github-bot-tutorial Documentation

Mariatta

Jul 05, 2019

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 libraries called gidgethub and aiohttp 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 3.6! F-strings included!

This tutorial is for PyCon US 2018 in Cleveland, Ohio. Video recording is here.

CHAPTER 1

About me

My name is Mariatta. I live in Vancouver, Canada where I work as a Software Engineer for Zapier. In my free time, I help organize Vancouver PyLadies, PyCascades conference, and contribute to open source.

I'm a Python Core Developer. I help maintain Core Python's GitHub bots: bedevere and miss-islington.

We'll be using the same tools and technologies used by Core Python's team to build their bots.

If you have any feedback or questions about this tutorial, please file an issue.

- E-mail: mariatta@python.org
- Twitter: @mariatta
- Mariatta on GitHub
- Mariatta on Patreon

Know of other places, conferences, or events where I can give this tutorial? Let me know!

CHAPTER 2

Code of Conduct

Be open, considerate, and respectful.

CHAPTER 3

License

CHAPTER 4

Agenda

4.1 Preparation

Before coming to the tutorial, please do the following:

1. Have Python 3.6 installed in your laptop. **Note:** It's important that you're running Python 3.6 (or above) because we will use some of the new features added in 3.6 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).

4.2 GitHub Bots: What and Why

Welcome to the Build-a-GitHub Bot workshop!

4.2.1 What are GitHub bots?

Applications that runs automation on GitHub, using GitHub WebHooks and APIs.

4.2.2 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!

4.2.3 Why use bots?

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

4.2.4 Example GitHub bots

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.

4.3 Resources

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

4.3.1 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`.

Create a new virtual environment using venv:

```
python3.6 -m venv tutorial-env
```

Activate the virtual environment. On Unix, Mac OS:

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

On Windows:

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

4.3.2 GitHub API v3 documentation

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

4.3.3 gidgethub

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

4.3.4 aiohttp

- Installation: `pip install aiohttp`.
- aiohttp documentation
- *aiohttp* source code
- Owner: Andrew Svetlov

4.3.5 f-strings

We will use some f-strings during this tutorial.

My 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))
```

(continues on next page)

(continued from previous page)

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

4.3.6 asyncio

Both *gidgethub* and *aiohttp* are async libraries. Read up the quick intro to asyncio.

4.3.7 Heroku

Python on Heroku documentation.

4.4 Using gidgethub 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 in GitHub website.

4.4.1 Install gidgethub and aiohttp

Install `gidgethub` and `aiohttp` if you have not already. Using a virtual environment is recommended.

```
python3.6 -m pip install gidgethub
python3.6 -m pip install aiohttp
```

4.4.2 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 yourself to 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.

4.4.3 Store the Personal Access Token as an environment variable

In Unix / Mac OS:

```
export GH_AUTH=your token
```

In Windows:

```
set GH_AUTH=your token
```

Note that these will only set the token for the current process. If you want this value stored permanently, you have to edit the bashrc file.

4.4.4 Let's get coding!

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

Copy the following into `create_issue.py`:

```
import asyncio

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

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Save and run it in the command line:

```
python3.6 -m create_issue
```

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

4.4.5 Create an issue

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

Add the following imports:

```
import os
import aiohttp
from gidgethub.aiohttp import GitHubAPI
```

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

```
async with aiohttp.ClientSession() as session:
    gh = GitHubAPI(session, "mariatta", oauth_token=os.getenv("GH_AUTH"))
```

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

The full code now looks like the following:

```
import asyncio
import os
import aiohttp
from gidgethub.aiohttp import GitHubAPI
```

(continues on next page)

(continued from previous page)

```
async def main():
    async with aiohttp.ClientSession() as session:
        gh = GitHubAPI(session, "mariatta", oauth_token=os.getenv("GH_AUTH"))

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

So instead of printing out hello world, we're now instantiating a GitHub API from gidgethub, we're telling it who we are ("mariatta" in this example), and we're giving it the GitHub personal access token, which were stored as the `GH_AUTH` 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 gidgethub, 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 GitHubAPI.

Your file should now look like the following:

```
import asyncio
import os
import aiohttp
from gidgethub.aiohttp import GitHubAPI

async def main():
   async with aiohttp.ClientSession() as session:
       gh = GitHubAPI(session, "mariatta", oauth_token=os.getenv("GH_AUTH"))
       await gh.post('/repos/mariatta/strange-relationship/issues',
             data={
                 'title': 'We got a problem',
                 'body': 'Use more emoji!',
             })

loop = asyncio.get_event_loop()
loop.run_until_complete(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 my repo. Check it out: https://github.com/mariatta/strange-relationship/issues

4.4.6 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.

4.4.7 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 deleting 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 gidgethub to patch the issue:

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

Replace `28` with the issue number you created.

4.4.8 Bonus exercise

Add reaction to an issue.

4.5 Using gidgethub to respond to webhooks

In the previous example, we've been interacting with GitHub by doing actions: 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.

4.5.1 Webhook events

When an event is triggered in GitHub, GitHub can notify you about the event by sending you a POST 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.

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!

4.5.2 Create two new repositories on GitHub

You'll need two repositories. The first one is to hold the codebase for the bot, essentially the webservice. This is where you will actually push the codes. Going forward I will call this the `webservice`.

The other repo, will mostly be empty for now. It will be the repo where the webhook will be installed, and your bot will interact with this repo. In real world, this will be the project that you're managing. I will be using my personal repo, `strange-relationship`.

You can create the repositories on GitHub, and then clone it to your local machine.

4.5.3 Create a webservice

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 webservice is at.

Inside that directory, create a *requirements.txt* file. Add both *gidgethub* and *aiohttp* to it.

requirements.txt:

```
aiohttp
gidgethub
```

In the same directory, create another directory with the same name as your webservice/bot's name. Inside this new directory, create `__main__.py`.

Your webservice should now look like the following:

```
/webservice
/webservice/requirements.txt
/webservice/webservice/__main__.py
```

We'll start by creating a simple aiohttp webserver in `__main__.py`.

Edit `__main__.py` as follows:

```
import os

from aiohttp import web

routes = web.RouteTableDef()

@routes.get("/")
async def main(request):
    return web.Response(status=200, text="Hello world!")


if __name__ == "__main__":
    app = web.Application()
    app.add_routes(routes)
    port = os.environ.get("PORT")
    if port is not None:
        port = int(port)

    web.run_app(app, port=port)
```

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

```
python3 -m webservice
```

You should now see the following output:

```
======== Running on http://127.0.0.1:8080 ========
(Press CTRL+C to quit)
```

Open your browser and point it to http://127.0.0.1:8080. Alternatively, you can open another terminal and type:

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

Whichever method you choose, you should see the output: "Hello World".

4.5.4 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 webservice
```

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

Your file structure should now look like the following:

```
/webservice
/webservice/requirements.txt
/webservice/Procfile
/webservice/webservice/__main__.py
```

Commit everything and push to GitHub.

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, choose the United States region, and click "Create app" button. If you leave it empty, Heroku will assign a name for you.

Once your web app has been created, 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 "webservice". Press "Search". Once it found the right repo, press "Connect".

Scroll down. Under Deploy a GitHub branch, choose "master", and click "Deploy Branch".

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 "Hello world.". Copy the website URL.

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>
```

4.5.5 Add the GitHub Webhook

Now we have a webservice, we can receive GitHub webhooks from it. Go to your project settings on GitHub (not the "webservice" repo, the other one you created earlier). I will use my own project, "strange-relationship".

In the Settings page, choose Webhooks, and click "Add webhook".

In the **Payload URL**, type in your webservice URL from Heroku.

Choose the **Content type** `application/json`.

For the **Secret**, for security reasons, type in some random characters. Read up more about Securing your webhooks. **Take note of this secret token**.

Choose `Let me select individual events.`. Check the following:

- Issues
- Issue comments
- Pull requests

Press Add Webhook.

4.5.6 Update the Config Variables in Heroku

Almost ready to actually start writing bots! Let's go back to your Heroku dashboard real quick.

Go to the **Settings** tab.

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

The first one called **GH_SECRET**. This is the secret webhook token you just created a few moment ago.

The next one is called **GH_AUTH**. This is your GitHub personal access token, which you used in the previous section, when we worked with gidgethub on the command line.

4.5.7 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 webservice codebase.

The first change the part where we did:

```
@routes.get("/")
```

into:

```
@routes.post("/")
```

This is because GitHub will send you **POST** requests to the webhook instead of **GET**.

Next, add the following imports:

```
import aiohttp

from aiohttp import web

from gidgethub import routing, sansio
from gidgethub import aiohttp as gh_aiohttp

router = routing.Router()
```

Update the **main** coroutine, instead of printing out "Hello World":

```
async def main(request):
    # read the GitHub webhook payload
    body = await request.read()

    # our authentication token and secret
    secret = os.environ.get("GH_SECRET")
    oauth_token = os.environ.get("GH_AUTH")

    # a representation of GitHub webhook event
    event = sansio.Event.from_http(request.headers, body, secret=secret)

    # instead of mariatta, use your own username
    async with aiohttp.ClientSession() as session:
        gh = gh_aiohttp.GitHubAPI(session, "mariatta",
                                  oauth_token=oauth_token)

        # call the appropriate callback for the event
        await router.dispatch(event, gh)

    # return a "Success"
    return web.Response(status=200)
```

Add the following coroutine (above **main**):

```
@router.register("issues", action="opened")
async def issue_opened_event(event, gh, *args, **kwargs):
    """ Whenever an issue is opened, greet the author and say thanks."""
    pass
```

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

The two important parameters here are: `event` and `gh`.

`event` here is the representation of GitHub's webhook event. We can access the event payload by doing `event.data`.

`gh` is the gidgethub GitHub API, which we've used in the previous section to make API calls to GitHub.

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 gh.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.

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:

```
url = event.data["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 `event.data["user"]["login"]`.

So now your comment message should be:

```
author = event.data["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:

```
@router.register("issues", action="opened")
async def issue_opened_event(event, gh, *args, **kwargs):
    """ Whenever an issue is opened, greet the author and say thanks."""

    url = event.data["issue"]["comments_url"]
```

(continues on next page)

(continued from previous page)

```
    author = event.data["issue"]["user"]["login"]

    message = f"Thanks for the report @{author}! I will look into it ASAP! (I'm a␣
↪bot)."
    await gh.post(url, data={"body": message})
```

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

```
import os
import aiohttp

from aiohttp import web

from gidgethub import routing, sansio
from gidgethub import aiohttp as gh_aiohttp

routes = web.RouteTableDef()

router = routing.Router()

@router.register("issues", action="opened")
async def issue_opened_event(event, gh, *args, **kwargs):
    """
    Whenever an issue is opened, greet the author and say thanks.
    """
    url = event.data["issue"]["comments_url"]
    author = event.data["issue"]["user"]["login"]

    message = f"Thanks for the report @{author}! I will look into it ASAP! (I'm a␣
↪bot)."
    await gh.post(url, data={"body": message})

@routes.post("/")
async def main(request):
    body = await request.read()

    secret = os.environ.get("GH_SECRET")
    oauth_token = os.environ.get("GH_AUTH")

    event = sansio.Event.from_http(request.headers, body, secret=secret)
    async with aiohttp.ClientSession() as session:
        gh = gh_aiohttp.GitHubAPI(session, "mariatta",
                                  oauth_token=oauth_token)
        await router.dispatch(event, gh)
    return web.Response(status=200)


if __name__ == "__main__":
    app = web.Application()
    app.add_routes(routes)
    port = os.environ.get("PORT")
    if port is not None:
        port = int(port)

    web.run_app(app, port=port)
```

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

Once deployed, 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 an issue 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.

The example payload for this event is here: https://developer.github.com/v3/activity/events/types/#webhook-payload-example-28

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 here: https://developer.github.com/v3/activity/events/types/#webhook-payload-example-14
- 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

4.6 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.

4.6.1 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.

4.6.2 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.

4.6.3 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.

4.7 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:
Bot:
Repo using the bot:
Other info: (what does your bot do?)
Which workshop: (PyCon US? Somewhere else? On your own?)
```

Name: Aaron Meurer
Bot: SymPy Bot
Repo using the bot: sympy/sympy
Other info: The bot automatically gets release notes for each pull request from the pull request description, or lets the author know how to write it if it is missing. When the pull request is merged, it automatically adds the release notes for it to the release notes document on the SymPy wiki.
Which workshop: On my own

4.8 Git Cheat Sheet

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

4.8.1 Clone a repo

```
git clone <url>
```

4.8.2 Create a new branch, and switch to it

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

4.8.3 Switching to an existing branch

```
git checkout <branchname>
```

4.8.4 Adding files to be committed

```
git add <filename>
```

4.8.5 Commit your changes

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

4.8.6 Pushing changes to remote

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

4.8.7 Rebase with a branch on remote

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