Creating a pre-receive hook script

In this article

Use pre-receive hook scripts to create requirements for accepting or rejecting a push based on the contents.

You can see examples of pre-receive hooks for GitHub Enterprise Server in the `github/platform-samples` repository.

Writing a pre-receive hook script

A pre-receive hook script executes in a pre-receive hook environment on your GitHub Enterprise Server instance. When you create a pre-receive hook script, consider the available input, output, exit status, and environment variables.

Input (`stdin`)

After a push occurs and before any refs are updated for the remote repository, the `git-receive-pack` process on your GitHub Enterprise Server instance invokes the pre-receive hook script. Standard input for the script, `stdin`, is a string containing a line for each ref to update. Each line contains the old object name for the ref, the new object name for the ref, and the full name of the ref.

```
<old-value> SP <new-value> SP <ref-name> LF
```

This string represents the following arguments.

Argument	Description
`<old-value>`	Old object name stored in the ref. When you create a new ref, the value is 40 zeroes.
`<new-value>`	New object name to be stored in the ref. When you delete a ref, the value is 40 zeroes.
`<ref-name>`	The full name of the ref.

For more information about `git-receive-pack`, see "git-receive-pack" in the Git

documentation. For more information about refs, see "Git References" in *Pro Git*.

Output (`stdout`)

The standard output for the script, `stdout`, is passed back to the client. Any `echo` statements will be visible to the user on the command line or in the user interface.

Exit status

The exit status of a pre-receive script determines if the push will be accepted.

Exit-status value	Action
0	The push will be accepted.
non-zero	The push will be rejected.

Environment variables

In addition to the standard input for your pre-receive hook script, `stdin`, GitHub Enterprise Server makes the following variables available in the Bash environment for your script's execution. For more information about `stdin` for your pre-receive hook script, see "Input (`stdin`)."

Different environment variables are available to your pre-receive hook script depending on what triggers the script to run.

- Always available
- Available for pushes from the web interface or API
- Available for pull request merges
- Available for pushes using SSH authentication

Always available

The following variables are always available in the pre-receive hook environment.

Variable	Description	Example value
`$GIT_DIR`	Path to the remote repository on the instance	/data/user/repositories/a/ab/a1/b2/34/100001234/1234.git
`$GIT_PUSH_OPTION_COUNT`	The number of push options that were sent by the client with `--push-option`. For more information, see "git-push" in the Git documentation.	1
`$GIT_PUSH_OPTION_N`	Where N is an integer starting at 0, this variable contains the push option string that was sent by the client. The first option that was sent is stored in `GIT_PUSH_OPTION_0`, the second option that was sent is stored in `GIT_PUSH_OPTION_1`, and so on. For more information about push options, see "git-push" in the Git documentation.	abcd

`$GIT_USER_AGENT`	User-agent string sent by the Git client that pushed the changes	git/2.0.0
`$GITHUB_REPO_NAME`	Name of the repository being updated in NAME/OWNER format	octo-org/hello-enterprise
`$GITHUB_REPO_PUBLIC`	Boolean representing whether the repository being updated is public	• true: Repository's visibility is public • false: Repository's visibility is private or internal
`$GITHUB_USER_IP`	IP address of client that initiated the push	192.0.2.1
`$GITHUB_USER_LOGIN`	Username for account that initiated the push	octocat

Available for pushes from the web interface or API

The `$GITHUB_VIA` variable is available in the pre-receive hook environment when the ref update that triggers the hook occurs via either the web interface or the API for GitHub Enterprise Server. The value describes the action that updated the ref.

Value	Action	More information
`auto-merge deployment api`	Automatic merge of the base branch via a deployment created with the API	"Deployments" in the REST API documentation
`blob#save`	Change to a file's contents in the web interface	"Editing files"
`branch merge api`	Merge of a branch via the API	"Branches" in the REST API documentation
`branches page delete button`	Deletion of a branch in the web interface	"Creating and deleting branches within your repository"
`git refs create api`	Creation of a ref via the API	"Git database" in the REST API documentation
`git refs delete api`	Deletion of a ref via the API	"Git database" in the REST API documentation
`git refs update api`	Update of a ref via the API	"Git database" in the REST API documentation
`git repo contents api`	Change to a file's contents via the API	"Repositories" in the REST API documentation
`merge`	Merge of a pull request using auto-merge	"Automatically merging a pull request"

`merge base into head`	Update of the topic branch from the base branch when the base branch requires strict status checks (via **Update branch** in a pull request, for example)	"About protected branches"
`pull request branch delete but`	Deletion of a topic branch from a pull request in the web interface	"Deleting and restoring branches in a pull request"
`pull request branch undo buttc`	Restoration of a topic branch from a pull request in the web interface	"Deleting and restoring branches in a pull request"
`pull request merge api`	Merge of a pull request via the API	"Pulls" in the REST API documentation
`pull request merge button`	Merge of a pull request in the web interface	"Merging a pull request"
`pull request revert button`	Revert of a pull request	"Reverting a pull request"
`releases delete button`	Deletion of a release	"Managing releases in a repository"
`stafftools branch restore`	Restoration of a branch from the site admin dashboard	"Site admin dashboard"
`tag create api`	Creation of a tag via the API	"Git database" in the REST API documentation
`slumlord (#SHA)`	Commit via Subversion	"Support for Subversion clients"
`web branch create`	Creation of a branch via the web interface	"Creating and deleting branches within your repository"

Available for pull request merges

The following variables are available in the pre-receive hook environment when the push that triggers the hook is a push due to the merge of a pull request.

Variable	Description	Example value
`$GITHUB_PULL_REQUEST_AUTHOR_LO`	Username of account that authored the pull request	octocat
`$GITHUB_PULL_REQUEST_HEAD`	The name of the pull request's topic branch, in the format `USERNAME:BRANCH`	octocat:fix-bug

`$GITHUB_PULL_REQUEST_BASE`	The name of the pull request's base branch, in the format `USERNAME:BRANCH`	octocat:main

Available for pushes using SSH authentication

Variable	Description	Example value
`$GITHUB_PUBLIC_KEY_FINGERPRINT`	The public key fingerprint for the user who pushed the changes	a1:b2:c3:d4:e5:f6:g7:h8:i9:j0:k1:l2:m3:n4:o5:p6

Setting permissions and pushing a pre-receive hook to GitHub Enterprise Server

A pre-receive hook script is contained in a repository on your GitHub Enterprise Server instance. A site administrator must take into consideration the repository permissions and ensure that only the appropriate users have access.

We recommend consolidating hooks to a single repository. If the consolidated hook repository is public, the `README.md` can be used to explain policy enforcements. Also, contributions can be accepted via pull requests. However, pre-receive hooks can only be added from the default branch. For a testing workflow, forks of the repository with configuration should be used.

1. For Mac users, ensure the scripts have execute permissions:

   ```
   $ sudo chmod +x SCRIPT_FILE.sh
   ```

   For Windows users, ensure the scripts have execute permissions:

   ```
   git update-index --chmod=+x SCRIPT_FILE.sh
   ```

2. Commit and push to the designated repository for pre-receive hooks on your GitHub Enterprise Server instance.

   ```
   $ git commit -m "YOUR COMMIT MESSAGE"
   $ git push
   ```

3. Create the pre-receive hook on the GitHub Enterprise Server instance.

Testing pre-receive scripts locally

You can test a pre-receive hook script locally before you create or update it on your GitHub Enterprise Server instance. One method is to create a local Docker environment to act as a remote repository that can execute the pre-receive hook.

1. Ensure Docker is installed locally.
2. Create a file called `Dockerfile.dev` containing:

   ```
   FROM gliderlabs/alpine:3.3
   RUN \
     apk add --no-cache git openssh bash && \
   ```

```
  ssh-keygen -A && \
  sed -i "s/#AuthorizedKeysFile/AuthorizedKeysFile/g" /etc/ssh/sshd_config && \
  adduser git -D -G root -h /home/git -s /bin/bash && \
  passwd -d git && \
  su git -c "mkdir /home/git/.ssh && \
  ssh-keygen -t ed25519 -f /home/git/.ssh/id_ed25519 -P '' && \
  mv /home/git/.ssh/id_ed25519.pub /home/git/.ssh/authorized_keys && \
  mkdir /home/git/test.git && \
  git --bare init /home/git/test.git"

VOLUME ["/home/git/.ssh", "/home/git/test.git/hooks"]
WORKDIR /home/git

CMD ["/usr/sbin/sshd", "-D"]
```

3. Create a test pre-receive script called `always_reject.sh`. This example script will reject all pushes, which is useful for locking a repository:

```
#!/usr/bin/env bash

echo "error: rejecting all pushes"
exit 1
```

4. Ensure the `always_reject.sh` scripts has execute permissions:

```
$ chmod +x always_reject.sh
```

5. From the directory containing `Dockerfile.dev`, build an image:

```
$ docker build -f Dockerfile.dev -t pre-receive.dev .
> Sending build context to Docker daemon 3.584 kB
> Step 1 : FROM gliderlabs/alpine:3.3
>  ---> 8944964f99f4
> Step 2 : RUN apk add --no-cache git openssh bash && ssh-keygen -A && sed -i "
>  ---> Running in e9d79ab3b92c
> fetch http://alpine.gliderlabs.com/alpine/v3.3/main/x86_64/APKINDEX.tar.gz
> fetch http://alpine.gliderlabs.com/alpine/v3.3/community/x86_64/APKINDEX.tar.
....truncated output....
> OK: 34 MiB in 26 packages
> ssh-keygen: generating new host keys: RSA DSA ECDSA ED25519
> Password for git changed by root
> Generating public/private ed25519 key pair.
> Your identification has been saved in /home/git/.ssh/id_ed25519.
> Your public key has been saved in /home/git/.ssh/id_ed25519.pub.
....truncated output....
> Initialized empty Git repository in /home/git/test.git/
> Successfully built dd8610c24f82
```

6. Run a data container that contains a generated SSH key:

```
$ docker run --name data pre-receive.dev /bin/true
```

7. Copy the test pre-receive hook `always_reject.sh` into the data container:

```
$ docker cp always_reject.sh data:/home/git/test.git/hooks/pre-receive
```

8. Run an application container that runs `sshd` and executes the hook. Take note of

the container id that is returned:

```
$ docker run -d -p 52311:22 --volumes-from data pre-receive.dev
> 7f888bc700b8d23405dbcaf039e6c71d486793cad7d8ae4dd184f4a47000bc58
```

9. Copy the generated SSH key from the data container to the local machine:

```
$ docker cp data:/home/git/.ssh/id_ed25519 .
```

10. Modify the remote of a test repository and push to the `test.git` repo within the Docker container. This example uses `git@github.com:octocat/Hello-World.git` but you can use any repository you want. This example assumes your local machine (127.0.0.1) is binding port 52311, but you can use a different IP address if docker is running on a remote machine.

```
$ git clone git@github.com:octocat/Hello-World.git
$ cd Hello-World
$ git remote add test git@127.0.0.1:test.git
$ GIT_SSH_COMMAND="ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking
> Warning: Permanently added '[192.168.99.100]:52311' (ECDSA) to the list of kr
> Counting objects: 7, done.
> Delta compression using up to 4 threads.
> Compressing objects: 100% (3/3), done.
> Writing objects: 100% (7/7), 700 bytes | 0 bytes/s, done.
> Total 7 (delta 0), reused 7 (delta 0)
> remote: error: rejecting all pushes
> To git@192.168.99.100:test.git
>  ! [remote rejected] main -> main (pre-receive hook declined)
> error: failed to push some refs to 'git@192.168.99.100:test.git'
```

Notice that the push was rejected after executing the pre-receive hook and echoing the output from the script.

Further reading

- "Customizing Git - An Example Git-Enforced Policy" from the *Pro Git website*