Managing deploy keys

In this article

Learn different ways to manage SSH keys on your servers when you automate deployment scripts and which way is best for you.

You can manage SSH keys on your servers when automating deployment scripts using SSH agent forwarding, HTTPS with OAuth tokens, deploy keys, or machine users.

SSH agent forwarding 🔗

In many cases, especially in the beginning of a project, SSH agent forwarding is the quickest and simplest method to use. Agent forwarding uses the same SSH keys that your local development computer uses.

Pros of SSH agent forwarding 🔗

- You do not have to generate or keep track of any new keys.
- There is no key management; users have the same permissions on the server that they do locally.
- No keys are stored on the server, so in case the server is compromised, you don't need to hunt down and remove the compromised keys.

Cons of SSH agent forwarding 🔗

- Users **must** SSH in to deploy; automated deploy processes can't be used.
- SSH agent forwarding can be troublesome to run for Windows users.

Set up SSH agent forwarding 🔗

1. Turn on agent forwarding locally. See our guide on SSH agent forwarding for more information.

2. Set your deploy scripts to use agent forwarding. For example, on a bash script, enabling agent forwarding would look something like this: `ssh -A serverA 'bash -s' < deploy.sh`

HTTPS cloning with OAuth tokens 🔗

If you don't want to use SSH keys, you can use HTTPS with OAuth tokens.

Pros of HTTPS cloning with OAuth tokens 🔗

- Anyone with access to the server can deploy the repository.
- Users don't have to change their local SSH settings.
- Multiple tokens (one for each user) are not needed; one token per server is enough.
- A token can be revoked at any time, turning it essentially into a one-use password.

Cons of HTTPS cloning with OAuth tokens 🔗

- You must make sure that you configure your token with the correct access scopes.
- Tokens are essentially passwords, and must be protected the same way.

Set up HTTPS cloning with OAuth tokens 🔗

See [our guide on creating a personal access token](#).

Deploy keys 🔗

You can launch projects from a repository on GitHub.com to your server by using a deploy key, which is an SSH key that grants access to a single repository. GitHub attaches the public part of the key directly to your repository instead of a personal account, and the private part of the key remains on your server. For more information, see "[Delivering deployments](#)."

Deploy keys with write access can perform the same actions as an organization member with admin access, or a collaborator on a personal repository. For more information, see "[Repository roles for an organization](#)" and "[Permission levels for a personal account repository](#)."

Pros of deploy keys 🔗

- Anyone with access to the repository and server has the ability to deploy the project.
- Users don't have to change their local SSH settings.
- Deploy keys are read-only by default, but you can give them write access when adding them to a repository.

Cons of deploy keys 🔗

- Deploy keys only grant access to a single repository. More complex projects may have many repositories to pull to the same server.
- Deploy keys are usually not protected by a passphrase, making the key easily accessible if the server is compromised.
- If the user who created the deploy key is removed from the repository, the deploy key will still be active as it isn't tied to the specific user, but rather to the repository.

Set up deploy keys 🔗

1. [Run the `ssh-keygen` procedure](#) on your server, and remember where you save the generated public and private rsa key pair.

2. On GitHub.com, navigate to the main page of the repository.

3. Under your repository name, click ⚙ **Settings**. If you cannot see the "Settings" tab,

select the ••• dropdown menu, then click **Settings**.
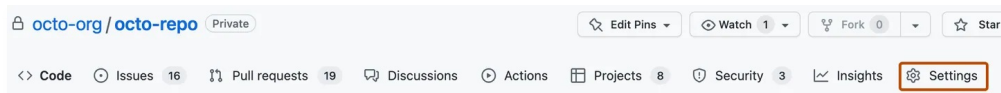
4. In the sidebar, click **Deploy Keys**.

5. Click **Add deploy key**.

6. In the "Title" field, provide a title.

7. In the "Key" field, paste your public key.

8. Select **Allow write access** if you want this key to have write access to the repository. A deploy key with write access lets a deployment push to the repository.

9. Click **Add key**.

Using multiple repositories on one server 🔗

If you use multiple repositories on one server, you will need to generate a dedicated key pair for each one. You can't reuse a deploy key for multiple repositories.

In the server's SSH configuration file (usually `~/.ssh/config`), add an alias entry for each repository. For example:

```
Host github.com-repo-0
        Hostname github.com
        IdentityFile=/home/user/.ssh/repo-0_deploy_key

Host github.com-repo-1
        Hostname github.com
        IdentityFile=/home/user/.ssh/repo-1_deploy_key
```

- `Host github.com-repo-0` - The repository's alias.
- `Hostname github.com` - Configures the hostname to use with the alias.
- `IdentityFile=/home/user/.ssh/repo-0_deploy_key` - Assigns a private key to the alias.

You can then use the hostname's alias to interact with the repository using SSH, which will use the unique deploy key assigned to that alias. For example:

```
git clone git@github.com-repo-1:OWNER/repo-1.git
```

GitHub App installation access tokens 🔗

If your server needs to access repositories across one or more organizations, you can use a GitHub App to define the access you need, and then generate *tightly-scoped*, installation access tokens from that GitHub App. The installation access tokens can be scoped to single or multiple repositories, and can have fine-grained permissions. For example, you can generate a token with read-only access to a repository's contents.

Since GitHub Apps are a first class actor on GitHub, the installation access tokens are decoupled from any GitHub user, which makes them comparable to "service tokens". Additionally, installation access tokens have dedicated rate limits that scale with the size of the organizations that they act upon. For more information, see [Rate limits for GitHub Apps](#).

Pros of installation access tokens ⊘

- Tightly-scoped tokens with well-defined permission sets and expiration times (1 hour, or less if revoked manually using the API).
- Dedicated rate limits that grow with your organization.
- Decoupled from GitHub user identities, so they do not consume any licensed seats.
- Never granted a password, so cannot be directly signed in to.

Cons of installation access tokens ⊘

- Additional setup is needed to create the GitHub App.
- Installation access tokens expire after 1 hour, and so need to be re-generated, typically on-demand using code.

Set up installation access tokens ⊘

1. Determine if your GitHub App should be public or private. If your GitHub App will only act on repositories within your organization, you likely want it private.

2. Determine the permissions your GitHub App requires, such as read-only access to repository contents.

3. Create your GitHub App via your organization's settings page. For more information, see [Creating a GitHub App](#).

4. Note your GitHub App `id`.

5. Generate and download your GitHub App's private key, and store this safely. For more information, see [Generating a private key](#).

6. Install your GitHub App on the repositories it needs to act upon, optionally you may install the GitHub App on all repositories in your organization.

7. Identify the `installation_id` that represents the connection between your GitHub App and the organization repositories it can access. Each GitHub App and organization pair have at most a single `installation_id`. You can identify this `installation_id` via [Get an organization installation for the authenticated app](#). This requires authenticating as a GitHub App using a JWT, for more information see [Authenticating as a GitHub App](#).

8. Generate an installation access token using the corresponding REST API endpoint, [Create an installation access token for an app](#). This requires authenticating as a GitHub App using a JWT, for more information see [Authenticating as a GitHub App](#), and [Authenticating as an installation](#).

9. Use this installation access token to interact with your repositories, either via the REST or GraphQL APIs, or via a Git client.

For more information, see "[Generating an installation access token for a GitHub App](#)."

Machine users ⊘

If your server needs to access multiple repositories, you can create a new account on GitHub.com and attach an SSH key that will be used exclusively for automation. Since this account on GitHub.com won't be used by a human, it's called a *machine user*. You can add the machine user as a [collaborator](#) on a personal repository (granting read and write access), as an [outside collaborator](#) on an organization repository (granting read,

write, or admin access), or to a [team](#) with access to the repositories it needs to automate (granting the permissions of the team).

> **Tip:** Our [terms of service](#) state:
>
> > *Accounts registered by "bots" or other automated methods are not permitted.*
>
> This means that you cannot automate the creation of accounts. But if you want to create a single machine user for automating tasks such as deploy scripts in your project or organization, that is totally cool.

Pros of machine users 🔗

- Anyone with access to the repository and server has the ability to deploy the project.
- No (human) users need to change their local SSH settings.
- Multiple keys are not needed; one per server is adequate.

Cons of machine users 🔗

- Only organizations can restrict machine users to read-only access. Personal repositories always grant collaborators read/write access.
- Machine user keys, like deploy keys, are usually not protected by a passphrase.

Set up machine users 🔗

1. [Run the `ssh-keygen` procedure](#) on your server and attach the public key to the machine user account.

2. Give the machine user account access to the repositories you want to automate. You can do this by adding the account as a [collaborator](#), as an [outside collaborator](#), or to a [team](#) in an organization.

Further reading 🔗

- [Configuring notifications](#)