Deploy a static site with GitHub Actions

Dec 26, 2019 | *9-minute read*

Overview

In this post, I'll show you how I automatically build and deploy this blog using GitHub Actions, GitHub's free continuous integration platform. I use Hugo to generate a static site and copy it to my web server using rsync over SSH. I run nginx on my server. There are already great tutorials on how to set up Hugo and how to set up nginx. Really, this process applies to any static site deployment to any web server that you have SSH access to.

I think that this process is simple enough that it is worth setting up yourself. However, you can find existing GitHub Actions that help accomplish rsync deployments over SSH.

You will need SSH access to the machine you deploy to so that you can configure it and to use rsync over SSH. The website source is kept in a repository on GitHub. The GitHub Action automatically builds it and syncs the files to a web server when new commits are pushed to the master branch.

Creating a workflow

Set up a GitHub Actions workflow by creating the file `.github/workflows/my_workflow.yml` within your repository. Start by defining the event that launches the workflow:

```
1 name: Deploy
2 on: push
```

The name of the workflow is arbitrary. You have specified that this action should run when you push a commit.

Next, define a job:

```
3  jobs:
4    my_job:
5      name: Deploy Job
6      runs-on: ubuntu-latest
7      steps:
8      - uses: actions/checkout@v1
9      - name: List files
10       run: ls
```

Jobs are larger tasks that can logically run in parallel. They can optionally have dependencies on other jobs. Steps are sequential tasks that make up a job. The job you defined has two steps:

1. Use the pre-defined `checkout` action to checkout your latest code. This action is defined by the GitHub repository `actions/checkout` and included using `uses`.
2. Define a step which will list the files that were checked out. This step uses `run` to define a shell script inline.

When you push this to a GitHub repository, you can watch it run from the repository's Actions tab.

Next, write a useful step that runs a Hugo build. You can install Hugo using snap, as recommended by the Hugo install page. Replace the `ls` step:

```
7      steps:
8      - uses: actions/checkout@v1
9      - name: Hugo build
10       run: |
11         sudo snap install hugo --channel=extended
12         hugo
```

Now, your GitHub Actions job will automatically build the Hugo site. Next, you will need to upload the built website to your web server.

Configuring the server

This step entails configuring users and file permissions for your web data. If you already have this set up, you can skip this step. However, you may not if you have only done a basic nginx install.

The base directory in which your web files are stored (e.g. `/var/www`) is your "web root." There are many ways to successfully manage file permissions on your web root. I will describe one method here. I think that this method is simple while still providing good security.

If you host multiple websites within your web root, you may optionally create one user for each subdirectory of your web root.

First, you should be familiar with Unix file modes. See also the "Permissions" section in `man 7 path_resolution`.

The strategy

The goal is to allow write access to some user account that will be used to edit the web files, and to allow only read access to nginx. Let a particular user (e.g. `www-data` or `my-rsync-user`) own the web root. Set the group of the web root to `nginx` (or some other group that your web server user is a part of.) Set the file mode of the web root to `750`. Log in as the owner of the web root to edit it.

The detailed steps are as follows. Run them as root.

1. Create a user with `adduser --disabled-password ${NEW_USER}`
2. Set the owner and group of your web root with `chown ${NEW_USER}:nginx /var/www`. If you are using some other group, substitute it for "nginx".
3. Set file permissions on the web root with `chmod 750 /var/www`

Creating `authorized_keys`

To make future steps easier, you should create an `authorized_keys` file for whatever user you will use for rsync. Do the following:

```
export RSYNC_USER=my_rsync_user
sudo mkdir -p /home/${RSYNC_USER}/.ssh
sudo touch /home/${RSYNC_USER}/.ssh/authorized_keys
sudo chown -R ${RSYNC_USER} /home/${RSYNC_USER}/.ssh
sudo chmod 600 /home/${RSYNC_USER}/.ssh/authorized_keys
```

SSH Access

In the previous step, you configured your server to allow some user to write to your web root. In this step, you will enable your GitHub Actions worker to upload files as this user via SSH.

I chose to upload my Hugo site using `rsync`, as the Hugo website recommends. I will describe how to set up SSH access for the GitHub Actions worker so that you can `rsync` securely over SSH. You will generate a new SSH key pair, providing the private key to your Actions workflow and providing the public key to your web server to authorize the holder of the private key to connect via SSH.

Generating an SSH key pair

Generate a new key pair on your personal machine.

```
ssh-keygen
```

You could do some Googling to decide what cipher settings to use for this. You could also check this answer on StackExchange. Running `ssh-keygen` will generate a private key and a public key (ending in `.pub`) in whatever location you specified.

Authorizing the key

You need to tell the SSH daemon that machines are authorized to log in to your rsync user using the key pair you just generated. You will do this by adding the

public key to the `authorized_keys` file. By default, this is located at `~/.ssh/authorized_keys`. If, for some reason, you need to relocate this file, you can change the `AuthorizedKeysFile` option in your sshd configuration.

You need to append the public key (ending in `.pub`) to the `authorized_keys` file. There are several ways to do this. The most simple is `ssh-copy-id`, but this requires you to be able to log in as your rsync user. If you have login disabled for that user, you can use your admin user to manually append the file. You should create the file and set its permissions in the Creating `authorized-keys` step, as it is easier to do there.

```
1  cat ${MY_KEY}.pub | ssh ${USER}@${HOST} "sudo tee -a /home/${RSYNC_U
```

Replace the path to your public key, your admin username, your hostname, and your rsync username.

Finally, you need to securely provide the secret key to the GitHub Actions worker.

GitHub repository secrets

There are two secrets involved which must be managed securely. The first the is private half of your recently-generated key pair. It is the file that does not end with `.pub`. The second is the SSH fingerprint of your web server. You should store both as secrets in your GitHub repository. In reality, the fingerprint is just another form of the public key of your server. However, storing it as a secret ensures that you can't be tricked into changing it with a commit.

Secret key

First, let's upload the secret key. Navigate to the Secrets page for your repository and create a new secret called, for example, `deploy_key`. Unfortunately, the interface (at the time of writing) is simply a text box. So, `cat` your `~/.ssh/my_rsync_key` and copy it into the secret value (or something to this effect.) Since we have authorized this key by appending the public half to `authorized_keys`, it will allow the Actions worker to SSH into your web server using your rsync account.

known_hosts

When the Actions worker tries to connect to your web server via SSH, `ssh` will verify the fingerprint of the server against its `~/.ssh/known_hosts` file to help mitigate man-in-the-middle attacks. Normally, when you SSH to a host for the first time, you are interactively prompted to add it to your `known_hosts` file. However, your Actions worker always runs on a new virtual machine instance. You could have it generate a `known_hosts` file each run, but that would be equivalent to bypassing the check entirely. If you don't care, go for it. Instead, you can save the fingerprints for your web server and store them as another secret. It will later be used to create the worker's `known_hosts` file. To generate fingerprints you can use `ssh-keyscan`:

```
1  ssh-keyscan ${HOST}
```

Create a new GitHub secret called `known_hosts` and paste in the output of `ssh-keyscan`.

Rsync deployment

Everything is in place to perform the deployment with `rsync`. You can access your newly-created secrets in your GitHub action to set up SSH Access to your web server.

Set up SSH access

First, the two secrets are exposed as environment variables. Then, you can use the secrets to create the `known_hosts` file and add the private key:

```
7      steps:
8      - uses: actions/checkout@v1
9      - name: Hugo build
10       run: |
11         sudo snap install hugo --channel=extended
12         hugo
13     - name: SSH setup
14       env:
           DEPLOY_KEY: ${{ secrets.deploy_key }}
```

```
15          KNOWN_HOSTS: ${{ secrets.known_hosts }}
16        run: |
17          mkdir -p ~/.ssh
18          echo "${KNOWN_HOSTS}" >> ~/.ssh/known_hosts
19          echo "${DEPLOY_KEY}" > ~/.ssh/my_rsync_key
20          echo "IdentityFile ~/.ssh/my_rsync_key" >> ~/.ssh/config
21          chmod -R 700 ~/.ssh
22
```

Set up rsync deployment

The final step is to use rsync to upload the files to the web server. Add a new step like this:

```
22      - name: Rsync deployment
23        run: |
24                  rsync -avz -e ssh --delete public/ ${RSYNC_USER}@${
```

This step runs `rsync` using `ssh`, uploading the contents of your local `public` directory (containing your website files) to the web root of your web server.

Completed workflow definition

That's it! Below is the entire Actions workflow file. Make sure to replace your rsync username and the web server's hostname.

```
name: Deploy
on:
  push:
    branches:
    - master

jobs:
  deploy_job:
    name: Deploy Job
```

```
    runs-on: ubuntu-18.04
    steps:
    - uses: actions/checkout@v1
    - name: Hugo build
      run: |
        sudo snap install hugo --channel=extended
        hugo
    - name: SSH setup
      env:
        DEPLOY_KEY: ${{ secrets.deploy_key }}
        KNOWN_HOSTS: ${{ secrets.known_hosts }}
      run: |
        mkdir -p ~/.ssh
        echo "${KNOWN_HOSTS}" >> ~/.ssh/known_hosts
        echo "${DEPLOY_KEY}" > ~/.ssh/my_rsync_key
        echo "IdentityFile ~/.ssh/my_rsync_key" >> ~/.ssh/config
        chmod -R 700 ~/.ssh
    - name: Rsync deployment
      run: |
                rsync -avz -e ssh --delete public/ ${RSYNC_USER}@${HOST}
```