



How to Generate SSH Keys for GitHub

Try Application Hosting for free

Run your Node.js, Python, Go, PHP, Ruby, Java, and Scala apps in three easy steps. Connect them to your favorite databases. Pay only for the resources you use.

Free trial



[Git and GitHub](#) are essential tools for every developer. They're widely used in almost every kind of software development project.

There are other Git hosting services like [Gitlab](#) and [Bitbucket](#), but GitHub is the most popular choice for developers. You can even edit your profile to seem more attractive to recruiters.

You can use Git and GitHub to organize your projects, collaborate with other developers, and — of course — at [Kinsta](#).

But because [Git and GitHub are related yet different tools](#), you need to update your workflow with each of them constantly.



We recommend using [SSH](#) keys for each one of your machines. So, in this tutorial, you'll learn what they are, some of their advantages, and how to generate and configure GitHub SSH keys.



Let's get started!

What Are SSH Keys?

Simply put, SSH keys are credentials used for the [SSH \(Secure Shell\) protocol](#) to enable secure access to remote computers over the internet. Usually, that authentication occurs in a command-line environment.

This protocol is based on client-server architecture, which means you as the user (or “client”) need to use special software, called an SSH client, to log into a remote server and execute commands. This is basically what you’re doing when authenticating via a terminal to GitHub.

```
Object-oriented-programming-in-python/structured_programming on  main [!] via  v3.9.7
> git commit -am "Added comment in the structured-programming example file"
[main f0f20ab] Added comment in the structured-programming example file
1 file changed, 3 insertions(+), 1 deletion(-)

Object-oriented-programming-in-python/structured_programming on  main [↑] via  v3.9.7
> git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 408 bytes | 408.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote: This repository moved. Please use the new location:
remote:  git@github.com:DaniDiazTech/Object-Oriented-Programming-in-Python.git
To github.com:DaniDiazTech/Object-oriented-programming-in-python.git
6c70e90..f0f20ab  main -> main
```

— Git push.

But SSH is not only used for GitHub. It’s broadly used by other platforms like Kinsta, Google Cloud, and Amazon Web services to create a secure channel to access their services.

Now, heading into how SSH keys really work, you need to understand the differences between public and private keys.

Public vs Private Keys

Let’s start with the basics.

The SSH protocol uses a technique of cryptography called **asymmetric encryption**. This term may sound complicated and weird, but nothing could be further from the truth.

Basically, asymmetric encryption is a system that uses a pair of keys, namely **public** and **private** keys.

As you may guess, the public key can be shared with anyone. Its main purpose is to encrypt data, converting the message into secret code or ciphertext. This key is usually sent to other systems — for example, servers — to encrypt the data before sending it over the internet.

On the other hand, the private key is the one that you must keep to yourself. It's used to decrypt the encrypted data with your public key. Without it, it's impossible to decode your encrypted information.

This method allows you and the server to keep a safe communication channel for transmitting the information.

Here's what happens in the background when you connect to a server via SSH:

1. The client sends the public key to the server.
2. The server asks the client to sign a random message encrypted with the public key using the private key.
3. The client signs the message and forwards the result to the server.
4. A secure connection is established between the client and the server.

It's important to keep your private keys safe and share them with anyone under no circumstances. They're literally the key to all the information sent to you.

Using SSH Keys With GitHub

Since August 13, 2021, Github no longer accepts password authentication for command-line access. This means now you need to authenticate via a personal access token or use an SSH key (a little bit more convenient).

Here's what happens when you try to authenticate with your GitHub password over HTTP in a terminal:

```
Username for 'https://github.com': yourusername

Password for 'https://yourusername@github.com':

remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://github.blog/2020-12-15-token-authentication-requirements-for-git-1.8.3/
fatal: Authentication failed for 'https://github.com/yourusername/repository'
```

GitHub needs your public key to authorize you to edit any of your repos via SSH.

Let's see how you can generate SSH keys locally.

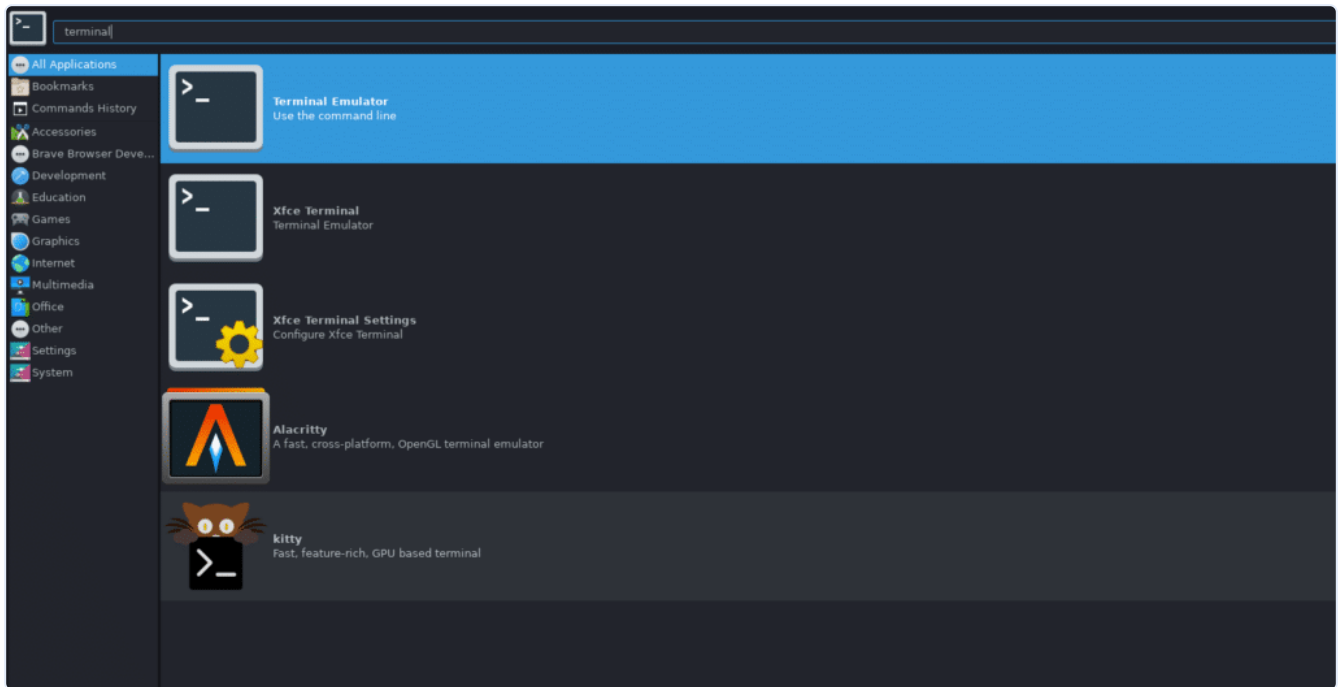
How to Generate SSH Keys Locally

Now that you understand a little bit about the SSH protocol and the differences between public and private keys, it's time to set up the secure SSH channel between your machine and your GitHub repos.

Before we move forward, you should already have a [GitHub account](#), and a terminal/command prompt with [Git](#) installed in your system. If you're running Windows, make sure you installed [Git bash](#), which has all the tools you'll need to follow along with this tutorial built-in.

The OpenSSH client is the most popular open-source software used to connect via SSH. You won't need to worry about your operating system because it's installed by default on Linux, [macOS](#), and [Windows 10](#).

You need to fire up a command prompt on Windows or a terminal on Unix-based systems to generate local SSH keys. Usually, you can do this by searching for "terminal", "cmd", or "powershell" in your application panel, then clicking the icon that shows up.



— Terminal application search.

Info

If you use Linux, most distros have the shortcut Ctrl + Alt + T to open up a terminal.

After doing this, you should have a window similar to the following image.



— Terminal application.

Run the following command to generate a local SSH pair of keys:

```
ssh-keygen -t ed25519 -C "kinstauser@kinsta.com"
```

It's time to tell you a secret: No one can really remember this command! Most developers have to Google it every time because:

1. It's a really long command, with forgettable, random-seeming numbers.
2. We use it rarely, so it's not worth it to commit it to memory most of the time.

However, it's important to understand each command that we introduce into our terminals, so let's see what each part of this one means.

- [ssh-keygen](#): The command-line tool used for creating a new pair of SSH keys. You can see its flags with `ssh-keygen help`
- **-t ed25519**: The `-t` flag is used to indicate the algorithm used to create the digital signature of the key pair. If your system supports it, `ed25519` is the best algorithm you can use to create SSH key pairs.
- **-C “email”**: The `-C` flag is used to provide a custom comment at the end of the public key, which usually is the email or identification of the creator of the key pair.

After you’ve typed the command into your terminal, you’ll have to enter the file to which you would like to save the keys. By default, it’s located in your home directory, in a hidden folder named `“.ssh”`, but you can change it to whatever you like.

Then you’ll be asked for a passphrase to add to your key pair. This adds an extra layer of security if, at any time, your device is compromised. It is not obligatory to add a passphrase, but it’s always recommended.

This is what the whole process looks like:


```

~ took 4m54s
> ssh-keygen -t ed25519 -C "kinstauser@kinsta.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/daniel/.ssh/id_ed25519): /home/daniel/.ssh/kinsta_keys
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/daniel/.ssh/kinsta_keys
Your public key has been saved in /home/daniel/.ssh/kinsta_keys.pub
The key fingerprint is:
SHA256:7I8drb3QzWh/xbRpZ3gnrFSPxPd5aMEVorYat/n+vWo kinstauser@kinsta.com
The key's randomart image is:
+---[ED25519 256]---+
|          .  .  |
|          .  .  |
|         o o .  |
|        . . . * o|
|       S. o + X=|
|      . +o++B=@|
|     ..o+=o++=|
|      + *oE  o|
|     . + ==++o|
+-----[SHA256]-----+

```

— ssh-keygen command.

As you can see, this command generates two files in the directory you selected (commonly `~/.ssh`): the public key with the `.pub` extension, and the private one without an extension.

We'll show you how to add the public key to your GitHub account later.

Add SSH Key to ssh-agent

The **ssh-agent** program runs in the background, holds your private keys and passphrases safely, and keeps them ready to use by ssh. It's a great utility that saves you from typing your passphrase every time you want to connect to a server.

Because of this, you're going to add your new private key to this agent. Here's how:

1. Make sure ssh-agent is running in the background.

```
eval `ssh-agent`  
# Agent pid 334065
```

If you get a message similar to this if everything is fine. It means the ssh-agent is running under a particular process id (PID).

2. Add your SSH private key (the one without extension) to the ssh-agent.

```
ssh-add ~/.ssh/kinsta_keys
```

Replace **kinsta_keys** with the name of your SSH key. If this is the first key you've created, it should be named "id_algorithm_used," for instance, **id_ed25519**.

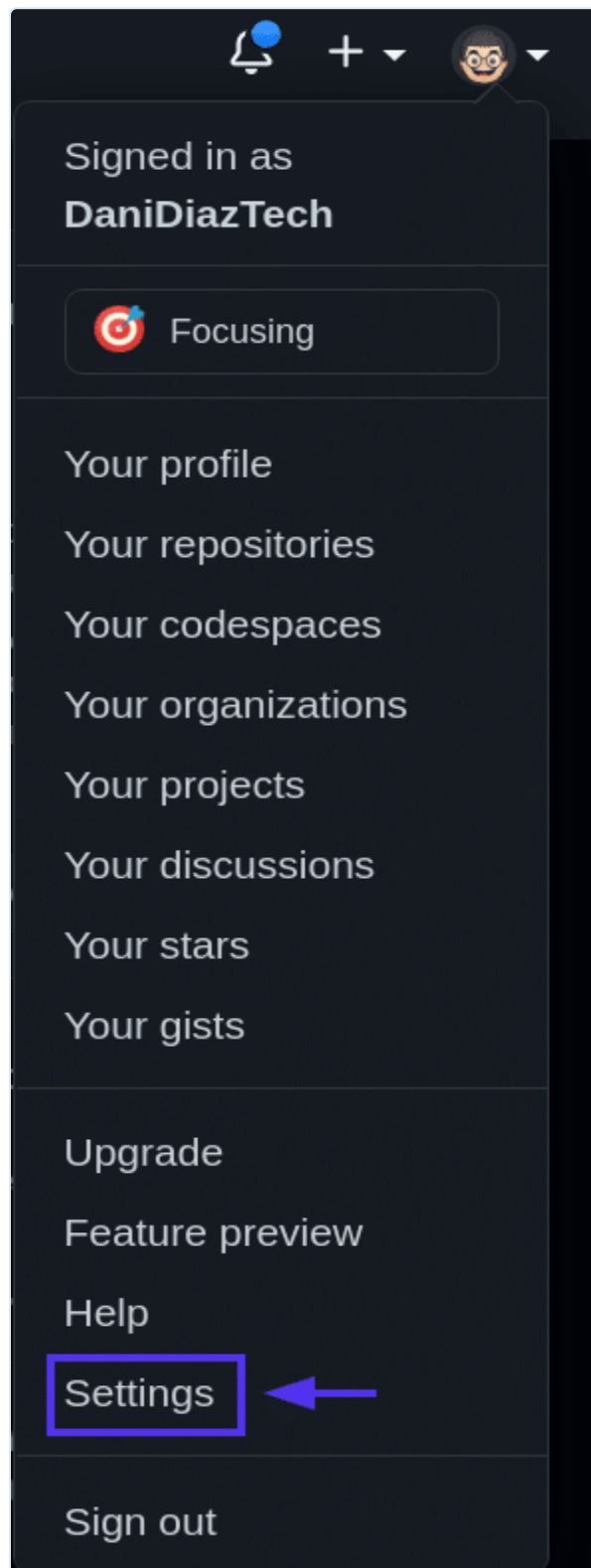
Add SSH Key to GitHub Account

The final step is to add your public key to your GitHub account. Just follow these instructions:

1. Copy your SSH public key to your clipboard. You can open the file where it is located with a text editor and copy it, or use the terminal to show its contents.

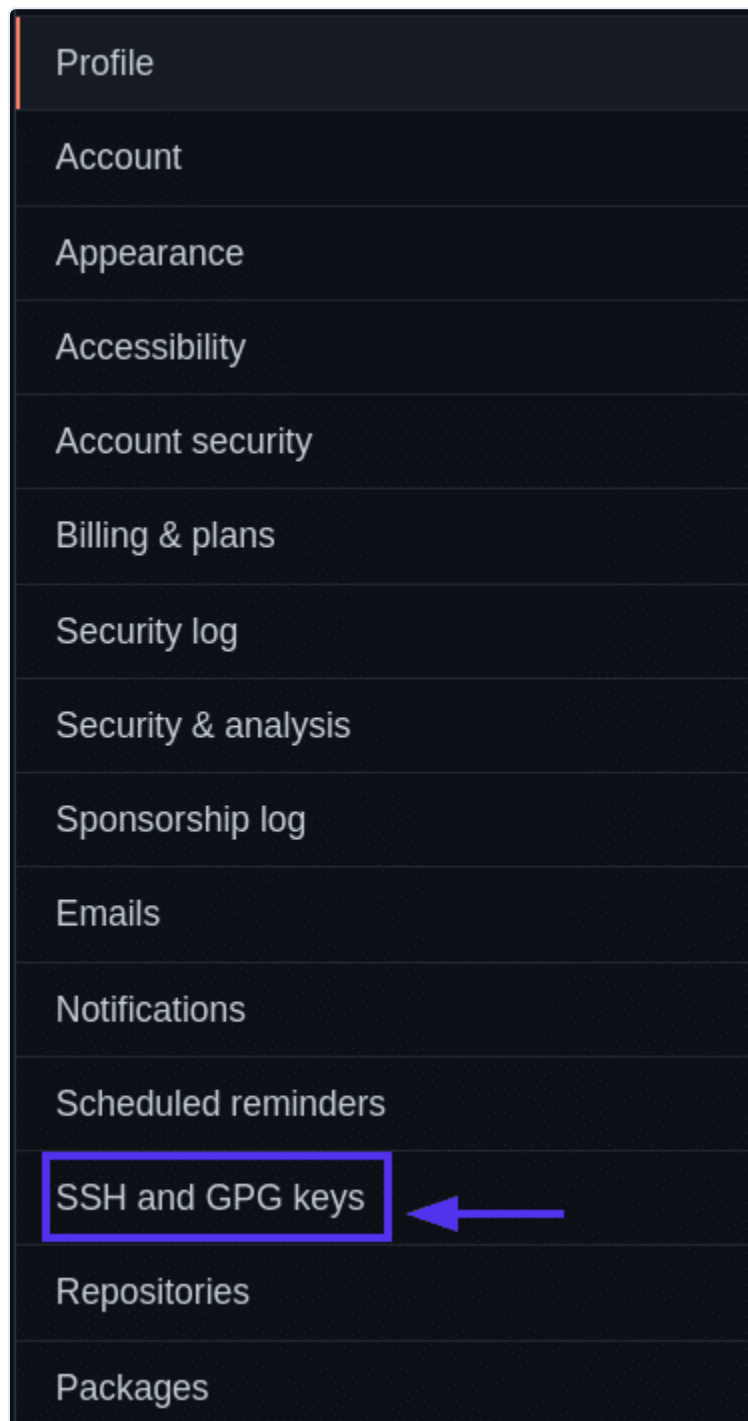
```
cat ~/.ssh/kinsta_keys.pub  
# ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIJl3dIeudNqd0DPMRD6OIh65tjkxFNO
```

2. [Log into GitHub](#) and go to the upper-right section of the page, click in your profile photo, and select **Settings**.



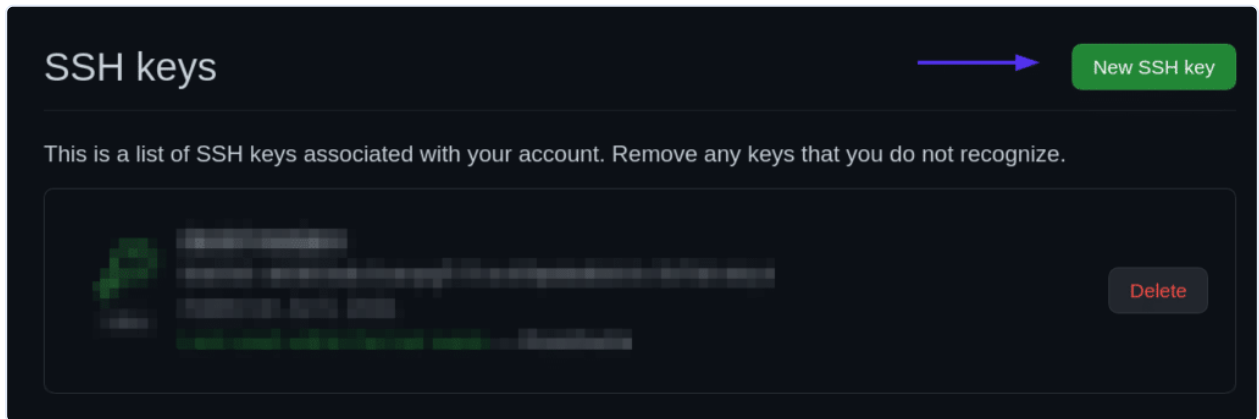
GitHub settings.

3. Then, in profile your settings, click **SSH and GPG keys**.



— SSH and GPG keys.

4. Click the **New SSH key** button.



— New SSH key button.

5. Give your new SSH key on GitHub a **Title** — usually, the device you'll use that key from. And then paste the key into the **Key** area.

SSH keys / Add new

Title

Main computer

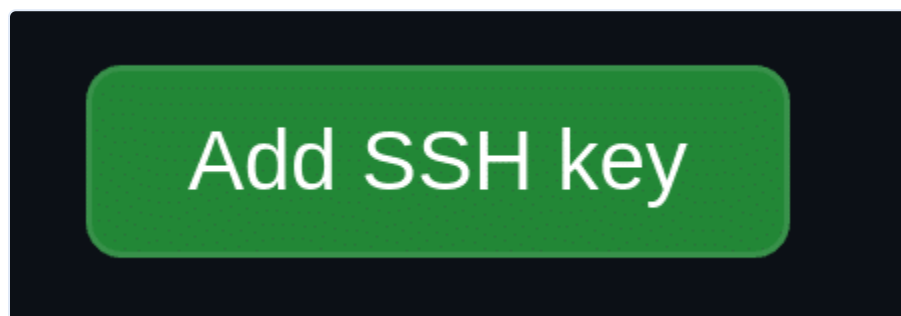
Key

```
ssh-ed25519 AAAAC3NzaC1IZDI1NTE5AAAAIJl3dleudNqd0DPMRD6OIh65tjxkFNOtwGcWB2gCgPhk  
kinstauser@kinsta.com
```

Add SSH key

— Add a new SSH key form.

6. Add your SSH key.



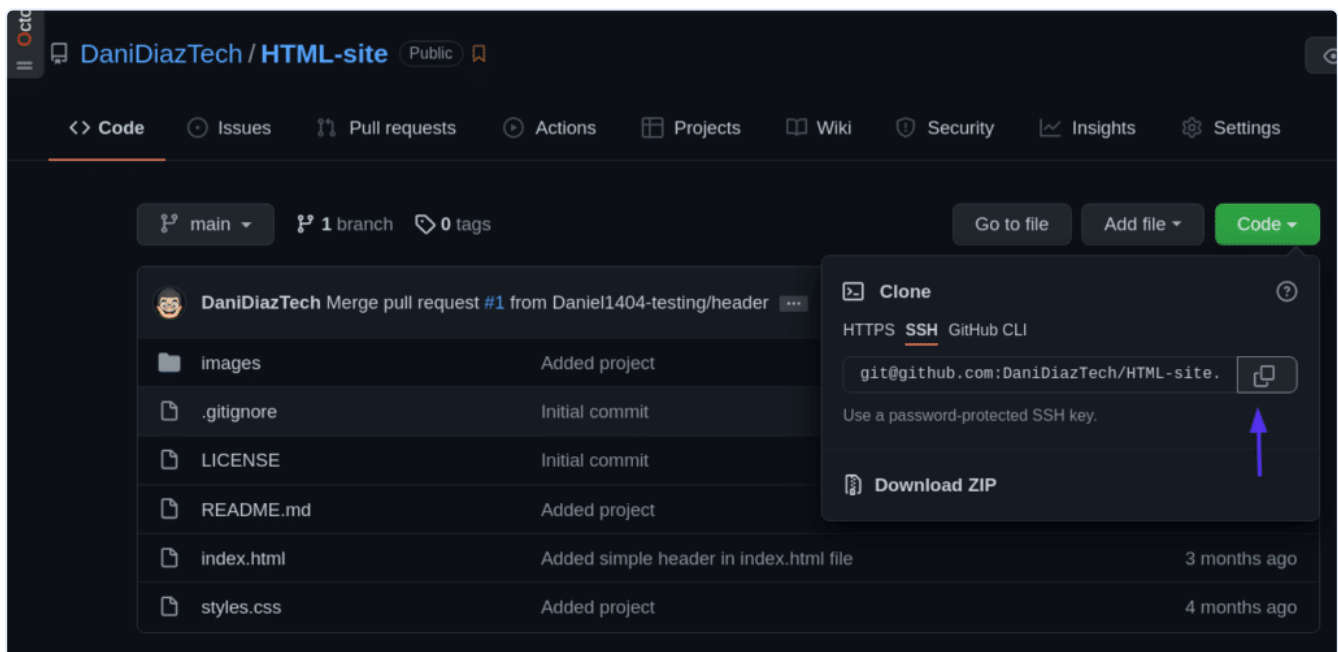
— Add SSH key button.

Test SSH Connection With a Repo Push

It's time to test everything you've done so far. You're going to change, commit, and push to one of your existing repos using SSH to ensure your connection is set up correctly.

For our example, we will modify the simple HTML site we created in our [Git for Web Development tutorial](#).

First, we'll need to clone the repository into our local machine. We can go to the repo page on GitHub and copy the SSH address it provides.



— SSH clone command.

Then, clone the repo using a terminal:

```
git clone git@github.com:DaniDiazTech/HTML-site.git
```

Now, let's add a simple `<h1>` tag in the **index.html** file:

```
...  
<div class="container my-2">  
  <h1 class="text-center">A new title!</h1>  
</div>  
  
<div class="container my-3">  
...  

```

A new title!



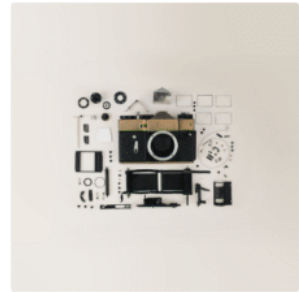
Super computer



Cool computer



Normal laptop



Old camera

— The simple HTML site.

We're not touching any JavaScript or CSS to keep this edit simple. But if you're skilled with JavaScript, you might find a place at Kinsta. Check the [coding skills you need to be part of the Kinsta team](#).

After doing this, commit the changes:

```
git commit -am "Added a simple title"
```

And push them into GitHub just as you'd normally do.

```
git push
```

If everything went fine, congratulations! You just set up an SSH connection between your machine and GitHub.

Manage Multiple SSH Keys for Different GitHub Accounts

If you have multiple GitHub accounts — let's say one for your personal projects and one for your work — it's difficult to use SSH for both of them. You would normally need separate machines to authenticate to different GitHub accounts.

But this can be solved easily by configuring the SSH config file.

Let's get into it.

1. Create another SSH key pair, and add it to your other GitHub account. Keep in mind the name of the file you're assigning the new key to.

```
ssh-keygen -t ed25519 -C "work@email.com"
```

2. Create the SSH config file. The config file tells the ssh program how it should behave. By default, the config file may not exist, so create it inside the `.ssh/` folder:

```
touch ~/.ssh/config
```

3. Modify the SSH config file. Open the config file and paste the code below:

```
#Your day-to-day GitHub account

Host github.com
  HostName github.com
  IdentityFile ~/.ssh/id_ed25519
  IdentitiesOnly yes

# Work account
Host github-work
  HostName github.com
  IdentityFile ~/.ssh/work_key_file
  IdentitiesOnly yes
```

Now, every time you need to authenticate via SSH using your work or secondary account, you tweak a bit the repo SSH address, from:

```
git@github.com:workaccount/project.git
```

...to:

```
git@github-work:workaccount/project.git
```

Summary

Congratulations — you've learned most of the practical knowledge you need to connect to GitHub via SSH!

This tutorial discussed the need for the SSH protocol, the differences between public and private keys, how to generate keys, add them to GitHub, and even manage multiple SSH keys for different GitHub accounts. Keep in mind that unless you want to lose access to everything, your private key needs to stay that way: private.

With this knowledge, now you're ready to develop a [flawless workflow with Git and GitHub](#). Keep coding!