Say It With Bots!

Jun 10, 2020

Contents:

This tutorial was prepared for PyCon US 2020. You're free to follow along from whereever you are.

This documentation lives at: https://github-app-tutorial.readthedocs.io/

CHAPTER 1

Welcoming Contributions to Your Project

In this tutorial we'll be building a GitHub bot that will automatically greet contributors to your project. The bot will be built as a GitHub App which can then be installed to your GitHub repositories.

We will be building a GitHub bot that can:

- say thanks to the maintainer who installed your bot
- say thanks to new contributors who made pull requests to your repository

CHAPTER 2

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 with PyLadies globally and locally in Vancouver, I'm one of the co-founders of PyCascades conference, and I also contribute to open source.

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

In this tutorial, 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
- Sponsor Mariatta

CHAPTER 3

Stay in touch

I love hearing how you did! If you had taken this tutorial, learned something new from it, or if you've built something cool after this tutorial, I'd love to hear about it. You can share it with me either through twitter, email, or even as an issue on GitHub.

CHAPTER 4

Code of Conduct

PyCon US Code of Conduct applies and will be enforced.

CHAPTER 5

License

CHAPTER 6

Agenda

6.1 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).

6.2 GitHub Bots: What and Why

6.2.1 What are GitHub bots?

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

6.2.2 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!

6.2.3 Why use bots?

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

As human, we can't always be up 24/7. Sometimes it is nice for your contributors if they can receive prompt response and feedback, instead of waiting until you're back.

6.2.4 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 the merged branch.

6.3 Gigdethub

Gidgethub is an async library in Python for working with GitHub APIs.

6.3.1 REST API calls

When you make API calls to GitHub, you need to provide API tokens and pass in certain request headers. Gidgethub provides the abstraction for making such API calls, as well as deciphering the request response.

Quick example of calling GitHub APIs using `requests` library.

```
import requests
# construct the request headers
request_headers = {
    "User-Agent": "cool-octocat-app",
    "Authorization": "token abcde",
    "Accept": "application/vnd.github.v3+json"
}
# make an API call
url = "https://api.github.com/repos/mariatta/gidgethub/strange-relationship/issues"
response = requests.get(url, headers=request_headers)
```

With gidgethub:

```
async with aiohttp.ClientSession() as session:
        gh = GitHubAPI(session,
            "cool-octocat-app",
            oauth_token="abcde"
        )
        response = await gh.getitem(
            '/repos/mariatta/strange-relationship/issues'
        )
```

We will go through more detailed examples in the *GitHub API using Command Line* section.

6.3.2 Webhook Events

Gidgethub provides routings for receiving webhook events from GitHub. Each routings allows for individual request handlers to be defined. We will cover this in detail in the *Responding to Webhook Events* section.

Additionally, gidgethub takes care of verifying the webhook delivery headers, and the webhook secret to help protect your webservice.

6.3.3 GitHub App

Certain API endpoints for GitHub Apps require JWT instead of OAuth access token. Since version 4.1, gidgethub comes several utilities to help with this. We will go further in detail in the *Building a GitHub App* section.

6.3.4 GitHub Actions

Since version 4.0, gidgethub provides support for working with GitHub Actions. We will not cover Actions in this tutorial.

6.4 Resources

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

6.4.1 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:

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

Activate the virtual environment. On Unix, Mac OS:

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

On Windows:

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

6.4.2 GitHub API v3 documentation

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

6.4.3 gidgethub (v 4.1.0 or up)

- Installation: `python3.7 -m pip install gidgethub`.
- gidgethub documentation
- gidgethub source code
- Owner: Brett Cannon

6.4.4 aiohttp

- Installation: `python3.7 -m pip install aiohttp`.
- aiohttp documentation
- *aiohttp* source code
- Hands-on Intro to aiohttp tutorial from PyCon US
- Owner: Andrew Svetlov

6.4.5 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))

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

6.4.6 asyncio

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

6.4.7 Heroku

Python on Heroku documentation.

6.5 GitHub API using 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's website.

6.5.1 Install gidgethub and aiohttp

Install `gidgethub` version 4.1.0 (or up) and `aiohttp` if you have not already done so. Using a virtual environment is recommended.

```
python3.7 -m pip install -U pip gidgethub==4.1.0 aiohttp
```

6.5.2 Create GitHub Personal Access Token

To get started using 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 meaningful, like: `say it with bots tutorial token`.

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.

6.5.3 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.

6.5.4 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.")

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!

6.5.5 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

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

asyncio.run(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")
        )
        response = await gh.post(
            '/repos/mariatta/strange-relationship/issues',
            data={
                'title': 'We got a problem',
```

(continues on next page)

(continued from previous page)

```
                'body': 'Use more emoji!',
            }
        )
        print(f"Issue created at {response['html_url']}")

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 my repo. Check it out: https://github.com/mariatta/strange-relationship/issues

6.5.6 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.

Download the solution for `commenting on an issue.`

6.5.7 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.

Download the solution for `closing an issue.`

6.5.8 Bonus exercise

Add reaction to an issue. You will need to pass in the `Accept` header `application/vnd.github.squirrel-girl-preview+json` in the API call. You can do this by passing it as `accept` argument when calling `gh.post`. Example:

```
await gh.post(
    url,
    data=...,
    accept="application/vnd.github.squirrel-girl-preview+json"
)
```

Note: You can only react on issues that are still **open**.

Download the solution for `reacting on an issue.`

6.6 Building a GitHub App

6.6.1 About webhooks

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

It's not so much of a "bot" that actually respond to anything. We've been running the code by ourselves.

Now let's learn about webhooks so we can build a bot.

6.6.2 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. This webservice will become our GitHub App.

There are plenty of options for hosting your webservice. For this tutorial, we will be deploying our webservice to Heroku.

6.6.3 How does authentication works with a GitHub App?

In the previous section, we used a personal access token when we executed our scripts. The token was how GitHub identifies you.

A GitHub App has a different mechanism for authentication. It is possible to build a GitHub App that can be installed in various repositories, and installed by multiple people. So how can we identify who's who in a GitHub App? Do we need to ask for everyone's access token?

When a GitHub App is installed in a repository, you get assigned an `installation_id`. When GitHub is sending you a webhook event, it will include the `installation_id` as the payload. You can then use the `installation_id` to create an `installation access token`. The `installation access token` can be used to make API calls, similar to how you use a personal access token.

More reading: https://developer.github.com/apps/building-github-apps/authenticating-with-github-apps/

How exactly can we create the `installation access token` from an `installation_id`? The documentation linked above has more details, but the process is as follows. We will be creating a JWT (JSON web token) with

the GitHub App's ID, and GitHub App's Private Key. We will then pass in the JWT and `installation_id` to GitHub, and GitHub will provide us with an `installation_access_token`.

Gidgethub provides a couple convenience functions to abstract these out. More details can be read at gidgethub.apps — Support for GitHub App.

6.6.4 Create a webservice

Let's create a webservice, that will become our bot. I've created a repository that you can fork and clone, as a starting point.

1. Go to https://github.com/Mariatta/github_app_boilerplate.
2. Fork and clone the repository:

```
$ git clone git@github.com:{your GitHub username}/github_app_boilerplate.git
```

Let's go over the repository content quickly.

Procfile This file is specifically for Heroku. Heroku will use this file to know what kind of dyno to run for your web service, and what command to run.

requirements.txt This lists the dependencies needed for the webservice. Heroku will read this file and install the dependencies.

runtime.txt This is also for Heroku. It tells Heroku which Python runtime environment to run for your webservice. We'll be using Python 3.7.6.

webservice/__main__.py This is the code of our web service. It runs an aiohttp server. I've defined a couple request handlers. The "/" endpoint (the `handle_get` coroutine) simply returns the text "Hello world".

The "/webhook" endpoint (the `webhook` coroutine) accepts a POST method. This is where we will receive all webhook events from GitHub.

You can try running the webservice locally first. Create and activate a virtual environment, and install the dependencies. From the root of the repository:

```
$ python3.7 -m venv venv
$ source venv/bin/activate
(venv) $ python3.7 -m pip install -U pip -r requirements.txt
```

Start the server:

```
(venv) $ python3.7 -m webservice
```

Note that this is the same command you see in the `Procfile`.

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 http://127.0.0.1:8080
```

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

6.6.5 Deploy the webservice to Heroku

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 `github_app_boilerplate` (assuming you forked my repo). Press "Search". Once it found the right repo, press "Connect".

Scroll down. Under **Deploy a GitHub branch**, choose "master", and click **"Deploy Branch"**. (You may also want to "Enable Automatic Deploys").

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.".

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

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

6.6.6 Create a GitHub App

Create a GitHub App by going to https://github.com/settings/apps. You can also get there by going to GitHub, click on your avatar, choose **Settings**, and **Developer Settings**.

Click the **New GitHub App** button.

You will be presented with a form. Choose a name for your app. This will become the name of your bot! (It can also be changed later). I suggest something descriptive. I already have a bot named `mariatta-bot`, so this time I will go with `mariatta-bot-again`.

Enter a description. You can leave most of the other fields empty. For this tutorial, the two important fields are: **Webhook URL** and **Webhook Secret**.

In the **Webhook URL** field, enter the url of your heroku website, ended with `/webhook`. For example `https://{yourappname}.herokuapp.com/webhook`.

In the **Webhook secret**, enter a passphrase (or any text). This secret will be used by our webservice. We need a way to know that the webhook we receive is indeed from GitHub, and it is meant for our bot. If other bot or other webservice somehow made a POST request to your endpoint, you don't really want to do anything about it. Therefore this secret should be known only by your webservice and your GitHub App (and yourself!) Whatever secret you put in here, remember it (or copy it somewhere), we will use it later.

Scroll down to the **Permissions** section. For this tutorial, set the permission for both **Issues** and **Pull requests** to **"Read & Write"**.

Scroll down to the **Subscribe to events** section. Tick the **Issues** and **Pull request** boxes.

For the question **Where can this GitHub App be installed?**, for now let's limit this to yourself, since we're still learning and developing it. You can always change this to **Any account** later on.

Click the **Create GitHub App** button!

6.6.7 Set up config vars in Heroku

We need to create the following config variables in Heroku. This is similar as if we're setting an environment variable in our Terminal. There are several values that are "secret and confidential" that we do not want to hardcode or commit to our codebase.

The first config var to create is `GH_APP_ID`. You'll see this in your GitHub App's settings page.

The next config var is `GH_SECRET`, which is the **webhook secret** from when you created the GitHub App in the previous step.

The last config var is `GH_PRIVATE_KEY`. This will be used for generating your bearer token (in `get_jwt()`). From your GitHub App's settings page, scroll down and click the **Generate Private Key** button. It will generate a private key, and automatically be downloaded as a `.pem` file. Copy the content of that file to the `GH_PRIVATE_KEY` config var.

Now that we have a webservice running, and config vars setup, we can start building our bot!

6.7 Responding to Webhook Events

Please first complete the setup in the previous section: *Building a GitHub App*.

6.7.1 Thank the maintainer for installing

Let's have a bot that **thanks the maintainer who installed your Github App**. Whenever your GitHub bot is installed, we will have the bot create an issue in the repository it was installed at, it will say something like "thanks for installing me", and then it will close the issue immediately.

We've learned how to create and close an issue earlier in *GitHub API using Command Line*. The only thing we need to implement is webhook event handler.

Let's go back to the list of GitHub events documentation. There are two events related to app installations: `installation` event and `installation_repositories` event.

Let's focus with just the `installation` event for now.

Go to the `__main__.py` file, in the webservice codebase.

I've added the following lines:

```
@router.register("installation", action="created")
async def repo_installation_added(event, gh, *args, **kwargs):
    installation_id = event.data["installation"]["id"]
    pass
```

This is where we are subscribing to the GitHub `installation` event, and specifically to the "created" 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.

It doesn't do anything now, so let's add the code.

We've said that the GitHub App was installed, we want to say thanks to the maintainer by creating an issue and then close it.

From previous example, we know how to do these tasks:

```
# creating an issue
response = await gh.post(url, data={
        'title': '...',
        'body': '...',
    })

# closing an issue (requires write permission)
await gh.patch(url,
    data={'state': 'closed'},
)
```

However, since this is a GitHub App, we can't use the personal access token. We'll need to use the installation access token, using the `get_installation_access_token` coroutine from `gidgethub.apps` module:

```
installation_access_token = await apps.get_installation_access_token(
    gh,
    installation_id=installation_id,
    app_id=os.environ.get("GH_APP_ID"),
    private_key=os.environ.get("GH_PRIVATE_KEY")
)
```

The API calls will need to be change as follows:

```
response = await gh.post(url, data={},
    oauth_token=installation_access_token["token"]
)
```

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

```
url = f"/repos/mariatta/strange-relationship/issues"
```

We do we know which repository your app was installed to?

Take a look at GitHub's installation event payload example.

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

```
{
  "action": "added",
  "repositories_added": [
    {
      "id": 186853007,
      "node_id": "MDEwOlJlcG9zaXRvcnkxODY4NTMwMDc=",
      "name": "Space",
      "full_name": "Codertocat/Space",
      "private": false
    }
  ],
  ...
}
```

Notice that the repository name is provided in the webhook, under the list of "repositories". So we can iterate on it and construct the url as follows:

```
for repository in event.data['repositories']:
    url = f"/repos/{repository['full_name']}/issues"
```

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

Take a look again at the issue event payload:

```
{
  "action": "added",
  "sender": {
    "login": "Codertocat",
       ...
}
```

The installer's username can be accessed by `event.data["sender"]["login"]`.

So now your comment message should be:

```
maintainer = event.data["sender"]["login"]
message = f"Thanks for installing me, @{maintainer}! (I'm a bot)."
```

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

```
@router.register("installation", action="created")
async def repo_installation_added(event, gh, *args, **kwargs):
    installation_id = event.data["installation"]["id"]
    installation_access_token = await apps.get_installation_access_token(
        gh,
        installation_id=installation_id,
        app_id=os.environ.get("GH_APP_ID"),
        private_key=os.environ.get("GH_PRIVATE_KEY"),
    )
    maintainer = event.data["sender"]["login"]
    message = f"Thanks for installing me, @{maintainer}! (I'm a bot)."

    for repository in event.data["repositories_added"]:
        url = f"/repos/{repository['full_name']}/issues"
        response = await gh.post(
            url,
            data={
                "title": "Mariatta's bot was installed",
                "body": message
            },
            oauth_token=installation_access_token["token"],
        )
```

Because our bot wants to be helpful, it wants to clean up after itself by closing the issue right away. How do we know the issue number that was created?

Both issue number, and the URL are returned in the response of the API call (see the documentation):

```
issue_url = response["url"]
await gh.patch(issue_url, data={"state": "closed"},
    oauth_token=installation_access_token["token"]
)
```

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

Go here for the completed solution.

Install your bot

Once deployed, you can install the GitHub App in one of your repositories and see it in action!!

From your GitHub App's settings page, click on the "Install" link on the left. Choose one repository.

Once it's done, check out the repository where you installed it to. You should see an issue created and closed immediately by the bot.

Congrats! You now have a bot in place!

Problems??

If there's any problem so far, there are a few ways you can debug this.

- Check the recent webhook deliveries under the "Advanced" link in your GitHub App settings page. You can see all the webhook events, the payload, and the status.
- Read the logs from heroku. If you have Heroku toolbelt installed, you can run:

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

- Add logs (or prints) to your code.
- Redeliver the webhook. After you made changes to your code, you don't have to re-install the App, or wait for new events to come in. You can redeliver the same webhook event that failed before.

6.7.2 Thank a new contributor for the pull request

Let's give the bot more job! Let's now have the bot **say thanks, whenever we receive a pull request**.

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

Some useful documentations:

- GitHub `pull_request` event documentation: https://developer.github.com/v3/activity/events/types/#pullrequestevent
- GitHub pull request API documentation: https://developer.github.com/v3/pulls/ Note to comments on a pull request are managed using the Issues API. Meaning you'll be use the same API as if you're commenting on an issue.

The example payload for the pull request event is here: https://developer.github.com/v3/activity/events/types/#webhook-payload-example-27

Try this on your own.

I'll give you a starting hint:

```
@router.register("pull_request", action="opened")
async def pr_opened(event, gh, *args, **kwargs):
    ...
```

How can you tell if the person is a new contributor, or an existing member of your organization? Perhaps you don't want this bot to be triggered if it is one of your co-maintainers.

In the pull_request webhook event, one of the data that was passed is the `author_association` field. It could be an `OWNER`, `MEMBER`, `CONTRIBUTOR`, or `None`, If the `author_association`

field is empty, you can guess that they are a first time contributor. (access this data as `event.data["pull_request"]["author_association"]`).

See my solution here.

6.7.3 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

See my solution on how to react to issue comments here.

6.7.4 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

Here's the solution.

6.8 What's Next?

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

However, the App 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.

6.8.1 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.

6.8.2 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.

Caching the Installation Access Token

When we receive an installation access token, it returns an expiry date. You can use the same installation access token until it becomes expired. You can think of ways to cache this token somehow, instead of requesting it each time we need it.

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.

6.9 Git Cheat Sheet

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

6.9.1 Clone a repo

```
git clone <url>
```

6.9.2 Create a new branch, and switch to it

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

6.9.3 Switching to an existing branch

```
git checkout <branchname>
```

6.9.4 Adding files to be committed

```
git add <filename>
```

6.9.5 Commit your changes

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

6.9.6 Pushing changes to remote

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

6.9.7 Rebase with a branch on remote

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