

BY BOBBY ILIEV

Introduction to Bash Scripting

FOR DEVELOPERS



Table of Contents

About the book	5
About the author	6
Sponsors	7
Ebook PDF Generation Tool	9
Book Cover	10
License	11
 Introduction to Git	 12
 Version Control	 14
 Installing Git	 15
 Basic Shell Commands	 18
 Git Configuration	 21
 Introduction to GitHub	 25
 Initializing a Git project	 30
 Git Status	 32
 Git Add	 34
 Git Commit	 36

Git Diff	39
Git Log	41
Gitignore	44
SSH Keys	53
Git Push	58
Git Pull	61
Git Workflow	65
Git Branches	66
Reverting changes	67
Forking in Git	68
Git Clone	69
Git And VS Code	70
Installing VS Code	71
Cloning a repository in VS Code	73
Create a branch	74
Setup a commit message template	75
Conclusion	76
Additional sources:	77

Conclusion 78

About the book

- **This version was published on March 12 2021**

This is an open-source introduction to Git and GitHub guide that will help you learn the basics of version control and start using Git for your SysOps, DevOps, and Dev projects. No matter if you are a DevOps/SysOps engineer, developer, or just a Linux enthusiast, you can use Git to track your code changes and collaborate with other members of your team or open source maintainers.

The guide is suitable for anyone working as a developer, system administrator, or a DevOps engineer and wants to learn the basics of Bash scripting.

The first 13 chapters would be purely focused on getting some solid Bash scripting foundations, then the rest of the chapters would give you some real-life examples and scripts.

About the author

My name is Bobby Iliev, and I have been working as a Linux DevOps Engineer since 2014. I am an avid Linux lover and supporter of the open-source movement philosophy. I am always doing that which I cannot do in order that I may learn how to do it, and I believe in sharing knowledge.

I think it's essential always to keep professional and surround yourself with good people, work hard, and be nice to everyone. You have to perform at a consistently higher level than others. That's the mark of a true professional.

For more information, please visit my blog at <https://bobbyiliev.com>, follow me on Twitter [@bobbyiliev_](#) and [YouTube](#).

Sponsors

This book is made possible thanks to these fantastic companies!

DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale.

It provides highly available, secure, and scalable compute, storage, and networking solutions that help developers build great software faster.

Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available.

For more information, please visit <https://www.digitalocean.com> or follow [@digitalocean](https://twitter.com/digitalocean) on Twitter.

If you are new to DigitalOcean, you can get a free \$100 credit and spin up your own servers via this referral link here:

[Free \\$100 Credit For DigitalOcean](#)

DevDojo

The DevDojo is a resource to learn all things web development and web design. Learn on your lunch break or wake up and enjoy a cup of coffee with us to learn something new.

Join this developer community, and we can all learn together, build together, and grow together.

[Join DevDojo](#)

For more information, please visit <https://www.devdojo.com> or follow [@thedeveloper](#) on Twitter.

Ebook PDF Generation Tool

This ebook was generated by [Ibis](#) developed by [Mohamed Said](#).

Ibis is a PHP tool that helps you write eBooks in markdown.

Book Cover

The cover for this ebook was created with [Canva.com](https://www.canva.com).

If you ever need to create a graphic, poster, invitation, logo, presentation – or anything that looks good — give Canva a go.

License

MIT License

Copyright (c) 2020 Bobby Iliev

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Introduction to Git

Welcome to this Git and GitHub basics training guide! In this **Git crash course**, you will learn the **basics of Git** so you can use Git to track your code changes and collaborate with other members of your team or open source maintainers.

Whether you are a newcomer to programming, or an experienced one, you have to know how to use Git. Most of the projects that a small or big group of developers work on are done through GitHub or GitLab.

It makes working with other developers so much more exciting and enjoyable. Just by creating a new branch, adding all your brilliant ideas to the code that can help the project, committing it, and then pushing it to GitHub or GitLab. Then after the PR(pull request) has been opened, reviewed, and then merged, you can get back to your code and continue adding more awesome stuff. After pulling the changes from the main/master branch, of course.

If what you just read doesn't make any sense to you, don't worry. Everything will be explained in this eBook!

This eBook will show you the basics on how to start using Git and try to help you get more comfortable with it.

It does look a bit scary in the beginning, but don't worry. It's not as frightening as it seems, and hopefully, after reading this eBook, you can get a bit more comfortable with Git.

Learning Git is essential for every programmer. Even some of the biggest companies use GitHub for their projects. Remember that the more you use it, the more you're going to get used to it.

Git is without a doubt the most popular open source version control system for tracking changes in source code out there.

The original author of git is Linus Torvalds who is also the creator of **Linux**.

Git is designed to help programmers coordinating work among each other. Its goals include speed, data integrity, and support for distributed workflows.

Version Control

Version control, also called *Source control*, allows you to track and manage all of the changes to your code.

The main benefit of version control is that multiple people could work on the same project at the same time. With version control tools like Git, you can track all of the changes to your code and in case of any problems you could easily revert back to a working state of your source code.



With distributed version control systems like Git, you would have your source code stored on a remote repository like GitHub and also a local repository stored on your computer.

You will learn more about remote and local repositories in the next few chapters, but one of the main points for the moment is that your source code would be stored on a remote repository, so in case that something goes wrong with your laptop you would not lose all of your changes but they will be safely stored on GitHub.

Installing Git

In order for you to be able to use Git on your local machine, you would need to install it.

Depending on the operating system that you are using, you can follow the steps here.

Install Git on Linux

With most Linux distributions the Git command line tool comes installed out of the box.

If this is not the case for you, you can install Git with the following command:

- On RHEL Linux:

```
sudo dnf install git-all
```

- On Debian based distributions including Ubuntu:

```
sudo apt install git-all
```

Install Git on Mac

If you are using Mac, Git should be available out of the box as well. However if this is not the case there are 2 main ways of installing Git on your Mac:

- Using Homebrew: in case that you are using Homebrew you can open your terminal and run the following:

```
brew install git
```

- Git installer: Alternatively, you could use the following installer:

[git-osx-installer](#)

I would personally stick to Homebrew.

Install Git on Windows

If you have a Windows PC, you can follow the steps on how to install Git on Windows here:

[Install Git on Windows](#)


During the installation, make sure to choose the Git Bash option as this would provide you with a Git Bash terminal which you will use while following along.

Check Git version

Once you have installed Git, in order to check the version of Git that you have installed on your machine, you could use the following command:

```
git --version
```

Example output:



```
git version 2.25.1
```

In my case I have Git 2.25.1 installed on my laptop.

Basic Shell Commands

As throughout this eBook we will be using mainly Git via the command line, it is important to know basic shell commands so that you could find your way around the terminal.

So before we get started, let's go over a few basic shell commands!

The `ls` command

The `ls` command allows you to list the contents of a folder/directory. All that you need to do in order to run the command is to open a terminal and run the following:

```
ls
```

The output will show you all of the files and folders that are located in your current directory. In my case the output is the following:

```
CONTRIBUTING.md ebook README.md
```

For more information about the `ls` command make sure to check out this page [here](#).

Note: This will work on a Linux/UNIX based systems. If you are on Windows and if you are using the built-in CMD, you would have to use the `dir` command

The `cd` command

The `cd` command stands for **Change Directory** and allows you to navigate through the filesystem of your computer or server. Let's say that I wanted to go inside the `ebook` directory from the output above, what I would need to do is to run the `cd` command followed by the directory that I want to access:

```
cd ebook
```

If I wanted to go back one level up, I would use the `cd ..` command.

The `pwd` command

The `pwd` command stands for **Print Working Directory** which essentially means that when you run the command, it will show you the current directory that you are in.

Let's take the example from above, if I run the `pwd` command I would get the full path to the folder that I'm currently in:

```
pwd
```

Output:

```
/home/bobby/introduction-to-git
```

Then I could use the `cd` command and access the `ebook` directory:

```
cd ebook
```

And finally if I was to run the `pwd` command again, I would see the

following output:

```
/home/bobby/introduction-to-git/ebook
```

Essentially what happened was that thanks to the `pwd` command, I was able to see that I'm at the `/home/bobby/introduction-to-git` directory and then after accessing the `ebook` directory, again by using `pwd` I was able to see that my new current directory is `/home/bobby/introduction-to-git/ebook`.

The `rm` command

The `rm` command stands for `remove` and allows you to delete files and folders. Let's say that I wanted to delete the `README.md` file, what I would have to do is run the following command:

```
rm README.md
```

In case that I had to delete a folder/directory, I would need to specify the `-r` flag:

```
rm -r ebook
```

Note: keep in mind that the `rm` command would completely delete the files and folders and the action is irreversible, meaning that you can't get them back.

One thing that you need to keep in mind is that all shell commands are case sensitive, so if you type `LS` it would not work.

With that, now you know some basic shell commands which will be beneficial for your day-to-day activities.

Git Configuration

The first time you setup Git on your machine, you would need to do some initial configuration.

There are a few main things that you would need to configure:

- Your details: like your name and email address
- Your Git Editor
- The default branch name: we will learn more about branches later on

We can change all of those things by using the `git config` command.

Let's get started with the initial configuration!

The `git config` command

In order to configure your Git details like your user name and your email address you need to use the following command:

- Configuring your Git user name:

```
git config --global user.name "Your Name"
```

- Configuring your Git email address:

```
git config --global user.email johndoe@example.com
```

Usually, it is good to have a matching user name and email for your local Git configuration and your GitHub profile details

- Configuring your Git default editor

In some cases when running Git commands via your terminal, an editor will open where you could type a commit message for example. To specify your default editor, you need to run the following command:

```
git config --global core.editor nano
```

You can change the `nano` editor with another editor like `vim` or `emacs` based on your personal preferences.

- Configuring the default branch name

Whenever creating a new repository on your local machine, it gets initialized with a specific branch name which might be different from the default branch on GitHub. To make sure that the branch name on your local machine matches the default branch name on GitHub, you can use the following command:

```
git config --global init.defaultBranch main
```

Finally once you are done with all changes, you can check your current Git configuration with the following command:

```
git config --list
```

Example output:

```
user.name=Bobby Iliev
user.email=bobby@bobbyiliev.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

The ~/.gitconfig file

As we used the `--global` command, all of those Global Git settings, would be stored in a `.gitconfig` file inside your home directory.

We can use the `cat` command to check the content of the file:

```
cat ~/.gitconfig
```

Example output:

```
[user]
  name = Bobby Iliev
  email = bobby@bobbyiliev.com
```

You can even change the file manually with your favourite text editor, but I personally prefer to use the `git config` command to prevent any syntax problems.

The .git directory

Whenever you initialize a new project or clone one from GitHub, it would have a `.git` directory where all of the Git commits would be recorded at and also a `config` file where the configuration settings for the particular project would be stored at.

You could use the `ls` command to check the contents of the `.git` folder:

```
ls .git
```

Output:

```
COMMIT_EDITMSG HEAD branches config description hooks  
index info logs objects refs
```

Note: Before running the command you would need to be inside your project's directory. We will learn about this in the next chapters when we learn more about the `git init` command and cloning an existing repository from GitHub with the `git clone` command

Introduction to GitHub

Before we deep dive into all of the various Git commands, let's quickly get familiar with GitHub.

Git is essentially the tool that you use to track your code changes, and GitHub on the otherside is a website where you can push your local projects to.

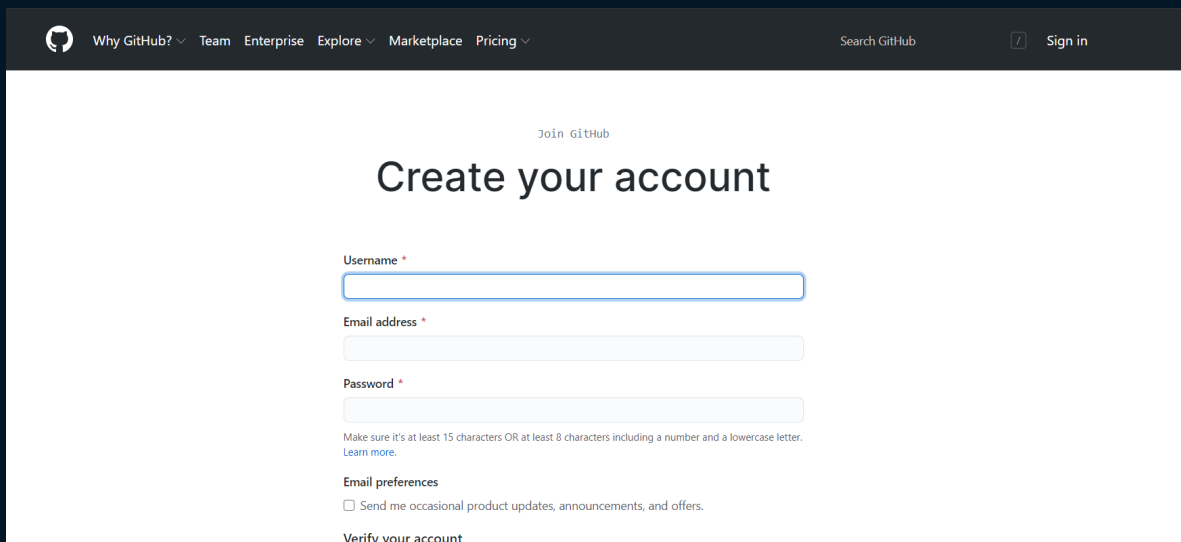
This is essentially needed as it would act as a central hub where you would store your project at and all of your team mates or other people working on the same project as you, would push their changes to.

GitHub Registration

Before you get started you would need to create an account with GitHub, you can do so via this link here:

- [Join GitHub](#)

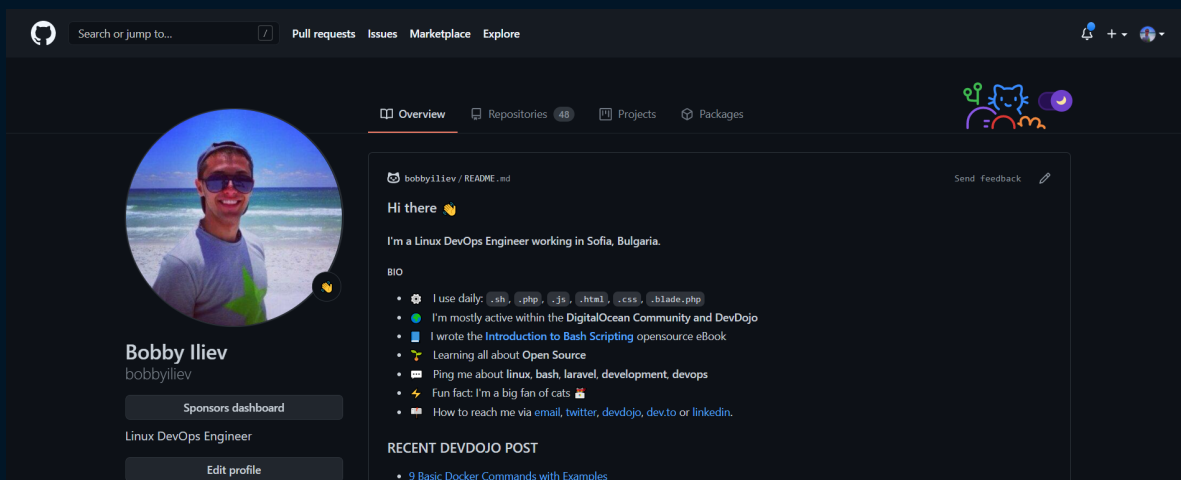
You would get to the following page where you would need to add your new account details:



The screenshot shows the GitHub registration page. At the top, there's a navigation bar with links like 'Why GitHub?', 'Team', 'Enterprise', 'Explore', 'Marketplace', and 'Pricing'. The main heading is 'Create your account'. Below it, there are three input fields: 'Username', 'Email address', and 'Password'. A note specifies password requirements: 'Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)'. There's an 'Email preferences' section with a checkbox for 'Send me occasional product updates, announcements, and offers.' and a 'Verify your account' link at the bottom.

GitHub Profile

Once you've registered, you can go to https://github.com/YOUR_USER_NAME, and you would be able to see your public profile where you could add some information about yourself. Here is an example profile which you could check: [GitHub Profile](#)



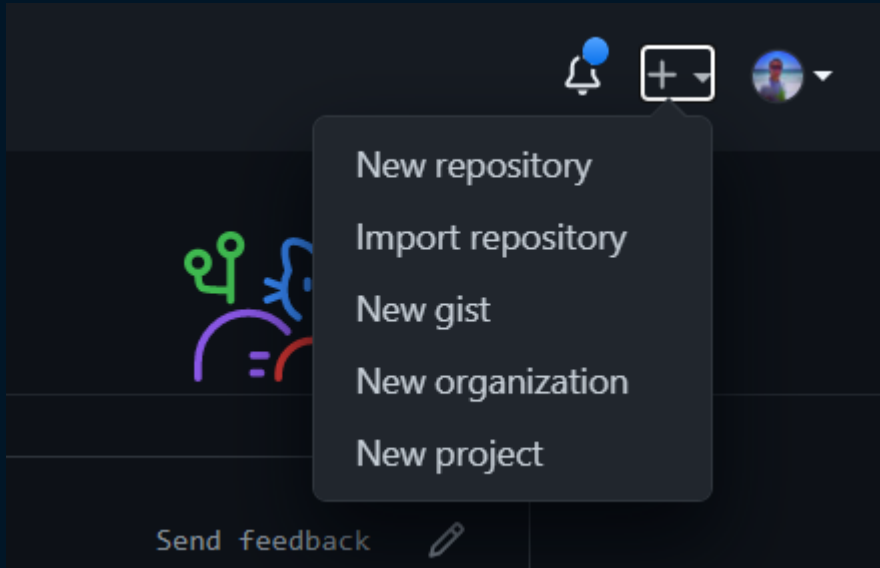
The screenshot shows a GitHub profile for 'Bobby Iliev' (username: bobbyiliev). The profile includes a circular profile picture of a man with sunglasses. To the right of the picture, there's a bio: 'Hi there 🍌 I'm a Linux DevOps Engineer working in Sofia, Bulgaria.' Below the bio, there's a 'BIO' section with a list of interests and skills, including 'I use daily: .sh, .php, .js, .html, .css, .blade.php', 'I'm mostly active within the DigitalOcean Community and DevDojo', 'I wrote the Introduction to Bash Scripting opensource eBook', 'Learning all about Open Source', 'Ping me about linux, bash, laravel, development, devops', 'Fun fact: I'm a big fan of cats 🐱', and 'How to reach me via email, twitter, devdojo, dev.to or linkedin.' At the bottom, there's a 'RECENT DEVDOJO POST' section with a link to '9 Basic Docker Commands with Examples'.

Creating a new repository

If you are not familiar with the word repository, you could think of it as a project, it would hold all of the files of your application or website that you are build. In many cases people would use repo instead of

repository for short.

To create a new repository on GitHub, you have to click on the + sign on the top right and then click on the **New Repository** button:

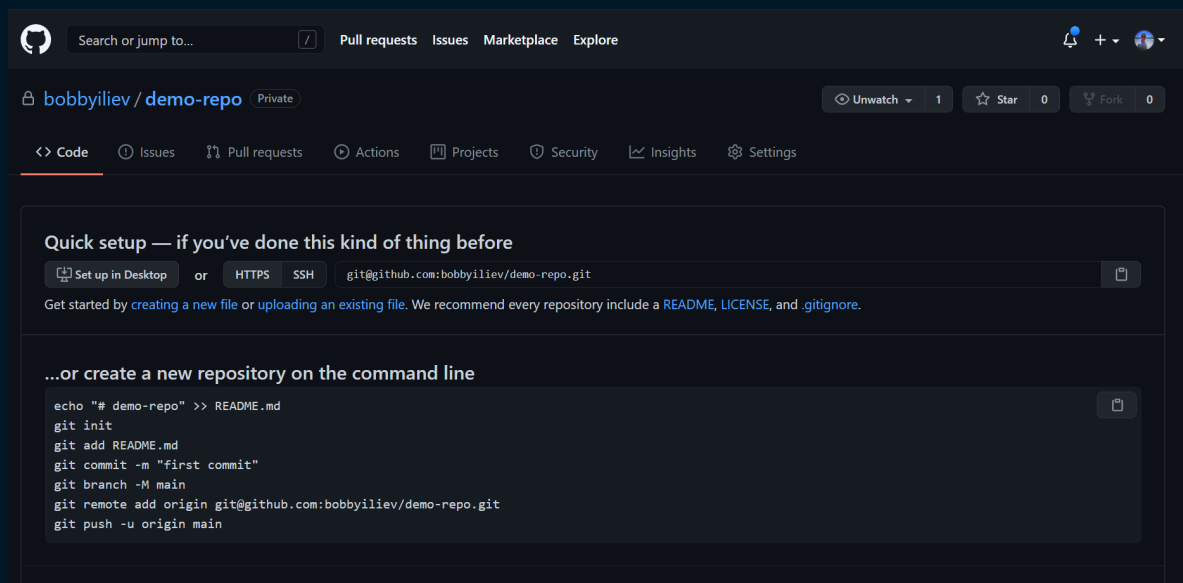


After that you would get to a page where you can specify the information for your new repository like:

- The name of the project: Here make sure to use something descriptive
- Some general description about the project and what it is about
- Choose whether you want the repository to be Public or Private

A screenshot of the 'Create a new repository' page on GitHub. The page has a dark theme. At the top, there is a navigation bar with the GitHub logo, a search bar, and links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. The main heading is 'Create a new repository'. Below it, a subtext says 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)'. There is a section for 'Repository template' with a 'No template' button. Below that, there are fields for 'Owner' (a dropdown menu showing 'bobbyiliev') and 'Repository name' (a text input field). A hint text says 'Great repository names are short and memorable. Need inspiration? How about [miniature-funicular?](#)'. There is an optional 'Description' text area. At the bottom, there are two radio buttons for visibility: 'Public' (selected) and 'Private'. The 'Public' option has a description: 'Anyone on the internet can see this repository. You choose who can commit.' The 'Private' option has a description: 'You choose who can see and commit to this repository.'

Once you've added the necessary information and hit the create button, you will get to a page with some instructions on how to push your local project to GitHub:



We will go over those steps more in depth in the next few chapters.

Public vs Private repositories

Depending on the project and whether or not it is open source, you could set your repository to be **public** or **private**.

The main difference is that with a public repository, anyone on the internet can see this repository. But even though that everyone would be able to see the repository and read the code, you would be the maintainer of the project and you will choose who can commit.

With a private repository, it would only be available for you and the people that you've invited.

Public repositories are used for open source projects.

The README.md file

The `README.md` file is an essential part of each project. The `.md` extension stands for Markdown.

You can think of the `README.md` file as the introduction to your repository. It's very helpful that while looking at someone's repo you can just scroll down to their README file and have a look at what their project is all about.

And it is very important that your project is properly introduced. Because if the project itself isn't introduced properly, no one is going to spend their time in helping to improve it and try to further develop it.

That's why having a good README file shouldn't be overlooked and you should spend a considerable amount of your time on it.

In this post, I am going to share some tips with you about how you can improve your README file, and hopefully, it can help you with your repos.

For more information, make sure to check out this post on [how to write a good README.md file](#).

Initializing a Git project

If you are starting a new project or if you have an existing project which you would like to add to Git and then push to GitHub, you need to initialize a new Git project with the `git init` command.

To keep things simple, let's say that we want to start building a fresh new project. The first thing that I would usually do is to create a new folder where my project files would be stored at. To do that I can use the `mkdir` command followed by the name of the folder which will create a new empty directory/folder:

```
mkdir new-project
```

The above command will create a folder called `new-project`. Then as we learned in chapter 4, we can use the `cd` command to accesss the directory:

```
cd new-project
```

After that by using the `ls` command we will be able to verify that the directory is completely empty:

```
ls -lah
```

Then with that we are ready to initialize a new Git project:

```
git init
```

You will get the following output:

```
Initialized empty Git repository in /home/devdojo/new-project/.git/
```

As you can see, what the `git init` command does is to create a new `.git` folder which we already discussed in chapter 5.

With that you've successfully create a new empty Git project! Let's move to the next chapter where you will learn how to use the `git status` command to check the current status of your repository.

Git Status

Whenever you make changes to your Git project, you would want to verify what has changed before making a commit or before pushing your changes to GitHub for example.

To check the current status of your project, you can use the `git status` command. If you run the `git status` command in the same directory where you initialized your Git project from the last chapter you will see the following output:

```
On branch main  
  
No commits yet  
  
nothing to commit (create/copy files and use "git add" to  
track)
```

As this is a fresh new repository there are no commits and no changes yet. So let's go ahead and create a `README.md` file with some generic content. We can run the following command to do so:

```
echo "# Demo Project" >> README.md
```

What this would do is to output the `# Demo Project` and store it in the `README.md` file.

If you run `git status` again you will then see the following output:


```
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        README.md
nothing added to commit but untracked files present (use "git
add" to track)
```

As you can see, Git is detecting that there is 1 new file that is not tracked at the moment called `README.md` which we just created. And already Git is prompting us to use the `git add` command to start tracking the file. We will learn more about the `git add` command in the next chapter!

We are going to be using the `git status` command throughout the next few chapters a lot! This is particularly helpful, especially when you've modified a lot of files and you want to check the current status and see all of the modified, updated or deleted files.

Git Add

By default, when you create a new file inside your Git project it is not being tracked by Git. So to tell git that it should start tracking the file, you need to use the `git add` command.

The syntax is the following:

```
git add NAME_OF_FILE
```

In our case, we have only 1 file inside our project called `README.md`, so to add this file to Git, we can use the following command:

```
git add README.md
```

If you then run `git status` again, you will see a different output:

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
   new file:   README.md
```

Here you would see that there are now some changes staged and ready to be committed. Also Git tells us that the `README.md` is a new file which was just staged and has not been tracked before.

In case that you have a couple of files, you could list them all divided by space after the `git add` command to stage them all rather than running `git add` multiple times for each individual file:

```
git add file1.html file2.html file3.html
```

With the above we will add the 3 files by running `git add` just once, however in some cases you might have a lot of new files and adding them one by one could be very time consuming.

So there is a way to stage absolutely all files in your current project and this is by specifying a dot after the `git add` command as follows:

```
git add .
```

Note: You need to be careful with this as in some cases there might be some files that you don't want to add to Git.

With that we are ready to move on and learn about the `git commit` command.

Git Commit

Once you have added/staged your files, the next step is to actually commit the changes. So if you run `git status` again you will be able to see that Git tells use that there are changes to be committed:

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

In this case it is only the `README.md` file that will be committed. So in order to do so, we can run the following command:

```
git commit -m "Your Commit Message Here"
```

Rundown of the command:

- `git commit`: here we are telling git that we want to commit the changes that we've staged with the `git add` command
- `-m`: this flag indicates that we will specify our commit message directly after that
- Finally in the quotes we've got our commit message, it is important to write short and descriptive commit messages

In our case we could set our commit message to something like `"Initial commit"` or `"Add README.md"` file for example.

If you don't specify the `-m` flag, Git will open the default text editor that we've configured in chapter 5 where you will be able to type the commit message directly.

After running the `git commit` command, we can use the `git status` command again to check the current status:

```
git status
```

Output:

```
On branch main
nothing to commit, working tree clean
```

As you can see, Git is telling us that there are no changes to be committed as we've already committed them.

Let's go ahead and make another change to the `README.md` file. You can open the file with your favourite text editor and make the change directly, or you can run the following command:

```
echo "Git is awesome!" >> README.md
```

The above would add a new line at the bottom of the `README.md` file. So if we were to run `git status` again we will see the following output:

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
  directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -
a")
```

As you can see Git has detected that the `README.md` file has been modified and is also prompting us to use the command that we've

learned to first stage/add the file!

The `git status` command gives us a great overview of the files that have changed, but it does not show us what actually the changes are. In the next chapter we are going to learn how to check the differences from the last commit and the current changes.

Git Diff

As mentioned in the last chapter the `git status` command gives us a great overview of the files that have changed, but it does not show us what actually the changes are.

You can check the actual changes that were made with the `git diff` command. If we were to run the command in our repository, we would see the following output:

```
diff --git a/README.md b/README.md
index 9366068..2b14655 100644
--- a/README.md
+++ b/README.md
@@ -1,2 @@
  # Demo Project
+Git is awesome
```

As we only changed the `README.md` file, Git is showing us the following:

- `diff --git a/README.md b/README.md`: here git indicates that it is showing the changes made to the `README.md` file since the last commit compared to the current version of the file.
- `@@ -1,2 @@`: here git indicates that 1 new line was added
- `+Git is awesome`: here the important part is the `+` which indicates that this is a new line that was added, in case that we remove a line you would see a `-` sign instead.

In our case, as we only added 1 new line to the file, Git indicates that only 1 file was changed and that only 1 new line was added.

Next let's go ahead and stage that change and commit it with the

commants that we've lerned from the previous chapters!

- Stage the changed file:

```
git add README.md
```

- Then again run `git status` to check the current status:

```
git status
```

The output would look like this indicating that there is 1 modified file:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
       modified:   README.md
```

- Commit the changes:

```
git commit -m "Update README.md"
```

Finally if you run `git status` again you will see that there are no changes ot be committed.

I always run `git status` and `git diff` before making any commits, just so that I'm sure what has changed.

In some cases you would like to see a list of the previous commits, we will learn how to do that in the next chapter.

Git Log

In order to list all of the previous commits, you can use the following command:

```
git log
```

This will provide you with your commit history, the output would look like this:

```
commit da46ce39a3fd663ff802d013f834431d4b4159a5 (HEAD -> main)
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Fri Mar 12 17:14:02 2021 +0000

    Update README.md

commit fa583473b4be2807b45f35b755aa84ac78922259
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Fri Mar 12 17:01:17 2021 +0000

    Initial commit
```

Rundown of the output:

- **commit da46ce39a3fd663ff802d013f834431d4b4159a5**: Here you can see the specific commit ID
- **Author: Bobby Iliev...** : Then you can see who created the changes
- **Date: Fri Mar 12...**: After that you've got the exact time and date when the commit was created
- Finally you have the commit message, this is one of the reasons why it is important to write short and descriptive commit

messages, so that later on you could tell what changes were introduced by the particular commit.

If you want to check the differences between the current state of your repository and a particular commit, what you could do is use the `git diff` command followed by the commit ID:

```
git diff fa583473b4be2807b45f35b755aa84ac78922259
```

In my case the output will be the following:

```
diff --git a/README.md b/README.md
index 9366068..2b14655 100644
--- a/README.md
+++ b/README.md
@@ -1,2 @@
-# Demo Project
+Git is awesome
```

So the difference between that specific commit and the current state of the repository is the change in the `README.md` file.

In case that you wanted to see only the commit IDs and commit messages on one line, you could add the `--oneline` argument:

```
git log --oneline
```

Output:

```
* da46ce3 (HEAD -> main) Update README.md
* fa58347 Initial commit
```

With that you now know how to check your commit history! Next let's

go ahead and learn how to exclude specific files from Git!

Gitignore

In some cases you might not want to commit some of your files to Git due to security reasons.

For example if you have a config file where you have all of your database credentials and other sensitive secrets, you should never add it to Git and push it to GitHub as other people will be able to get hold of that sensitive information.

To do so, you need to have a `gitignore` file which includes a list of all of the files and directories that should be excluded from your Git repository. In this chapter you will learn how to do that!

Ignoring a specific file

Let's have a look at the following example if you had a `PHP` project and a file called `config.php` which stores your database connection string details like username, password, host, etc.

In order to exclude that file from your git project, you could create a file called `.gitignore` inside your project's directory:

```
touch .gitignore
```

Then inside that file, all that you need to add is the name of the file that you want to ignore, so the content of the `.gitignore` file would look like this:

```
config.php
```

That way the next time you run `git add .` and then run `git commit` and `git push` the `config.php` file will be ignored and will not be added nor pushed to your Github repository.

That way you would keep your database credentials safe!

Ignoring a whole directory

In some cases, you might want to ignore a whole folder, for example, if you have a huge `node_modules` folder, there is no need to add it and commit it to your Git project, as that directory is generated automatically whenever you run `npm install`.

The same would go for the `vendor` folder in Laravel. You should not really add the `vendor` folder to your Git project, as all of the content of that folder, is generated automatically whenever you run `composer install`.

So in order to ignore the `vendors` and `node_modules` folders, you could just add them to your `.gitignore` file:

```
# Ignored folders
/vendor/
node_modules/
```

Getting a gitignore file for Laravel

In order to get a `gitignore` file for Laravel, you could get the file from [the official Laravel Github repository] here(<https://github.com/laravel/laravel/>).

The file would look something like this:

```
/node_modules
/public/hot
/public/storage
/storage/*.key
/vendor
.env
.env.backup
.phpunit.result.cache
Homestead.json
Homestead.yaml
npm-debug.log
yarn-error.log
```

It essentially includes all of the files and folders that are not needed to get the application up and running.

Using gitignore.io

As the number of frameworks and application grows day by day, it might be hard to keep your `.gitignore` files up to date or it could be intimidating if you had to search for the correct `.gitignore` file for every specific framework that you use.

I recently discovered an opensource project called gitignore.io. It is a site and a CLI tool that has a huge list of predefined `gitignore` files for different frameworks.

All that you need to do is visit the site and search for the specific framework that you are using.

For example, let's search for a `.gitignore` file for Node.js:



Then just hit the **Create button** and you would instantly get a well

documented [.gitignore](#) file for your Node.js project, which will look like this:

```
# Created by
https://www.toptal.com/developers/gitignore/api/node
# Edit at
https://www.toptal.com/developers/gitignore?templates=node

### Node ###
# Logs
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
lerna-debug.log*

# Diagnostic reports (https://nodejs.org/api/report.html)
report.[0-9]*.[0-9]*.[0-9]*.[0-9]*.json

# Runtime data
pids
*.pid
*.seed
*.pid.lock

# Directory for instrumented libs generated by
# jscoverage/JSCover
lib-cov

# Coverage directory used by tools like Istanbul
coverage
*.lcov

# nyc test coverage
.nyc_output

# Grunt intermediate storage
# (https://gruntjs.com/creating-plugins#storing-task-files)
.grunt

# Bower dependency directory (https://bower.io/)
bower_components

# node-waf configuration
```

```
.lock-wscript

# Compiled binary addons (https://nodejs.org/api/addons.html)
build/Release

# Dependency directories
node_modules/
jspm_packages/

# TypeScript v1 declaration files
typings/

# TypeScript cache
*.tsbuildinfo

# Optional npm cache directory
.npm

# Optional eslint cache
.eslintcache

# Microbundle cache
.rpt2_cache/
.rts2_cache_cjs/
.rts2_cache_es/
.rts2_cache_umd/

# Optional REPL history
.node_repl_history

# Output of 'npm pack'
*.tgz

# Yarn Integrity file
.yarn-integrity

# dotenv environment variables file
.env
.env.test

# parcel-bundler cache (https://parceljs.org/)
.cache

# Next.js build output
.next
```



```
# Nuxt.js build / generate output
.nuxt
dist

# Gatsby files
.cache/
# Comment in the public line in if your project uses Gatsby
and not Next.js
# https://nextjs.org/blog/next-9-1#public-directory-support
# public

# vuepress build output
.vuepress/dist

# Serverless directories
.serverless/

# FuseBox cache
.fusebox/

# DynamoDB Local files
.dynamodb/

# TernJS port file
.tern-port

# Stores VSCode versions used for testing VSCode extensions
.vscode-test

# End of https://www.toptal.com/developers/gitignore/api/node
```

Using gitignore.io CLI

If you are a fan of the command-line, the gitignore.io project offers a CLI version as well.

To get it installed on Linux, just run the following command:

```
git config --global alias.ignore \  
'!gi() { curl -sL  
https://www.toptal.com/developers/gitignore/api/$@ ;}; gi'
```

If you are using a different OS, I would recommend checking out the documentation [here](#) on how to get it installed for your specific Shell or OS.

Once you have the `gi` command installed, you could list all of the available `.gitignore` files from gitignore.io by running the following command:

```
gi list
```

For example, if you quickly needed a `.gitignore` file for Laravel, you could just run:

```
gi laravel
```

And you would get a response back with a well-documented Laravel `.gitignore` file:

```
# Created by
https://www.toptal.com/developers/gitignore/api/laravel
# Edit at
https://www.toptal.com/developers/gitignore?templates=laravel

### Laravel ###
/vendor/
node_modules/
npm-debug.log
yarn-error.log

# Laravel 4 specific
bootstrap/compiled.php
app/storage/

# Laravel 5 & Lumen specific
public/storage
public/hot

# Laravel 5 & Lumen specific with changed public path
public_html/storage
public_html/hot

storage/*.key
.env
Homestead.yaml
Homestead.json
/.vagrant
.phpunit.result.cache

# Laravel IDE helper
*.meta.*
_ide_*

# End of
https://www.toptal.com/developers/gitignore/api/laravel
```

Conclusion

Having a **gitignore** file is essential, it is great that you could use a tool like the [gitignore.io](https://www.gitignore.io) to generate your **gitignore** file automatically

depending on your project!

If you like the gitignore.io project, make sure to check out and contribute to the project [here](#).


SSH Keys

There are a few ways to authenticate with GitHub. Essentially you would need this so that you could push your local changes from your laptop to your GitHub repository.

You could use one of the following methods:

- HTTPS: Essentially this would require your GitHub username and Password each time you try to push your changes
- SSH: With SSH, you could generate an SSH Key pair and add your public key to GitHub. That way you would not be asked for your username and password everytime you push your changes to GitHub.

Quick setup — if you've done this kind of thing before

 Set up in Desktop

or

HTTPS

SSH

`git@github.com:bobbyiliev/demo-repo.git`

One thing that you need to keep in mind is that the GitHub repository URL is different depending on whether you are using SSH or HTTPS:

- HTTPS: `https://github.com/bobbyiliev/demo-repo.git`
- SSH: `git@github.com:bobbyiliev/demo-repo.git`

Note that when you choose SSH the `https://` part is changed with `git@` and you have `:` after `github.com` rather than `/`. This is important as this defines how you would like to authenticate each time.

Generating SSH Keys

In order to generate a new SSH key pair in case that you don't have

one, you can run the following command:

```
ssh-keygen
```

For security reasons you can specify a passphrase, which essentially is the password for your private SSH key.

The above would generate 2 files:

- 1 **private** SSH key and 1 public SSH key. The private key should always be stored safely on your laptop and you should not share it with anyone.
- 1 **public** SSH key which you need to upload to GitHub.

The two files will be automatically generated at the following folder:

```
~/.ssh
```

You can use the `cd` command to access the folder:

```
cd ~/.ssh
```

Then with `ls` you can check the content:

```
ls
```

Output:

```
id_rsa  id_rsa.pub
```

The `id_rsa` is your private key, and again you should not share it with

anyone.

The `id_rsa.pub` is the public key which would need to be uploaded to GitHub.

Adding the public SSH key to GitHub

Once you've created your SSH keys, you need to upload the **public** SSH key to your GitHub account. To do so you first need to get the content of the file.

To get the content of the file, you can use the `cat` command:

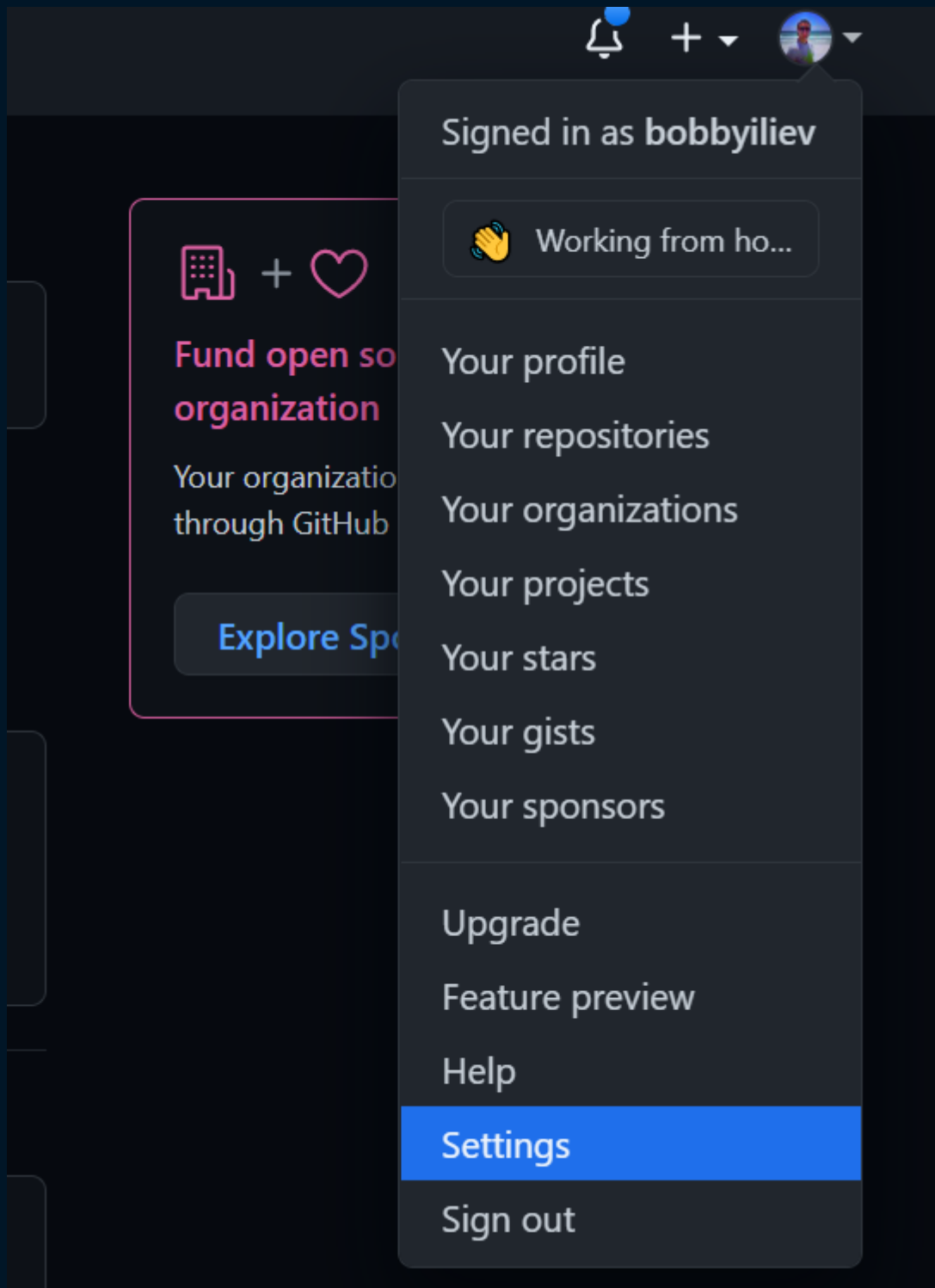
```
cat ~/.ssh/id_rsa.pub
```

The output will look like this:

```
ssh-rsa AAB3NzaC1yc2EAAAADAQAB..... your_user@your_host
```

Copy the whole thing and then visit [GitHub](#) and follow these steps:

- Click on your profile picture on the right top
- Then click on settings

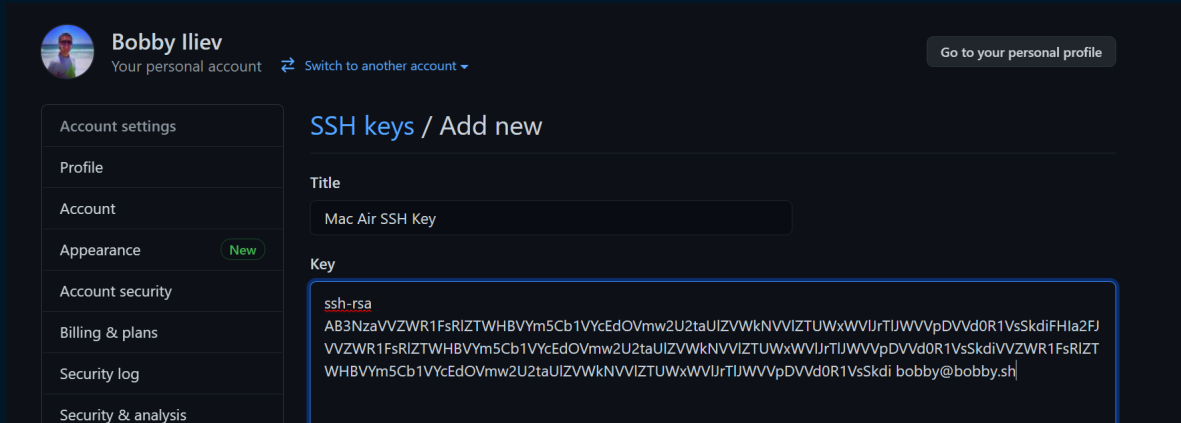


- On the left click on **SSH and GPG Keys**:



- After that click on the **New SSH Key** button

- Then specify a title of the SSH key, it should be something descriptive, for example: **Work Laptop SSH Key**. And in the **Key** area paste your public SSH key:



The screenshot shows the GitHub 'SSH keys / Add new' page. On the left is a sidebar with account settings: Account settings, Profile, Account, Appearance (marked 'New'), Account security, Billing & plans, Security log, and Security & analysis. The main content area has a header 'SSH keys / Add new' and a 'Title' field containing 'Mac Air SSH Key'. Below that is a 'Key' field containing a long public SSH key starting with 'ssh-rsa' and ending with 'bobby@bobby.sh'.

- Finally click the **Add SSH Key** button at the bottom of the page

Conclusion

With that you now have your SSH Keys generated and added to GitHub. That way you will be able to push your changes without having to type your GitHub password and user each time.

For more information about SSH keys, make sure to check this tutorial [here](#).

Git Push

Then finally once you've made all of your changes, you've staged them with the `git add .` command and then you committed the changes with the `git commit` command, you have to push those changes to your remote GitHub repository.

Before you can push to your remote GitHub repository, you would need to first create your remote repository via GitHub as per Chapter 6.

Once you have your remote GitHub repository ready, you can add it to your local project with the following command:

```
git remote add origin  
git@github.com:your_username/your_repo_name.git
```

Note: Make sure to change the `your_username` and `your_repo_name` details accordingly.

This is how you would link your local Git project with your remote GitHub repository.

If you've read the previous chapter, you will most likely notice we are using SSH as the authentication method.

If you did not follow the steps from the previous chapter, you can use HTTPS rather than SSH:

```
git remote add origin  
https://github.com/your_username/your_repo_name.git
```

To do so just use the `git push` command:

```
git push origin main
```

If you are using SSH with your SSH key uploaded to GitHub, the push command will not ask you for a password but it would push your changes to GitHub straight away.

In case that you did not run the `git remote add` command you will get the following error:

```
fatal: 'origin' does not appear to be a git repository  
fatal: Could not read from remote repository.
```

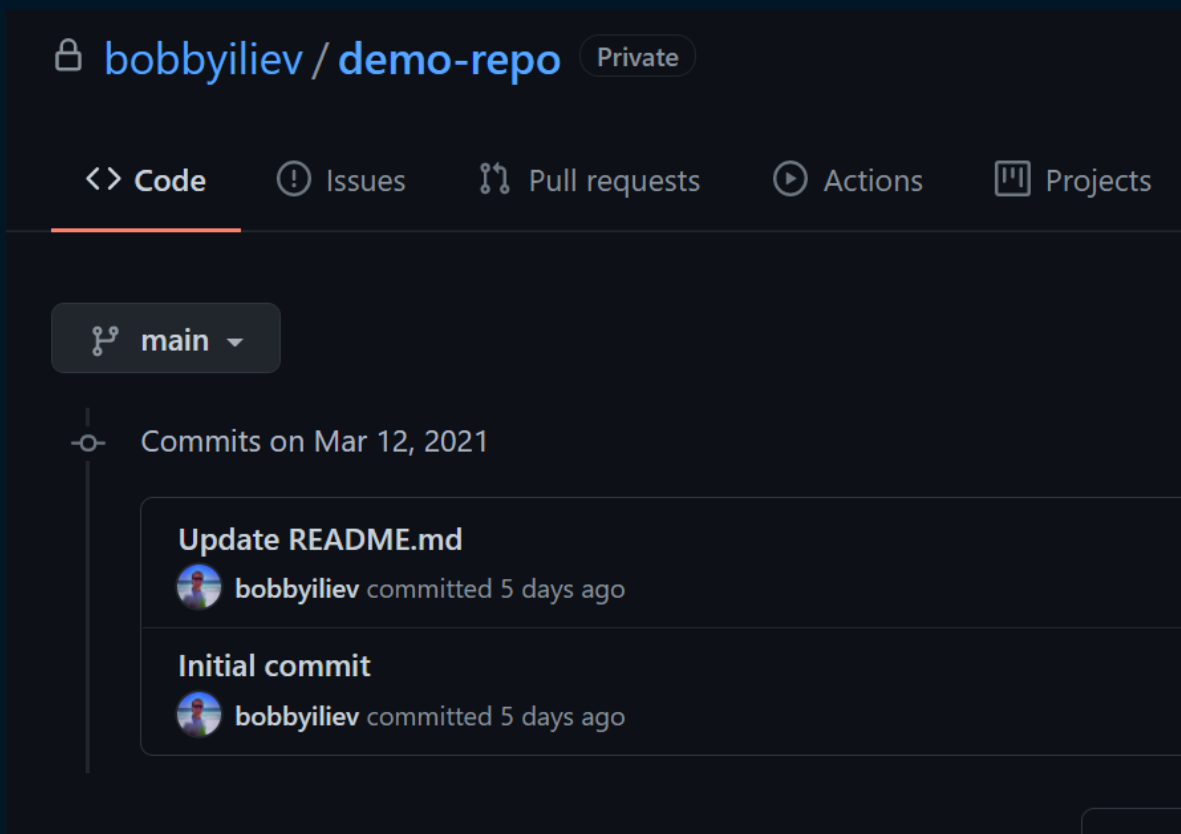
```
Please make sure you have the correct access rights  
and the repository exists.
```

This would mean that you've not added your GitHub repository as the remote repository. This is why we run the `git remote add` command in order to create that connection between your local repository and the remote GitHub repository.

Note that the connection would be in place if you used the `git clone` command to clone an existing repository from GitHub to your local machine. We will go through the `git pull` command in the next few chapters as well.

After running the `git push` command, you can head over to your GitHub project and you will be able to see the commits that you've made locally, present on GitHub. If you were to click on the `commits` link, you will be able to see all commits just as if you were to run the

`git log` command:



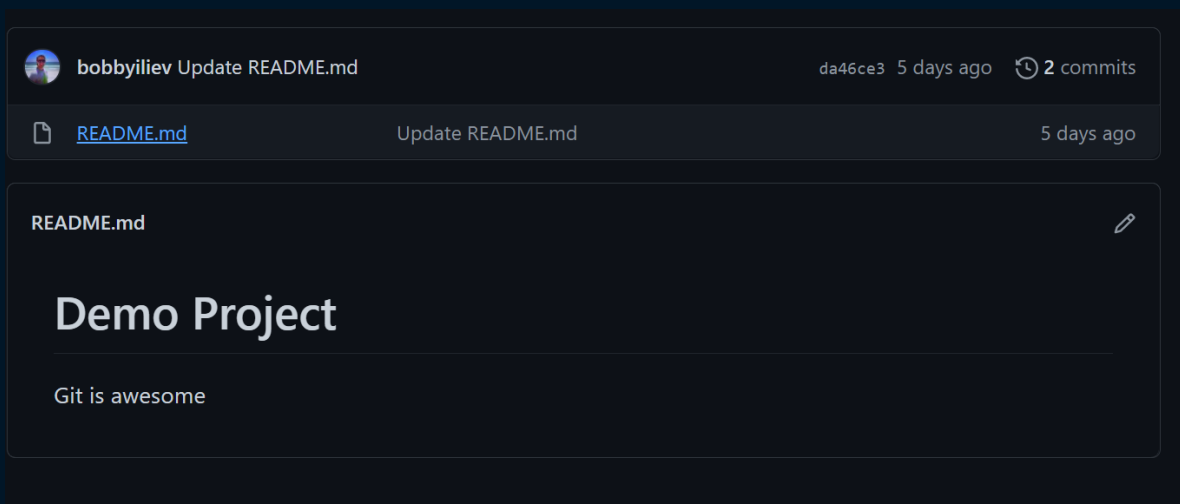
Now that you now how to push your latest changes from your local Git project to your GitHub repository, it's time to learn how to pull the latest changes from GitHub to your local project.

Git Pull

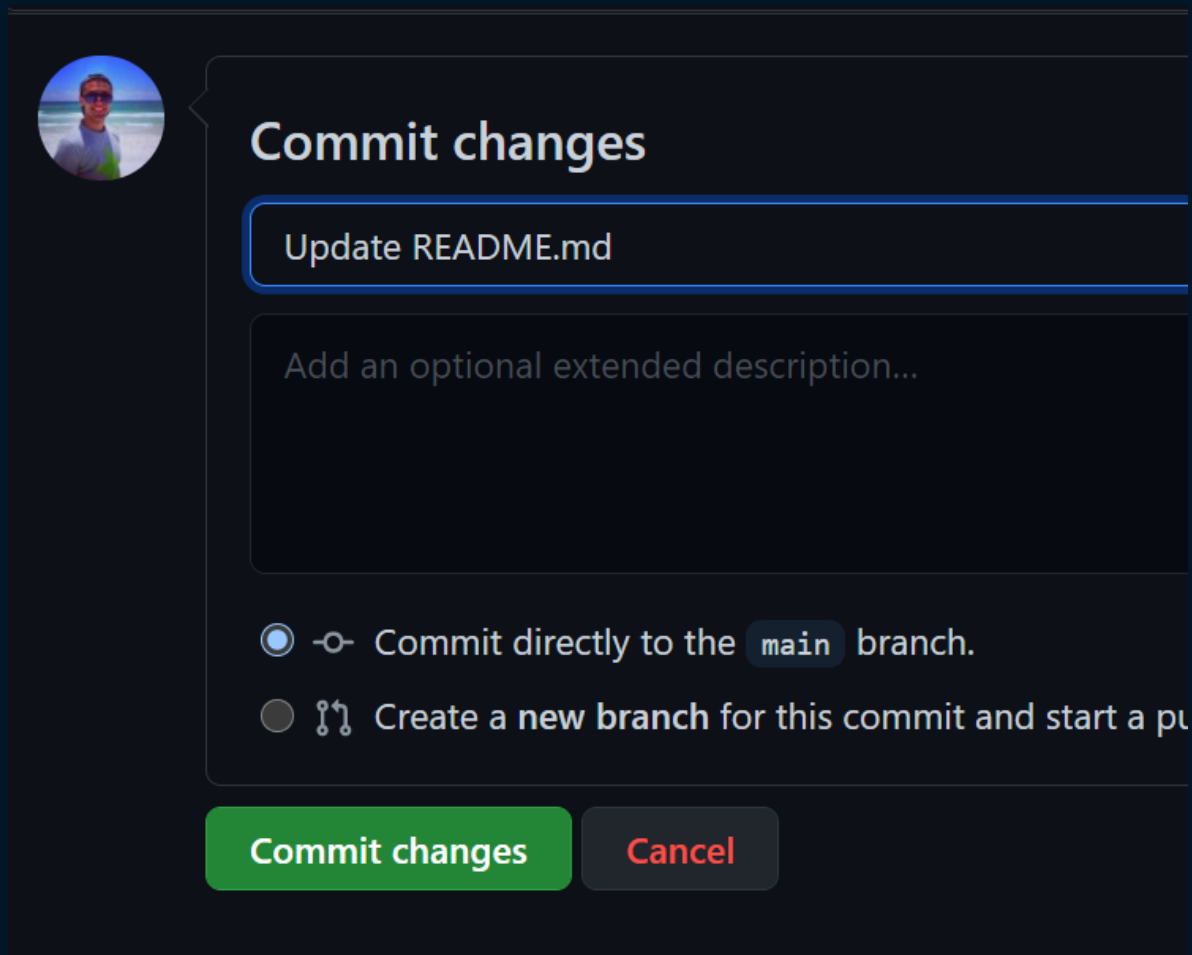
If you are working on a project with multiple people, chances are that the codebase will change very often. So you would need to have a way to get the latest changes from the GitHub repository to your local machine.

You already know that you can use the `git push` command to push your latest commits, so in order to do the opposite and pull the latest commits from GitHub to your local project, you need to use the `git pull` command.

To test this, let's go ahead and make a change directly on GitHub directly. Once you are there click on the `README.md` file and then click on the pencil icon to edit the file:



Make a minor change to the file, add a descriptive commit message and click on the `Commit Changes` button:

A screenshot of the GitHub web interface's 'Commit changes' dialog. On the left is a circular profile picture of a person with glasses and a blue shirt. The main area has a title 'Commit changes' in bold. Below it is a text input field containing 'Update README.md'. Underneath is a larger text area with the placeholder 'Add an optional extended description...'. At the bottom, there are two radio button options: the first is selected and says '-o- Commit directly to the main branch.' (with 'main' in a dark box), and the second is unselected and says '-b- Create a new branch for this commit and start a pull request'. At the very bottom are two buttons: a green 'Commit changes' button and a grey 'Cancel' button.

Commit changes

Update README.md

Add an optional extended description...

☒ -o- Commit directly to the **main** branch.

☐ -b- Create a new branch for this commit and start a pull request

Commit changes Cancel

With that, you've now made a commit directly on GitHub so your local repository will be behind the remote GitHub repository.

If you were to try and push a change now to that same branch, it will fail with the following error:

```
! [rejected]          main -> main (fetch first)
error: failed to push some refs to
'git@github.com:bobbyiliev/demo-repo.git'
hint: Updates were rejected because the remote contains work
that you do
hint: not have locally. This is usually caused by another
repository pushing
hint: to the same ref. You may want to first integrate the
remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help'
for details.
```

As stated in the output, the remote repository is head of your local one, so you need to run the `git pull` command to get the latest changes:

```
git pull origin main
```

The output that you will get will look like this:

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 646 bytes | 646.00 KiB/s, done.
From github.com:bobbyiliev/demo-repo
* branch                main          -> FETCH_HEAD
   da46ce3..442afa5      main          -> origin/main

README.md | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)
```

We can see that the `README.md` file was change and that there were 2 new lines added and 1 line deleted.

Now if you were to run `git log` you will see the commit that you've made on GitHub available locally.

Of course this is a simplified scenario. In the real world, you would not make any changes directly to GitHub, but you would most likely work with other people on the same project and you would have to pull their latest changes on regular basis.

You need to make sure that you pull the latest changes everytime before you try to push your changes.

Now that you know the basic Git commands, in the next chapter we will review a standard Git workflow.

Git Workflow

Git Branches

Reverting changes

Forking in Git

Git Clone

Git And VS Code

As much as I love to use the terminal in order to do my daily tasks in the end I would rather do multiple tasks within one window (GUI) or perform everything from the terminal itself.

In the past, I was using the text editors (vim, nano and etc) in my terminal to edit the code in my repositories and then go along with the git client to commit my changes, but then I switched to Visual Studio Code to manage and develop my code.

I will recommend you to check this article on why you should use Visual Studio. It is an article from Visual Studio's website itself.

[Why you should use Visual Studio](#)

Visual Studio Code has integrated source control management (SCM) and includes Git support in-the-box. Many other source control providers are available through extensions on the VS Code Marketplace. It also has support for handling multiple Source Control providers simultaneously so you can open all of your projects at the same time and make changes whenever this is needed. I personally find this really handy.

Installing VS Code

You need to install Visual Studio Code. It runs on the macOS, Linux, and Windows operating systems.

Follow the platform-specific guides below:

- [macOS](#)
- [Linux](#)
- [Windows](#)

You need to install Git first before you get these features. Make sure you install at least version 2.0.0. If you do not have git installed on your machine feel free to check this really useful article on [How to get started with Git](#)

You need to set your username and email in the Git configuration or git will fail back to using information from your local machine when you commit. We need to provide this information because Git embeds this information into each commit we do.

In order to set this you can execute the following commands:

```
git config --global user.name "John Doe"
git config --global user.email "johnde@domain.com"
```

The information will be saved in your `~/.gitconfig` file

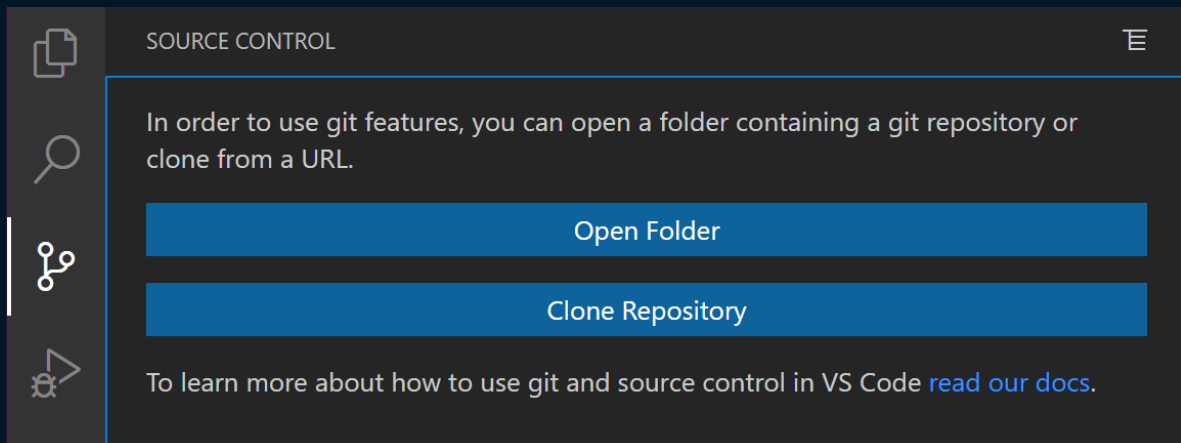
```
[user]
  name = John Doe
  email = johndoe@domain.com
```

With Git installed and set up on your local machine, you are now ready

to use Git for version control with Visual Studio or using the terminal.

Cloning a repository in VS Code

The good thing is that Visual Studio will auto-detect if you've opened a folder that is actually a repository. If you've already opened a repository it will be visible in the Source Control View.



If you haven't opened a folder yet, the Source Control view will give you the options to Open Folder from your local machine or Clone Repository.

If you select Clone Repository, you will be asked for the URL of the remote repository (for example on GitHub) and the parent directory under which to put the local repository.

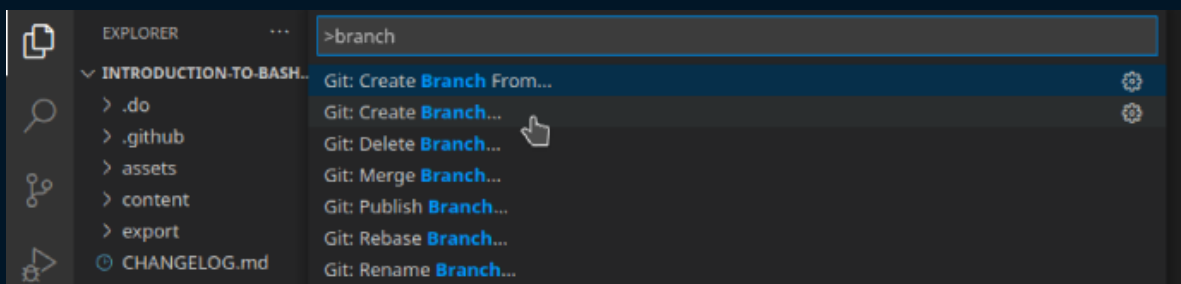
For a GitHub repository, you would find the URL from the GitHub Code dialog.

Create a branch

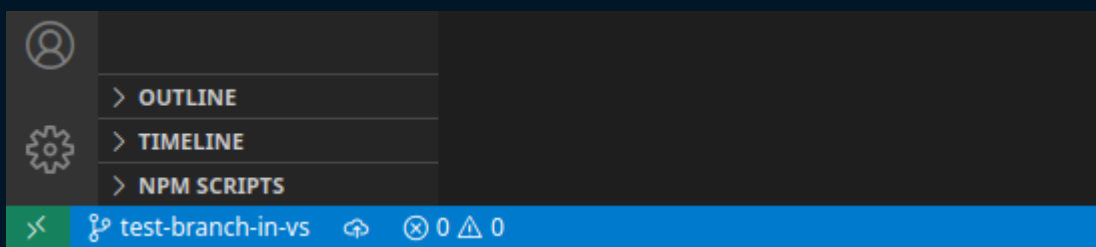
In order to create a branch open the command pallet:

- Windows: Ctrl + Shift + P
- Linux: Ctrl + Shift _ P
- MacOS: Shift + CMD + P

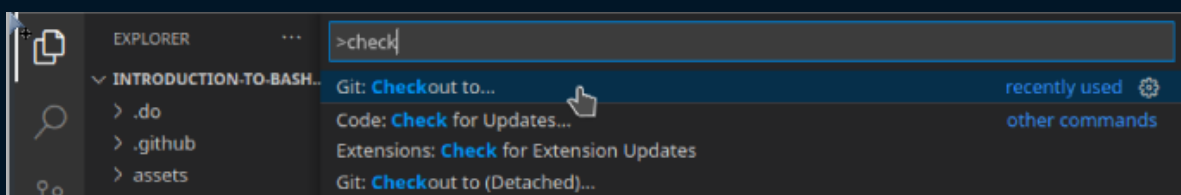
And select **Git Create Branch...**



Then you just need to enter a name for the branch. Keep in mind that in the bottom left corner you can see in which branch you are. The default one will be the main and if you successfully create the branch you should see the name of the newly created branch



If you want to switch branches you can open the command pallet and search for **Git checkout to** and then select the main branch or switch to a different branch.



Setup a commit message template

If you want to speed up the process and have a predefined template for your commit messages you can create a simple file that will contain this information.

In order to do that open your terminal if you're on Linux or macOS and create the following file: `.gitmessage` in your home directory. In order to create the file, you can open it in your favourite text editor and then simply put the default content you would like and then just save and exit the file. Example content is:

```
cat ~/.gitmessage
```

```
#Title
```

```
#Summary of the commit
```

```
#Include Co-authored-by for all contributors.
```

To tell Git to use it as the default message that appears in your editor when you run `git commit` and set the `commit.template` configuration value:

```
$ git config --global commit.template ~/.gitmessage
$ git commit
```

Conclusion

If you prefer to code in Visual Studio Code and you also use version control I will definitely recommend you to give it a go and interact with the repositories in VS code. I believe that everyone has their own style and they might do things differently depending from their mood as well. As long as you can add/modify your code and then commit your changes to the repository there is no exactly correct/wrong way to achieve this. For example you can edit your code in vim and push the changes using the git client in your terminal or do the coding in Visual Studio and then commit the changes using the terminal as well. You're free to do it the way you want it and the way you find it more convenient as well. I believe that using git within VS code can make your workflow more efficient and robust.

Additional sources:

- [Version Control](#) - Read more about integrated Git support.
- [Setup Overview](#) - Set up and start using VS Code.
- [GitHub with Visual Studio](#) - Read more about the GitHub support in VS code
- You can also check this mini video tutorial on how to use the basics of Git version control in Visual Studio Code

Contribured by: [Alex Georgiev](#) Initially posted [here](#).

Conclusion

Congratulations! You have just completed the Git basics guide!

If you found this useful, be sure to star the project on [GitHub](#)!

If you have any suggestions for improvements, make sure to contribute pull requests or open issues.

In this introduction to Git and GitHub eBook, we just covered the basics, but you still have enough under your belt to start using Git and start contributing to some awesome open source projects!

As a next step try to create a GitHub project, clone it locally and push a project that you've been working on to GitHub!

In case that this eBook inspired you to contribute to some amazing open source project, make sure to tweet about it and tag [@bobbyiliev](#) so that we could check it out!

Congrats again on completing this eBook!