

6.5 GitHub - Scripting GitHub

Scripting GitHub

So now we’ve covered all of the major features and workflows of GitHub, but any large group or project will have customizations they may want to make or external services they may want to integrate.

Luckily for us, GitHub is really quite hackable in many ways. In this section we’ll cover how to use the GitHub hooks system and its API to make GitHub work how we want it to.

Services and Hooks

The Hooks and Services section of GitHub repository administration is the easiest way to have GitHub interact with external systems.

Services

First we’ll take a look at Services. Both the Hooks and Services integrations can be found in the Settings section of your repository, where we previously looked at adding Collaborators and changing the default branch of your project. Under the “Webhooks and Services” tab you will see something like [Services and Hooks configuration section](#).

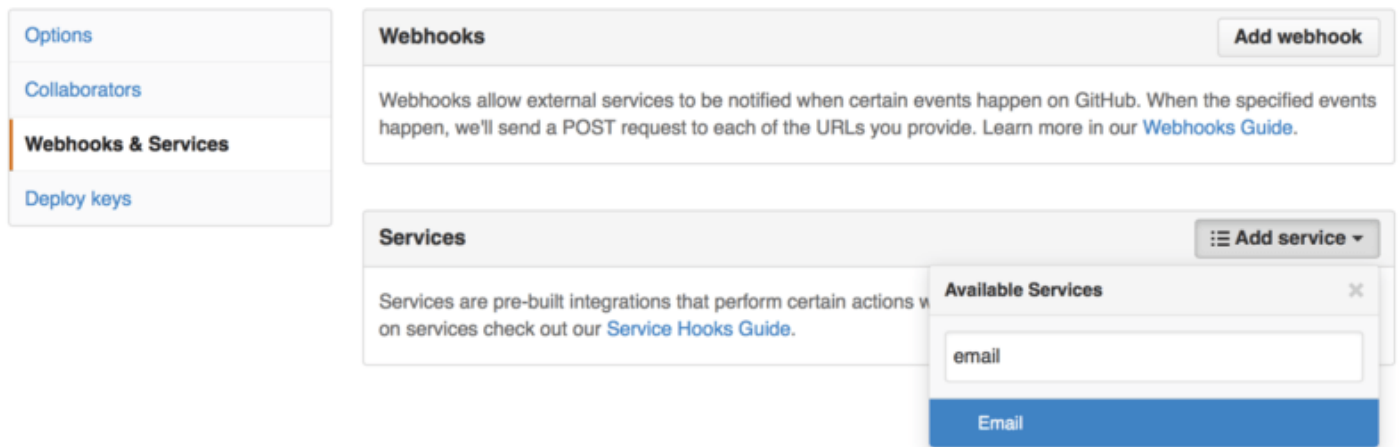


Figure 129. Services and Hooks configuration section

There are dozens of services you can choose from, most of them integrations into other commercial and open source systems. Most of them are for Continuous Integration services, bug and issue trackers, chat room systems and documentation systems. We’ll walk through setting up a very simple one, the Email hook. If you choose “email” from the “Add Service” dropdown, you’ll get a configuration screen like [Email service configuration](#).

Options

Collaborators

Webhooks & Services

Deploy keys

Services / **Add Email**

Install Notes

1. `address` whitespace separated email addresses (at most two)
2. `secret` fills out the Approved header to automatically approve the message in a read-only or moderated mailing list.
3. `send_from_author` uses the commit author email address in the From address of the email.

Address

tchacon@example.com

Secret

☐ Send from author

☒ **Active**
We will run this service when an event is triggered.

Add service

Figure 130. Email service configuration

In this case, if we hit the “Add service” button, the email address we specified will get an email every time someone pushes to the repository. Services can listen for lots of different types of events, but most only listen for push events and then do something with that data.

If there is a system you are using that you would like to integrate with GitHub, you should check here to see if there is an existing service integration available. For example, if you’re using Jenkins to run tests on your codebase, you can enable the Jenkins builtin service integration to kick off a test run every time someone pushes to your repository.

Hooks

If you need something more specific or you want to integrate with a service or site that is not included in this list, you can instead use the more generic hooks system. GitHub repository hooks are pretty simple. You specify a URL and GitHub will post an HTTP payload to that URL on any event you want.

Generally the way this works is you can setup a small web service to listen for a GitHub hook payload and then do something with the data when it is received.

To enable a hook, you click the “Add webhook” button in [Services and Hooks configuration section](#). This will bring you to a page that looks like [Web hook configuration](#).

Options
Collaborators
Webhooks & Services
Deploy keys

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

Content type

Secret

Which events would you like to trigger this webhook?

☒ Just the push event.

☐ Send me **everything**.

☐ Let me select individual events.

☒ **Active**
We will deliver event details when this hook is triggered.

Add webhook

Figure 131. Web hook configuration

The configuration for a web hook is pretty simple. In most cases you simply enter a URL and a secret key and hit “Add webhook”. There are a few options for which events you want GitHub to send you a payload for — the default is to only get a payload for the push event, when someone pushes new code to any branch of your repository.

Let’s see a small example of a web service you may set up to handle a web hook. We’ll use the Ruby web framework Sinatra since it’s fairly concise and you should be able to easily see what we’re doing.

Let’s say we want to get an email if a specific person pushes to a specific branch of our project modifying a specific file. We could fairly easily do that with code like this:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
    branch == 'ref/heads/special-branch' &&
    files.include?('special-file.txt')




    Mail.deliver do
      from 'tchacon@example.com'
      to 'tchacon@example.com'
      subject 'Scott Changed the File'
      body "ALARM"
    end
  end
end
```

end
end

Here we're taking the JSON payload that GitHub delivers us and looking up who pushed it, what branch they pushed to and what files were touched in all the commits that were pushed. Then we check that against our criteria and send an email if it matches.

In order to develop and test something like this, you have a nice developer console in the same screen where you set the hook up. You can see the last few deliveries that GitHub has tried to make for that webhook. For each hook you can dig down into when it was delivered, if it was successful and the body and headers for both the request and the response. This makes it incredibly easy to test and debug your hooks.

Recent Deliveries

	4aeae280-4e38-11e4-9bac-c130e992644b	2014-10-07 17:40:41	...
	aff20880-4e37-11e4-9089-35319435e08b	2014-10-07 17:36:21	...
	90f37680-4e37-11e4-9508-227d13b2ccfc	2014-10-07 17:35:29	...

Request

Response 200

Completed in 0.61 seconds.

Redeliver

Headers

Request URL: https://hooks.example.com/payload
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-GitHub-Event: push

Payload

```
{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bffa827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffa827f8a9e7cde00cbb0ab06a35e48...9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
    }
  ]
}
```

Figure 132. Web hook debugging information

The other great feature of this is that you can redeliver any of the payloads to test your service easily.

For more information on how to write webhooks and all the different event types you can listen for, go to the GitHub Developer documentation at <https://developer.github.com/webhooks/>.

The GitHub API

Services and hooks give you a way to receive push notifications about events that happen on your repositories, but what if you need more information about these events? What if you need to automate something like adding collaborators or labeling issues?

This is where the GitHub API comes in handy. GitHub has tons of API endpoints for doing nearly anything you can do on the website in an automated fashion. In this section we'll learn how to authenticate and connect to the API, how to comment on an issue and how to change the status of a Pull Request through the API.

Basic Usage

The most basic thing you can do is a simple GET request on an endpoint that doesn't require authentication. This could be a user or read-only information on an open source project. For example, if we want to know more about a user named "schacon", we can run something like this:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
  # ...
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

There are tons of endpoints like this to get information about organizations, projects, issues, commits — just about anything you can publicly see on GitHub. You can even use the API to render arbitrary Markdown or find a .gitignoretemplate.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see https://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
```

Commenting on an Issue

However, if you want to do an action on the website such as comment on an Issue or Pull Request or if you want to view or interact with private content, you'll need to authenticate.

There are several ways to authenticate. You can use basic authentication with just your username and password, but generally it's a better idea to use a personal access token. You can generate this from the "Applications" tab of your settings page.

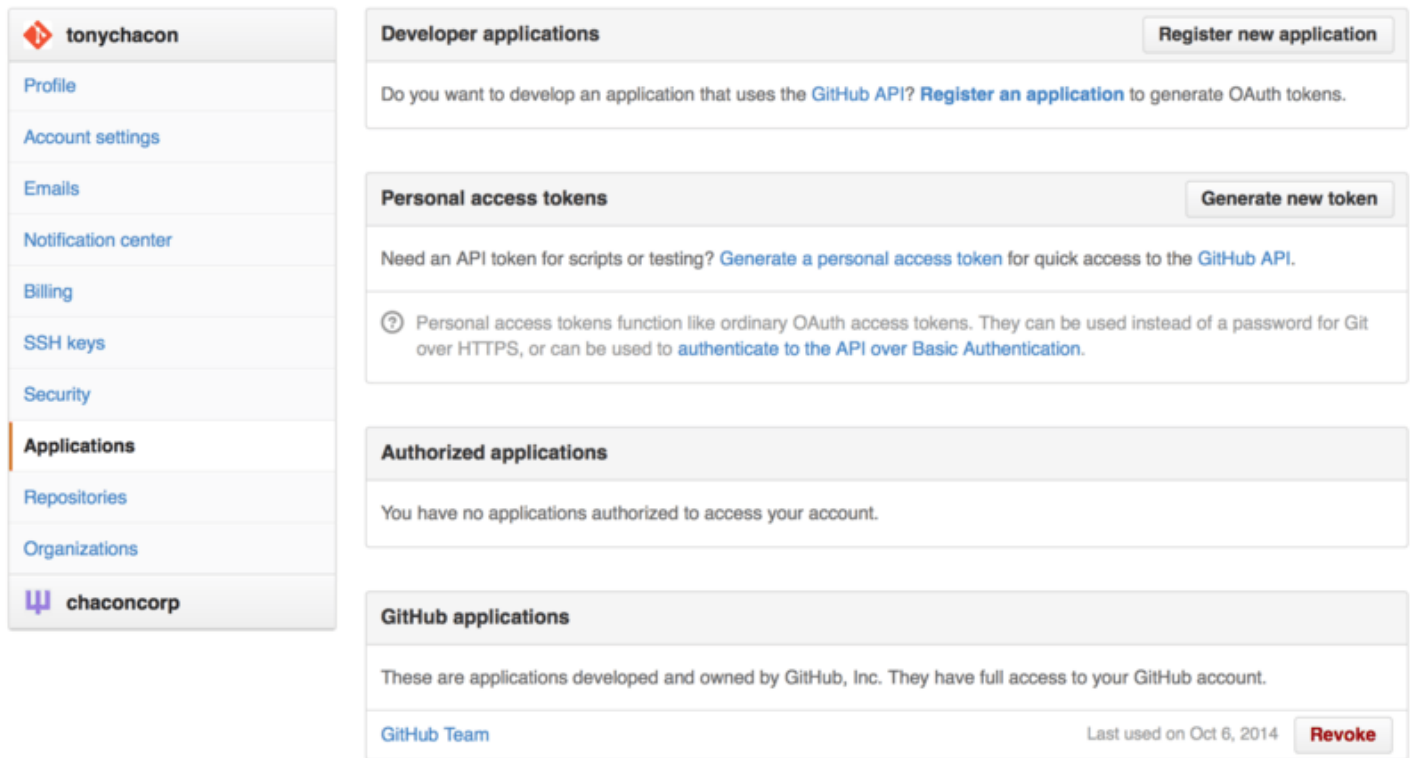


Figure 133. Generate your access token from the “Applications” tab of your settings page

It will ask you which scopes you want for this token and a description. Make sure to use a good description so you feel comfortable removing the token when your script or application is no longer used.

GitHub will only show you the token once, so be sure to copy it. You can now use this to authenticate in your script instead of using a username and password. This is nice because you can limit the scope of what you want to do and the token is revocable.

This also has the added advantage of increasing your rate limit. Without authenticating, you will be limited to 60 requests per hour. If you authenticate you can make up to 5,000 requests per hour.

So let’s use it to make a comment on one of our issues. Let’s say we want to leave a comment on a specific issue, Issue #6. To do so we have to do an HTTP POST request to `repos/<user>/<repo>/issues/<num>/comments` with the token we just generated as an Authorization header.

```
$ curl -H "Content-Type: application/json" \
-H "Authorization: token TOKEN" \
--data '{"body":"A new comment, :+1:"}' \
https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

Now if you go to that issue, you can see the comment that we just successfully posted as in [A comment posted from the GitHub API](#).

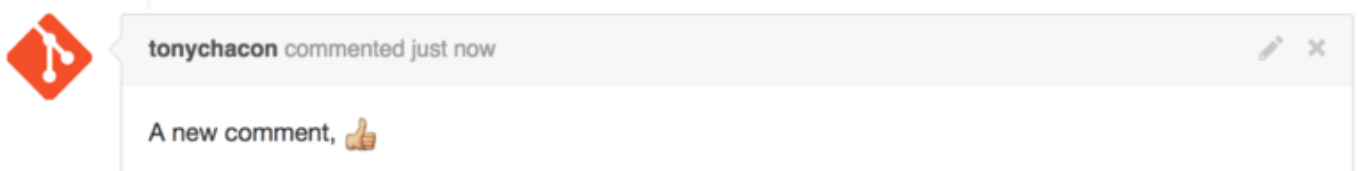


Figure 134. A comment posted from the GitHub API

You can use the API to do just about anything you can do on the website — creating and setting milestones, assigning people to Issues and Pull Requests, creating and changing labels, accessing commit data, creating new commits and branches, opening, closing or merging Pull Requests, creating and editing teams, commenting on lines of code in a Pull Request, searching the site and on and on.

Changing the Status of a Pull Request

There is one final example we'll look at since it's really useful if you're working with Pull Requests. Each commit can have one or more statuses associated with it and there is an API to add and query that status.

Most of the Continuous Integration and testing services make use of this API to react to pushes by testing the code that was pushed, and then report back if that commit has passed all the tests. You could also use this to check if the commit message is properly formatted, if the submitter followed all your contribution guidelines, if the commit was validly signed — any number of things.

Let's say you set up a webhook on your repository that hits a small web service that checks for a Signed-off-by string in the commit message.

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

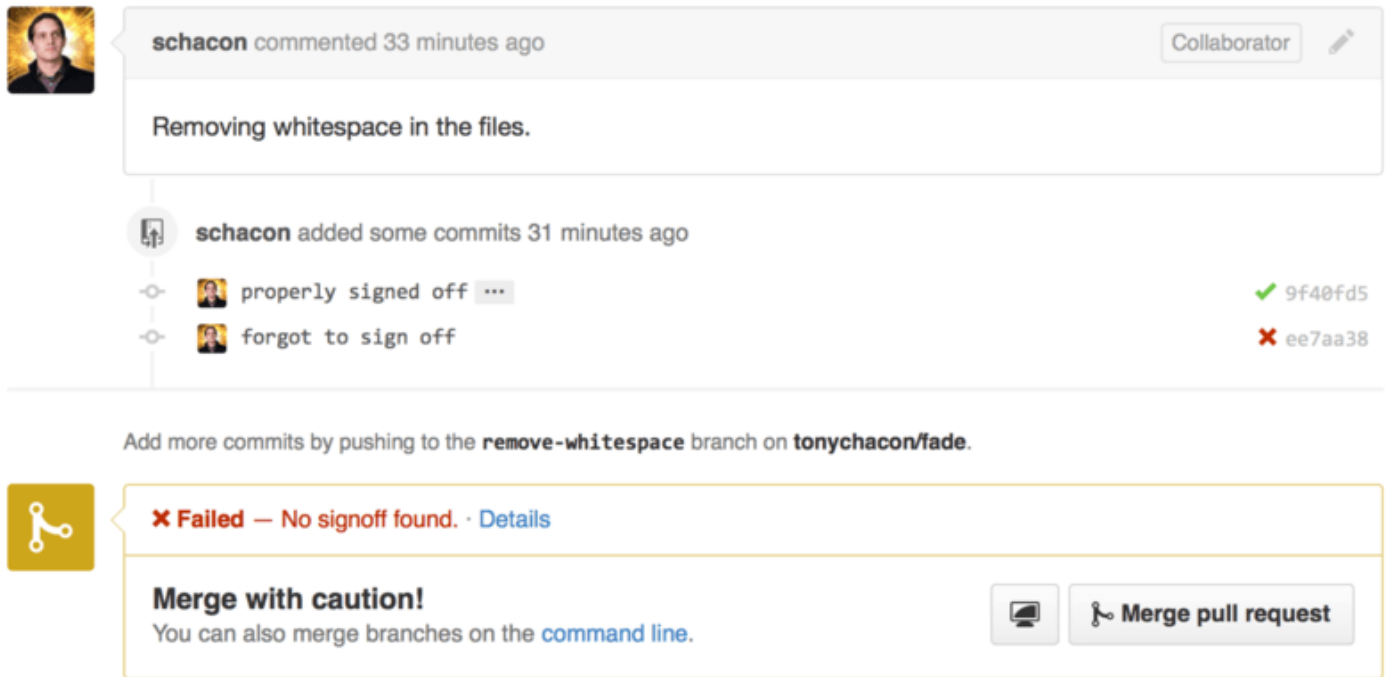
    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"      => state,
      "description" => description,
      "target_url" => "http://example.com/how-to-signoff",
      "context"    => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type' => 'application/json',
        'User-Agent'   => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

Hopefully this is fairly simple to follow. In this web hook handler we look through each commit that was just pushed, we look for the string 'Signed-off-by' in the commit message and finally we POST via HTTP to the `/repos/<user>/<repo>/statuses/<commit_sha>` API endpoint with the status.

In this case you can send a state ('success', 'failure', 'error'), a description of what happened, a target URL the user can go to for more information and a “context” in case there are multiple statuses for a single commit. For example, a testing service may provide a status and a validation service like this may also provide a status — the “context” field is how they're differentiated.

If someone opens a new Pull Request on GitHub and this hook is set up, you may see something like [Commit status via the API](#).



The screenshot displays a GitHub commit status interface. At the top, a comment from 'schacon' 33 minutes ago states 'Removing whitespace in the files.' Below this, a status bar indicates 'schacon added some commits 31 minutes ago'. Two commit entries are listed: one 'properly signed off' with a green checkmark and hash '9f40fd5', and another 'forgot to sign off' with a red cross and hash 'ee7aa38'. A message prompts to 'Add more commits by pushing to the remove-whitespace branch on tonychacon/fade.' At the bottom, a yellow box shows a 'Failed' status with the message 'No signoff found.' and a 'Merge pull request' button.

schacon commented 33 minutes ago Collaborator

Removing whitespace in the files.

schacon added some commits 31 minutes ago

- properly signed off ... ✓ 9f40fd5
- forgot to sign off ✗ ee7aa38

Add more commits by pushing to the **remove-whitespace** branch on **tonychacon/fade**.

✗ Failed — No signoff found. · Details

Merge with caution! You can also merge branches on the [command line](#).

Merge pull request

Figure 135. Commit status via the API

You can now see a little green check mark next to the commit that has a “Signed-off-by” string in the message and a red cross through the one where the author forgot to sign off. You can also see that the Pull Request takes the status of the last commit on the branch and warns you if it is a failure. This is really useful if you’re using this API for test results so you don’t accidentally merge something where the last commit is failing tests.

Octokit

Though we’ve been doing nearly everything through `curl` and simple HTTP requests in these examples, several open-source libraries exist that make this API available in a more idiomatic way. At the time of this writing, the supported languages include Go, Objective-C, Ruby, and .NET. Check out <https://github.com/octokit> for more information on these, as they handle much of the HTTP for you.

Hopefully these tools can help you customize and modify GitHub to work better for your specific workflows. For complete documentation on the entire API as well as guides for common tasks, check out <https://developer.github.com>.

[prev](#) | [next](#)