >> .env
```

Now, copy-paste the *App ID* from the General App Settings page, which is still open in your browser and put it into *.env* file as a value for *GITHUB_APP_IDENTIFIER* variable. Also put there *DEBUG=true* and *ENV=dev*

`.env` should look like this now:

```
GITHUB_PRIVATE_KEY='-----BEGIN RSA PRIVATE KEY-----\n[..snip..]\n-----END RSA PRIVATE␣
↪KEY-----'
GITHUB_APP_IDENTIFIER=99999

DEBUG=true
ENV=dev
```

After that, create a `.gitignore` in the same folder, it should contain `.env` entry. You can use the following command to download appropriate template:

```
wget -O - https://www.gitignore.io/api/git%2Cdotenv%2Clinux%2Cpydev%2Cpython%2Cwindows
↪%2Cpycharm%2Ball%2Cjupyternotebooks%2Cvim%2Cwebstorm%2Cemacs >> .gitignore
```

In the same directory, create another directory called `github_bot`. Inside this new directory, create `__main__.py`.

Your `github-bot/` should now look as follows:

```
/github-bot
/github-bot/.env
/github-bot/.gitignore
/github-bot/requirements.txt
/github-bot/github_bot/__main__.py
```

We'll start by creating a simple octomachinery app in `__main__.py`.

Edit `__main__.py` as follows:

```
from octomachinery.app.server.runner import run as run_app


if __name__ == '__main__':
    run_app(
        name='PyCon-Bot-by-webknjaz',
        version='1.0.0',
        url='https://github.com/apps/pyyyyyycoooon-booooot111',
    )
```

Save the file. Your webserver is now ready. From the command line and at the root of your project, enter the following:

```
python3 -m github_bot
```

You should now see the following output:

```
DEBUG:octomachinery.app.server.runner:================ App version: 1.0.0␣
↪==================
DEBUG:asyncio:Using selector: EpollSelector
DEBUG:octomachinery.app.server.machinery:The GitHub App env is set to `dev`
INFO:octomachinery.app.server.machinery:Webhook secret is [NOT SET]: SIGNED WEBHOOKS␣
↪WILL BE REJECTED
INFO:octomachinery.app.server.machinery:Starting the following GitHub App:
INFO:octomachinery.app.server.machinery:* app id: 21717
INFO:octomachinery.app.server.machinery:* private key SHA-1 fingerprint:␣
↪7d:96:e8:e5:8f:07:b5:10:97:85:2a:f4:33:72:b7:08:a5:81:82:92
INFO:octomachinery.app.server.machinery:* user agent: PyCon-Bot-by-webknjaz/1.0.0␣
↪(+https://github.com/apps/pyyyyyycoooon-booooot111)
INFO:octomachinery.github.api.app_client:This GitHub App is installed into:
INFO:octomachinery.github.api.app_client:* Installation id 491111 (installed to␣
↪webknjaz)
INFO:octomachinery.app.server.machinery:================= Serving on http://
↪localhost:8080 ==================
```

Warning: If you see some configuration error about invalid value of a setting, try checking env vars exported in your current terminal session. The dotenv library (`envparse`) used in the framework doesn't substitute those vars with values from `.env` file if they already exist in your env. You may need to `unset` them before proceeding.

Open your browser and point it to http://localhost:8080. Alternatively, you can open another terminal and type:

```
curl -X GET localhost:8080
```

Whichever method you choose, you should see the output: "405: Method Not Allowed". That's expected: since the GitHub Apps event receiver is only supposed to process HTTP POST requests, other methods are not allowed.

3.5.6 Update the Config Variables in Heroku

Almost ready to actually start writing bots! Are you still on the Heroku dashboard? We are not done there just yet :)

Go to the **Settings** tab.

Click on the **Reveal Config Vars** button. Add three config variables here.

The first one called **GITHUB_APP_IDENTIFIER**. Copy it from `.env` file you've created earlier.

The next one is called **GITHUB_PRIVATE_KEY**. Copy it directly from the private key file you've downloaded earlier. No conversion is needed this time.

Finally, set **HOST** to `0.0.0.0`, `DEBUG=false` and `ENV=prod`.

3.5.7 Deploy to Heroku

Before we go further, let's first get that webservice deployed to Heroku.

At the root of your project, create a new file called `Procfile`, (without any extension). This file tells Heroku how it should run your app.

Inside `Procfile`:

```
web: python3 -m github_bot
```

This will tell Heroku to run a web dyno using the command `python3 -m github_bot`.

Additionally, create `runtime.txt` file next to it containing:

```
python-3.7.2
```

This ensures that Heroku will provide Python 3.7 for us. Just as we need!

Your file structure should now look like the following:

```
/github-bot
/github-bot/.env
/github-bot/.gitignore
/github-bot/requirements.txt
/github-bot/runtime.txt
/github-bot/Procfile
/github-bot/github_bot/__main__.py
```

Commit everything (except for `.env` file!) and push to GitHub.

Open Heroku app dashboard (it may still be open somewhere among your browser tabs).

Go to the "Deploy" tab. Under "Deployment method", choose GitHub. Connect your GitHub account if you haven't done that.

Under "Search for a repository to connect to", enter your project name, e.g "github-bot". Press "Search". Once it found the right repo, press "Connect".

Scroll down. Under Deploy a GitHub branch, choose "master", and click "Deploy Branch". (Optionally, enable automatic deployments)

Watch the build log, and wait until it finished.

When you see "Your app was successfully deployed", click on the "View" button.

You should see "405: Method Not Allowed" (just as it was locally).

Tip: Install Heroku toolbelt to see your logs. Once you have Heroku toolbelt installed, you can read the logs by:

```
heroku logs -a <app name>
```

Pro tip: Install Timber.io Logging addon or similar to have a nicer view to more logs right in your browser.

3.5.8 Your first GitHub bot!

Ok NOW everything is finally ready. Let's start with something simple. Let's have a bot that **responds to every newly created issue in your project**. For example, whenever someone creates an issue, the bot will automatically say something like: "Thanks for the report, @user. I will look into this ASAP!"

Go to the `__main__.py` file, in your `github_bot` codebase.

The first change the part where we did is to add the following imports:

```
from octomachinery.routing import process_event_actions
from octomachinery.routing.decorators import process_webhook_payload
from octomachinery.runtime.context import RUNTIME_CONTEXT
```

Add the following coroutine (above **if __name__ == "__main__":**):

```
@process_event_actions('issues', {'opened'})
@process_webhook_payload
async def on_issue_opened(*, issue, **_kw):
    """Whenever an issue is opened, greet the author and say thanks."""
    github_api = RUNTIME_CONTEXT.app_installation_client
```

This is where we are essentially subscribing to the GitHub `issues` event, and specifically to the "opened" issues event.

`@process_webhook_payload` decorator automagically "unpacks" the event payload fields into the function arguments.

`github_api` is a GitHub API client wrapper, which we've used in the previous section to make API calls to GitHub. Here, we get it from the contextvar proxy context, offered by octomachinery under the hood. This client is authorized against the installation bound to the current incoming event.

Leave a comment whenever an issue is opened

Back to the task at hand. We want to *leave a comment whenever someone opened an issue*. Now that we're subscribed to the event, all we have to do now is to actually create the comment.

We've done this in the previous section on the command line. You will recall the code is something like the following:

```
await github_api.post(url, data={"body": message})
```

Let's think about the `url` in this case. Previously, you might have constructed the url manually as follows:

```
url = f"/repos/mariatta/strange-relationship/issues/{issue_number}/comments"
```

When we receive the webhook event however, the issue comment url is actually supplied in the payload.

Take a look at GitHub's issue event payload example (scroll it a bit).

It's a big JSON object. The portion we're interested in is:

```
{
  "action": "opened",
  "issue": {
    "url": ...,
    "comments_url": "https://api.github.com/repos/baxterthehacker/public-repo/issues/
↪2/comments",
    "events_url": "...",
    "html_url": "...",
  ...
}
```

Notice that `["issue"]["comments_url"]` is actually the URL for posting comments to this particular issue. With this knowledge, your url is now:

```
comments_api_url = issue['comments_url']
```

The next piece we want to figure out is what should the comment message be. For this exercise, we want to greet the author, and say something like "Thanks @author!".

Take a look again at the issue event payload:

```
{
  "action": "opened",
  "issue": {
    "url": "...",
     ...
    "user": {
      "login": "baxterthehacker",
      "id": ...,
  ...
}
```

Did you spot it? The author's username can be accessed by `issue["user"]["login"]`.

So now your comment message should be:

```
author = issue['user']['login']
message = (
    f'Thanks for the report @{author}! '
    "I will look into it ASAP! (I'm a bot )."
)
```

Piece all of that together, and actually make the API call to GitHub to create the comment:

```
@process_event_actions('issues', {'opened'})
@process_webhook_payload
async def on_issue_opened(*, issue, **_kw):
    """Whenever an issue is opened, greet the author and say thanks."""
    github_api = RUNTIME_CONTEXT.app_installation_client

    comments_api_url = issue['comments_url']
    author = issue['user']['login']
    message = (
        f'Thanks for the report @{author}! '
        "I will look into it ASAP! (I'm a bot )."
    )
    await github_api.post(comments_api_url, data={'body': message})
```

Your entire **__main__.py** should look like the following:

```
from octomachinery.app.server.runner import run as run_app
from octomachinery.routing import process_event_actions
from octomachinery.routing.decorators import process_webhook_payload
from octomachinery.runtime.context import RUNTIME_CONTEXT


@process_event_actions('issues', {'opened'})
@process_webhook_payload
async def on_issue_opened(*, issue, **_kw):
    """Whenever an issue is opened, greet the author and say thanks."""
    github_api = RUNTIME_CONTEXT.app_installation_client

    comments_api_url = issue['comments_url']
    author = issue['user']['login']
    message = (
        f'Thanks for the report @{author}! '
        "I will look into it ASAP! (I'm a bot )."
    )
    await github_api.post(comments_api_url, data={'body': message})


if __name__ == '__main__':
    run_app(
        name='PyCon-Bot-by-webknjaz',
        version='1.0.0',
        url='https://github.com/apps/pyyyyyycoooon-booooot111',
    )
```

Commit that file, push it to GitHub, and deploy it in Heroku.

Almost there!

Go to “Install App” tab in the GitHub App settings and install it into your test repo from there. It’s needed so that your bot would start actually receiving events from that repository.

Try and create an issue in the repo. See your bot in action!!

Congrats! You now have a bot in place! Let’s give it another job.

Say thanks when a PR has been merged

Let’s now have the bot **say thanks, whenever a pull request has been merged**.

For this case, you’ll want to subscribe to the `pull_request` event, specifically when the `action` to the event is `closed`.

For reference, the relevant GitHub API documentation for the `pull_request` event is here: https://developer.github.com/v3/activity/events/types/#pullrequestevent.

Scroll a bit to see the example payload for this event.

Try it on your own.

Note: A pull request can be closed without it getting merged. You’ll need a way to determine whether the pull request was merged, or simply closed.

React to issue comments

Everyone has opinion on the internet. Encourage more discussion by **automatically leaving a thumbs up reaction** for every comments in the issue. Ok you might not want to actually do that, (and whether it can actually encourage more discussion is questionable). Still, this can be a fun exercise.

How about if the bot always gives **you** a thumbs up?

Try it out on your own.

- The relevant documentation is here: https://developer.github.com/v3/activity/events/types/#issuecommentevent
- The example payload for the event is next to it
- The API documentation for reacting to an issue comment is here: https://developer.github.com/v3/reactions/#create-reaction-for-an-issue-comment

Label the pull request

Let's make your bot do even more hard work. **Each time someone opens a pull request, have it automatically apply a label**. This can be a "pending review" or "needs review" label.

The relevant API call is this: https://developer.github.com/v3/issues/#edit-an-issue

3.6 Make your GitHub bot activity feedback more colorful

Previously, we've used things which can be directly created in the GitHub UI by regular users. But what if we need more? What if our robot has complex multistage flows with rich result set? In such case, we need to use Checks API — it's a way to built your bot's activity outcomes right into GitHub's UI. You can add some markdown-formatted content, upload a set of pictures, add annotations right to your PRs' diff views and even put a few completely custom buttons on your App's Checks page empowering users to trigger some actions which your application would process. And, of course, it facilitates separate indication of the processing state and check outcome.

In this section, we'll extend the application that we've built earlier.

3.6.1 Work-in-progress indicator

Sometimes, when you work on Pull Request, you want to block it from being merged accidentally.

Let's build a status check which will indicate that the PR is WIP and will help to block it from being merged accidentally.

Add the following PR event handler:

```
@process_event_actions('pull_request', {'opened', 'edited'})
@process_webhook_payload
async def on_pr_check_wip(*, pull_request, repository, **_kw):
    """React to an opened or changed PR event.

    Send a status update to GitHub via Checks API.
    """
    github_api = RUNTIME_CONTEXT.app_installation_client
```

This one is pretty easy, you're already familiar with this structure: it's an event handler for PR openning and editing.

Let's extend it with some useful logic now.

This is where Checks API interaction begins. Every GitHub App's got a Checks Suite attached to it. It's visible on the Checks page to where you can get using either Checks tab in PRs or click a commit status from the commit indicator on the branch page with commits list.

First thing we need to do is to create a Check Run which is an entity representing a single task of validating something having a separate subpage and status indicator in the GitHub UI.

Once we grab its ID we'll be able to use that in order to update progress, status and details of this check task as more of its code gets executed.

Here we add two GitHub API calls: one creates a check run with the *queued* initial status and the other updates that status to *in_progress*.

Warning: Please don't use emoji in the `check_run_name` (corresponding to Check Run) of the payload when working with Checks API. At least not until the GitHub protected branches bug gets solved

```
from datetime import datetime
```

```
check_run_name = 'Work-in-progress state'

pr_head_sha = pull_request['head']['sha']
repo_api_url = repository['url']

check_runs_base_uri = f'{repo_api_url}/check-runs'

resp = await github_api.post(
    check_runs_base_uri,
    preview_api_version='antiope',
    data={
        'name': check_run_name,
        'head_sha': pr_head_sha,
        'status': 'queued',
        'started_at': f'{datetime.utcnow().isoformat()}Z',
    },
)

check_runs_updates_uri = (
    f'{check_runs_base_uri}/{resp["id"]:d}'
)

resp = await github_api.patch(
    check_runs_updates_uri,
    preview_api_version='antiope',
    data={
        'name': check_run_name,
        'status': 'in_progress',
    },
)
```

Warning: Using this API requires setting a special marker with *antiope* codename in order to flag GitHub that you *really* want to access this preview api version. If you miss that, attempting to use this API will result in an error response from the GitHub platform.

Now, let's check the PR title and figure out whether it looks WIP or not:

```
pr_title = pull_request['title'].lower()
wip_markers = (
    'wip', '', 'dnm',
    'work in progress', 'work-in-progress',
    'do not merge', 'do-not-merge',
    'draft',
)

is_wip_pr = any(m in pr_title for m in wip_markers)
```

The last thing left is sending this information to GitHub. Let's include some illustrative data to the Checks page. For this, we'll use Markdown markup and some emojis .

Add this snippet in the end of our `on_pr_check_wip` event handler:

```
await github_api.patch(
    check_runs_updates_uri,
    preview_api_version='antiope',
    data={
        'name': check_run_name,
        'status': 'completed',
        'conclusion': 'success' if not is_wip_pr else 'neutral',
        'completed_at': f'{datetime.utcnow().isoformat()}Z',
        'output': {
            'title':
                ' This PR is not Work-in-progress: Good to go',
            'text':
                'Debug info:\n'
                f'is_wip_pr={is_wip_pr!s}\n'
                f'pr_title={pr_title!s}\n'
                f'wip_markers={wip_markers!r}',
            'summary':
                'This change is ready to be reviewed.'
                '\n\n'
                '![Go ahead and review it!]('
                'https://farm1.staticflickr.com'
                '/173/400428874_e087aa720d_b.jpg)',
        } if not is_wip_pr else {
            'title':
                ' This PR is Work-in-progress: '
                'It is incomplete',
            'text':
                'Debug info:\n'
                f'is_wip_pr={is_wip_pr!s}\n'
                f'pr_title={pr_title!s}\n'
                f'wip_markers={wip_markers!r}',
            'summary':
                ' Please do not merge this PR '
                'as it is still under construction.'
                '\n\n'
                '![Under constuction tape]('
                'https://cdn.pixabay.com'
                '/photo/2012/04/14/14/59'
                '/border-34209_960_720.png)\n'
                "![Homer's on the job]("
                'https://farm3.staticflickr.com'
                '/2150/2101058680_64fa63971e.jpg)',
        },
```

(continues on next page)

(continued from previous page)

```
    },
)
```

That's it! You can now commit, push and deploy your app to Heroku. Then, go create a PR in you test repo, try out adding WIP into its title and removing it. See what happens, visit Checks page...

3.6.2 Action buttons

Manual editing of PR title is nice but let's have more fun and add a button to the Checks page!

Extend the `data` argument of the last API call like this:

```
'actions': [
    {
        'label': 'WIP it!',
        'description': 'Mark the PR as WIP',
        'identifier': 'wip',
    } if not is_wip_pr else {
        'label': 'UnWIP it!',
        'description': 'Remove WIP mark from the PR',
        'identifier': 'unwip',
    },
],
```

Now, your Checks page will have *WIP it!* or *UnWIP it!* button available on the UI.

Clicking that button causes another event in GitHub. So now we have to write another handler to properly process and react to it.

Add this code to achieve what we need:

```
@process_event_actions('check_run', {'requested_action'})
@process_webhook_payload
async def on_pr_action_button_click(*, check_run, requested_action, **_kw):
    """Flip the WIP switch when user hits a button."""
    requested_action_id = requested_action['identifier']
    if requested_action_id not in {'wip', 'unwip'}:
        return

    github_api = RUNTIME_CONTEXT.app_installation_client

    wip_it = requested_action_id == 'wip'

    pull_request = check_run['pull_requests'][0]
    pr_api_uri = pull_request['url']

    pr_details = await github_api.getitem(pr_api_uri)

    pr_title = pr_details['title']

    if wip_it:
        new_title = f'WIP: {pr_title}'
    else:
        wip_markers = (
            'wip', '', 'dnm',
            'work in progress', 'work-in-progress',
```

(continues on next page)

(continued from previous page)

```
            'do not merge', 'do-not-merge',
            'draft',
        )

        wip_regex = fr'(\s*({"|".join(wip_markers)}):?\s+)'
        new_title = re.sub(
            wip_regex, '', pr_title, flags=re.I,
        ).replace('', '')

    await github_api.patch(
        pr_api_uri,
        data={
            'title': new_title,
        },
    )
```

We will also need to import regex library, add it in the top of our module.

```
import re
```

So this basically edits PR title depending on which of two buttons have been clicked.

Redeploy your updated code to Heroku and have some fun with it!

3.7 What's Next?

You now have built yourself a fully functional GitHub bot! Congratulations!!

However, the bot you've built today might not be the GitHub bot you really want. That's fine. The good thing is you've learned how to build one yourself, and you have access to all the libraries, tools, documentation needed in order to build another GitHub bot.

3.7.1 Additional ideas and inspirations

Automatically delete a merged branch

Related API: https://developer.github.com/v3/git/refs/#delete-a-reference.

The branch name can be found in the pull request webhook event.

Monitor comments in issues / pull request

Have your bot detect blacklisted keywords (e.g. offensive words, spammy contents) in issue comments. From there you can choose if you want to delete the comment, close the issue, or simply notify you of such behavior.

Automatically merge PRs when all status checks passed

Folks using GitLab have said that they wished that this is available on GitHub. You can have a bot that does this! We made miss-islington do this for CPython.

Detect when PR has merge conflict

When merge conflict occurs in a pull request, perhaps you can apply a label or tell the PR author about it, and ask them to rebase. You might have to do this as a scheduled task or a cron job.

3.7.2 Other topics

Rate limit

You have a limit of 5000 API calls per hour using the OAuth token. The Rate Limit API docs have more info on this.

Unit tests with pytest

bedevere has 100% test coverage using pytest and pytest-asyncio. You can check the source code to find out how to test your bot.

Error handling

gidgethub exceptions documentation.

3.7.3 Shout out to your bot

Share with the world the bot that you just made. This is completely optional, but highly encouraged. Go to the *Hall of Fame: Bots By Students* for more details.

3.8 Hall of Fame: Bots By Students

Thank you so much for taking this tutorial!!

Whether you've taken this tutorial at PyCon, or on your own at home, feel free to share the bot that you've built with us.

Create a pull request and add a new entry below.

Example:

```
Name:
Source code:
Bot:
Repo using the bot:
Other info: (what does your bot do?)
Which workshop: (PyCon SK? PyCon CZ? Somewhere else? On your own?)
```

3.8.1 The Foreman PR Processor

- Name: PR Processor
- Source code: https://github.com/theforeman/prprocessor/tree/app
- Repo using the bot: theforeman/*

- Other info: Links the relevant Redmine issues based on commit messages. It verifies they're correct, such as part of the correct Redmine project.
- Which workshop: manual

3.8.2 Sefkhet-Abwy

- Name: Sefkhet-Abwy
- Source code: https://github.com/AICoE/Sefkhet-Abwy
- Repo using the bot: all of https://github.com/thoth-station/ and some of https://github.com/AICoE/
- Other info: This bot is supporting some team to manage source code: adding labels, preparing reviews of PR, ...
- Which workshop: manual

3.9 Git Cheat Sheet

There are other more complete git tutorials out there, but the following should help you during this workshop.

3.9.1 Clone a repo

```
git clone <url>
```

3.9.2 Create a new branch, and switch to it

```
git checkout -b <branchname>
```

3.9.3 Switching to an existing branch

```
git checkout <branchname>
```

3.9.4 Adding files to be committed

```
git add <filename>
```

3.9.5 Commit your changes

```
git commit -m "<commit message>"
```

3.9.6 Pushing changes to remote

```
git push <remote> <branchname>
```

3.9.7 Rebase with a branch on remote

```
git rebase <remote>/<branchname>
```

3.9.8 Extras

For more instructions, please consult with the Flight rules for Git.