SLC-LABS.COM

Duru Elli, Jordan Stella

Git & GitHub Handoff

PDF Download

Purpose

This documentation is designed to be a guide for Juniors joining our DevOps team. It will teach how to utilize Git and Github, the management & workflow of our code, and important practices such as CI/CD which are closely involved in our work.

We will be going over the requirements, the basics explained, the importance of CI/CD, the necessary steps, and common mistakes.

Tools & Requirements

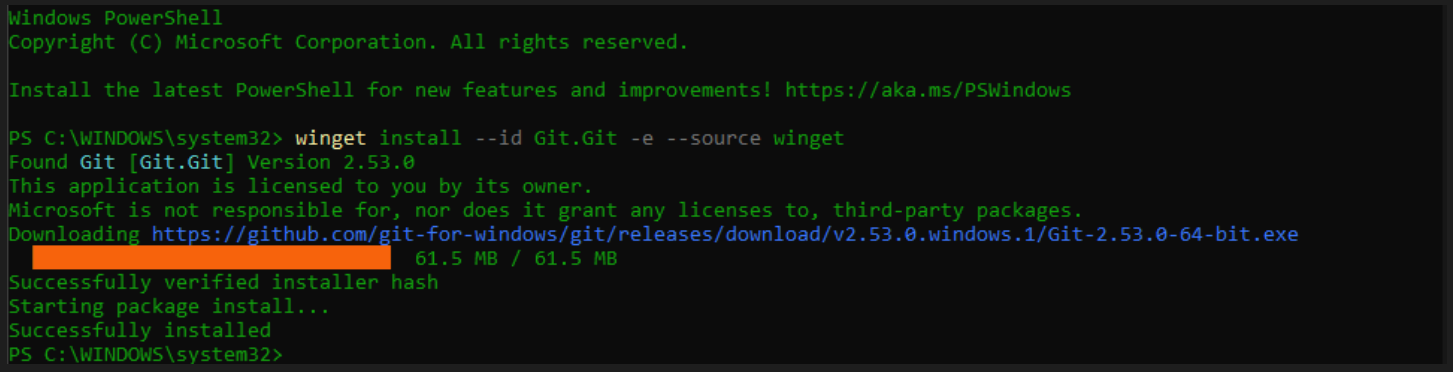
Requirements

• Laptop/PC
• Latest version of Git installed
• GitHub account & access

Installing Git

The simplest way to install Git to the Windows Enviroment is by first opening an Elevated Power Shell Execute the following command inside the Elevated Powershell

```
winget install --id Git.Git -e --source winget
```



For Windows Subsystem for Linux launch the console enviroment
Execute the following command to install Git for the WSL Enviroment

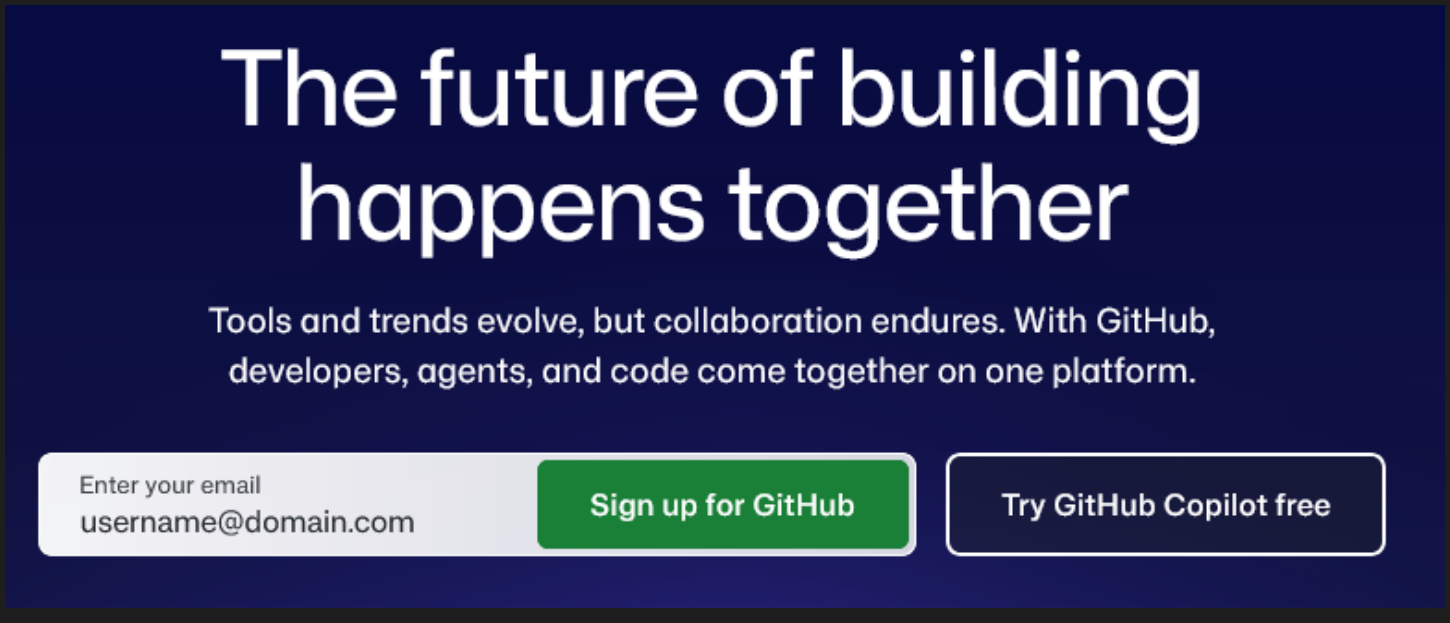
```
sudo apt-get install git
```

```
Ubuntu $
Ubuntu $
Ubuntu $ sudo apt-get install git
[sudo] password for UserName:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn
The following NEW packages will be installed:
  git
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 0 B/3680 kB of archives.
After this operation, 22.2 MB of additional disk space will be used.
Selecting previously unselected package git.
(Reading database ... 39927 files and directories currently installed.)
Preparing to unpack .../git_1%3a2.43.0-1ubuntu7.3_amd64.deb ...
Unpacking git (1:2.43.0-1ubuntu7.3) ...
Setting up git (1:2.43.0-1ubuntu7.3) ...
Ubuntu $
```

Please refer to GitHub's personalized installation guide of Git for additional operating systems located here.

Creating Your GitHub account

Navigate to # GitHub.com
Enter the email you wish to sign up with and click the green "Sign up for GitHub" button.



Fill out the following page with a unique e-mail and username.
When prompted, enter the validation code you have recived in the e-mail you signed up with.

Sign up for GitHub

Continue with Google

Continue with Apple

or

Email*

username@domain.com

⚠ The email you have provided is already associated with an account.
Sign in or reset your password.

Password*

●●●●●●●●●●●●●●

Password should be at least 15 characters OR at least 8 characters including a number and a lowercase letter.

Username*

MyNameGoesHere

⚠ Username MyNameGoesHere is not available.

MyNameGoesHere-ctrl, MyNameGoesHere-arch, or MyNameGoesHere743 are available.

Username may only contain alphanumeric characters or single hyphens, and cannot begin or end with a hyphen.

Your Country/Region*

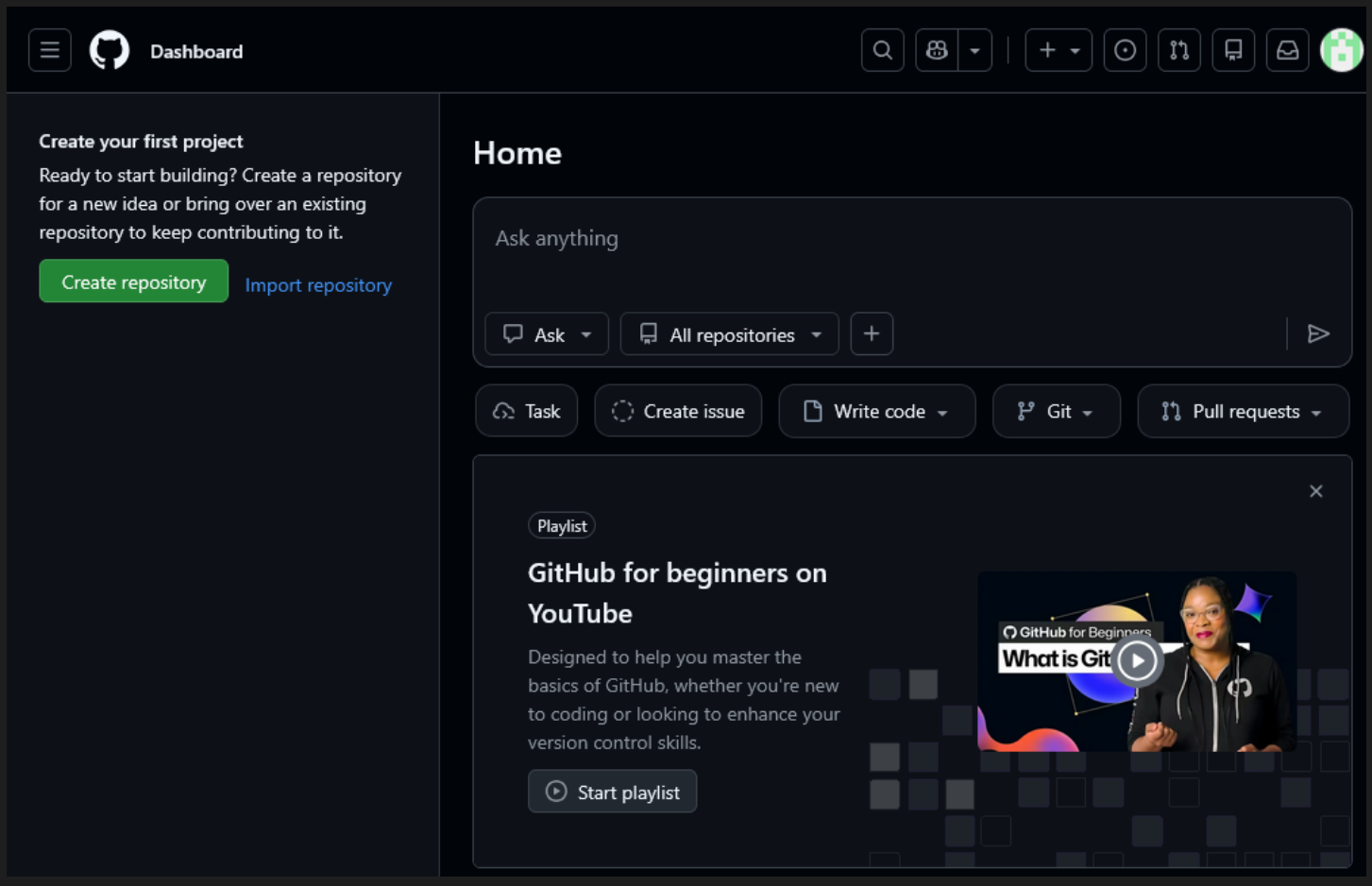
Canada

For compliance reasons, we're required to collect country information to send you occasional updates and announcements.

Create account ›

By creating an account, you agree to the Terms of Service. For more information about GitHub's privacy practices, see the GitHub Privacy Statement. We'll occasionally send you account-related emails.

You have now successfully created your new GitHub Account.



For more details on ACCOUNT CREATION see the Official DOCS

SSH Authentication

Correctly setting up SSH Authentication is vital for proper Git workflow.
To do so we must create our Public and Private SSH keypairs to allow remote access to GitHUb.
Navigate to .ssh located in your home directory, if the directory does not exist you may need to create it first
The following commands apply to both Powershell and WSL
Replace usernamea@yexample.com with the e-mail associated with your GitHub account
when prompted press [enter] x3 as we will accept the default values for this example

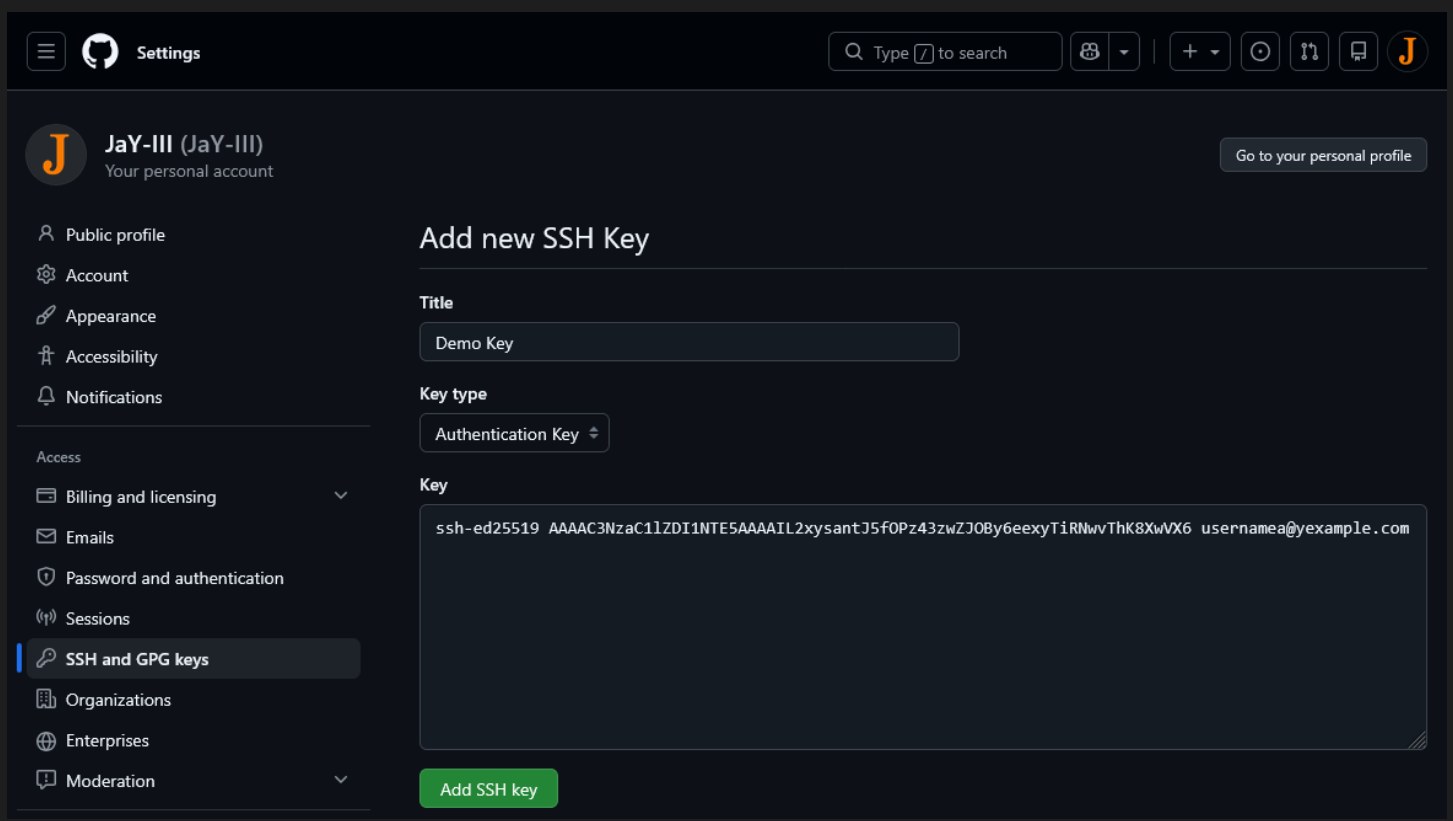
```
mkdir ~/.ssh
cd ~/.ssh
ssh-keygen -C "usernamea@yexample.com"
```

Next we need to view and copy the Public Key we created
Then copy it into our Account SSH setting in Github

```
cat id_ed25519.pub
```

```
Ubuntu $ mkdir ~/.ssh
Ubuntu $ cd ~/.ssh
Ubuntu $ ssh-keygen -C "usernamea@yexample.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/user/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_ed25519
Your public key has been saved in /home/user/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:i4prO8OsVuruP6aY5tLf8mDiblu+s0JvAPfB2u2gJRs usernamea@yexample.com
The key's randomart image is:
+--[ED25519 256]--+
|                 |
|                 |
|   .             |
|. . o            |
| o + o  S        |
|  E.= .. .       |
| =+O+o. .        |
|o*%O=+.          |
|%@@@B*o          |
+----[SHA256]-----+
Ubuntu $ cat id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIL2xysantJ5fOPz43zwZJOBy6eexyTiRNwvThK8XwVX6 usernamea@yexample.com
Ubuntu $
```

Click your user Icon on the top right of GitHub -> Settings ->
On the following page on the left side click -> SSH and GPG keys ->
Finally on the right side the Green button -> New SSH Key ->
Paste in the public key you just prior copied -> Add SSH key



Finally we will test our key with the following command
type yes when prompted

```
ssh -T git@github.com
yes
```

```
Ubuntu $ ssh -T git@github.com
The authenticity of host 'github.com (140.82.112.3)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com' (ED25519) to the list of known hosts.
Hi JaY-III! You've successfully authenticated, but GitHub does not provide shell access.
Ubuntu $
```

Basics

Git

Git is a distributed version control system (DVCS). This provides many features such as to track the contribution and code changes of multiple collaborators, allow rollbacks to earlier version and view history, and also work from a local environment with an easy connection to the remote repository. Most of our work will be directly involved with Git.
Git Official DOCS

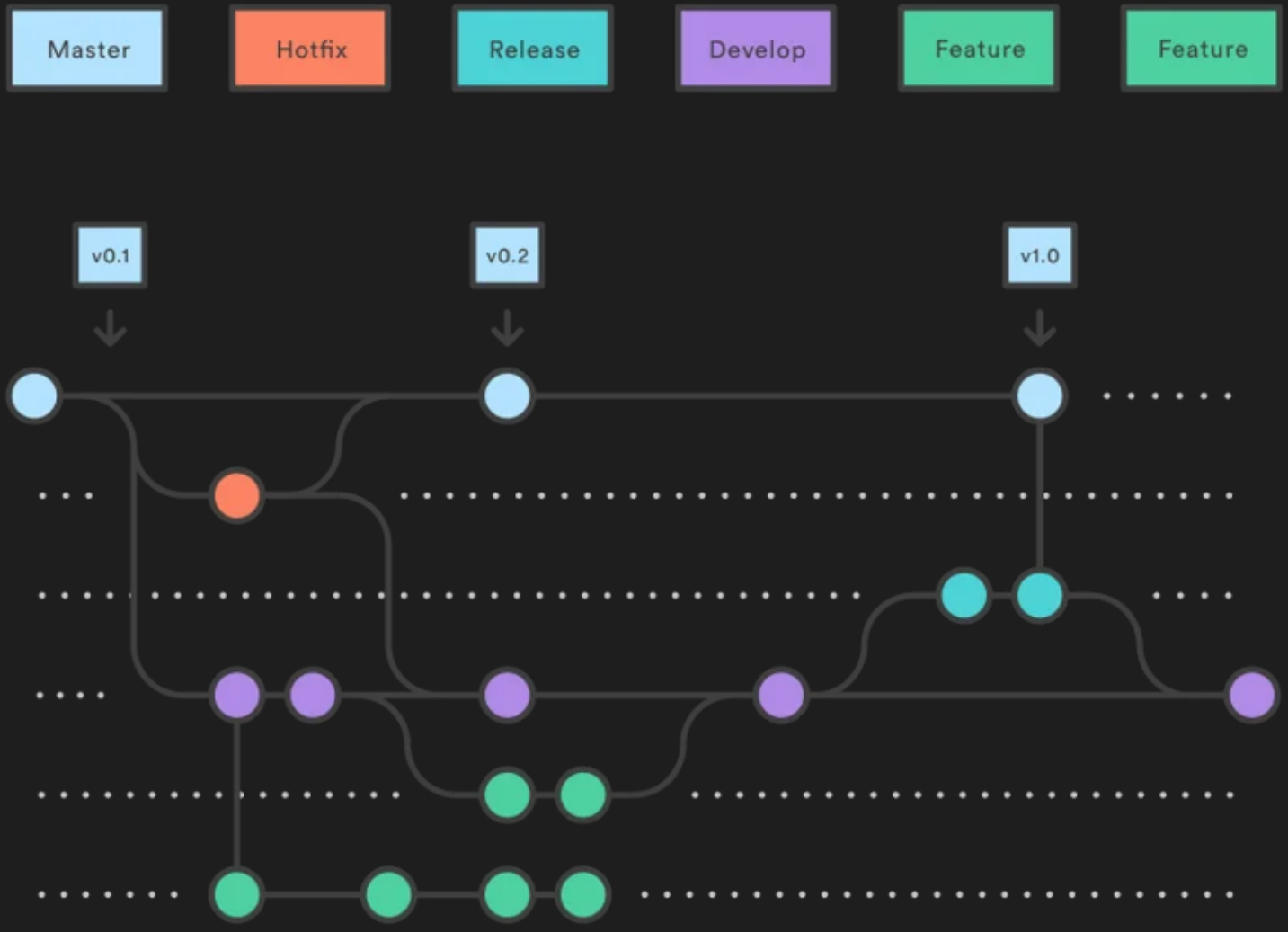
GitHub

GitHub is a server where remote Git repositories are stored. It connects developers to a cloud repository regardless of timezone and distance. Developers are able to easily clone the cloud repository to their machine and begin working on their local repository. When they have completed their current goal, they are able to easily push their local repository back up to the cloud repository and update it seamlessly. Additionally, GitHub's website serves as a visual method of viewing the repository and all the files contained, making changes, pushing/pulling.

GitHub Official DOCS

Git Workflow

Understanding the Git Workflow proccess, including the features and difficulties associated with the proccess.



Git Workflow is a thorough process generally used by large teams/projects to maintain their release standards. It includes more used branches than general GitHub Workflow due to the importance of good quality content and still be efficient as well. It is a clearly structured design where there are many branches for their own unique purpose which we will go over.

The branches included are: 'main', 'develop', 'feature/*', 'release/*', 'hotfix/*'

Main is the most important branch, it is crucial to not send any updates that are possibly fault to this branch and even more important to never commit immediately to main branch. This is our production-ready branch where it must be functioning as intended with zero bugs intentionally released.

Develop is the branch designated for branching & integrating feature branches with active development. It is derived from the main branch. Here we confirm that the features will function properly and verify the code is consistent and reliable.

Feature is designed for building new features, testing purposes, or bug-fixing. They are branched out from the develop branch and will be merged back into the develop branch again when it is complete and ready for the verification proccesses.

Release is when our feature branches from the develop branches are verified and ready to undergo QA, bug fixes, and smaller updates. This is branched from a progressed develop branch, it is the last branch before the code is confidently integrated into main and ready for release.

Hotfix is when an important issue is found and needs to be fixed as soon as possible. It is branched from main and then pushed back into main when the hotfix is ready to be deployed. It allows code production to be very quick and efficient and is designed to be a quick fix for whatever the issue is.

CI/CD connection

CI/CD (Continuous Integration / Continuous Development) is a crucial practice deeply involved in our work process. It comes with many principles, which we will frequently utilize within our workspace.

A clear example of one of the main principles is Version Control simply through us utilizing Git. Since Git allows us to track all changes, review code easily, and allows rollbacks, and allows multiple developers to work altogether. It is essential for our workflow as it enables us to collaborate effectively and efficiently while maintaining history and easy rollbacks for potential issues we may come across.

One of the most crucial ones is Continuous Integration. By committing code often, bugs will be found quicker, changes are manageable and easier to integrate. This is an important principle that must be constantly followed as it greatly impacts our efficiency and the overall health of our codebase by making code easier to work with at each step whether it is troubleshooting or validating.

Another core principle is to Build in Quality, which is essential to ensure that we prioritize quality before releasing any code to the main branch. By doing quality assurance tests and bug fixing in the release/develop branch, we will be confirming that our code meets quality expectations and prevents issues from arising after the code is already released. This means we will always be doing pull requests and reviewing code before pushing and committing it to the next stage.

"Done" means released. This principle ensures that when code is ready to be released, it is released. No time is wasted and the code is delivered to the next stage always. We will deliver good quality work at every stage and as soon as possible. When a commit is ready and has been properly reviewed, it will be moved onto the next stage without wasting any time. Response times will be short and allow our flow to be as efficient as possible.

Our Workflow

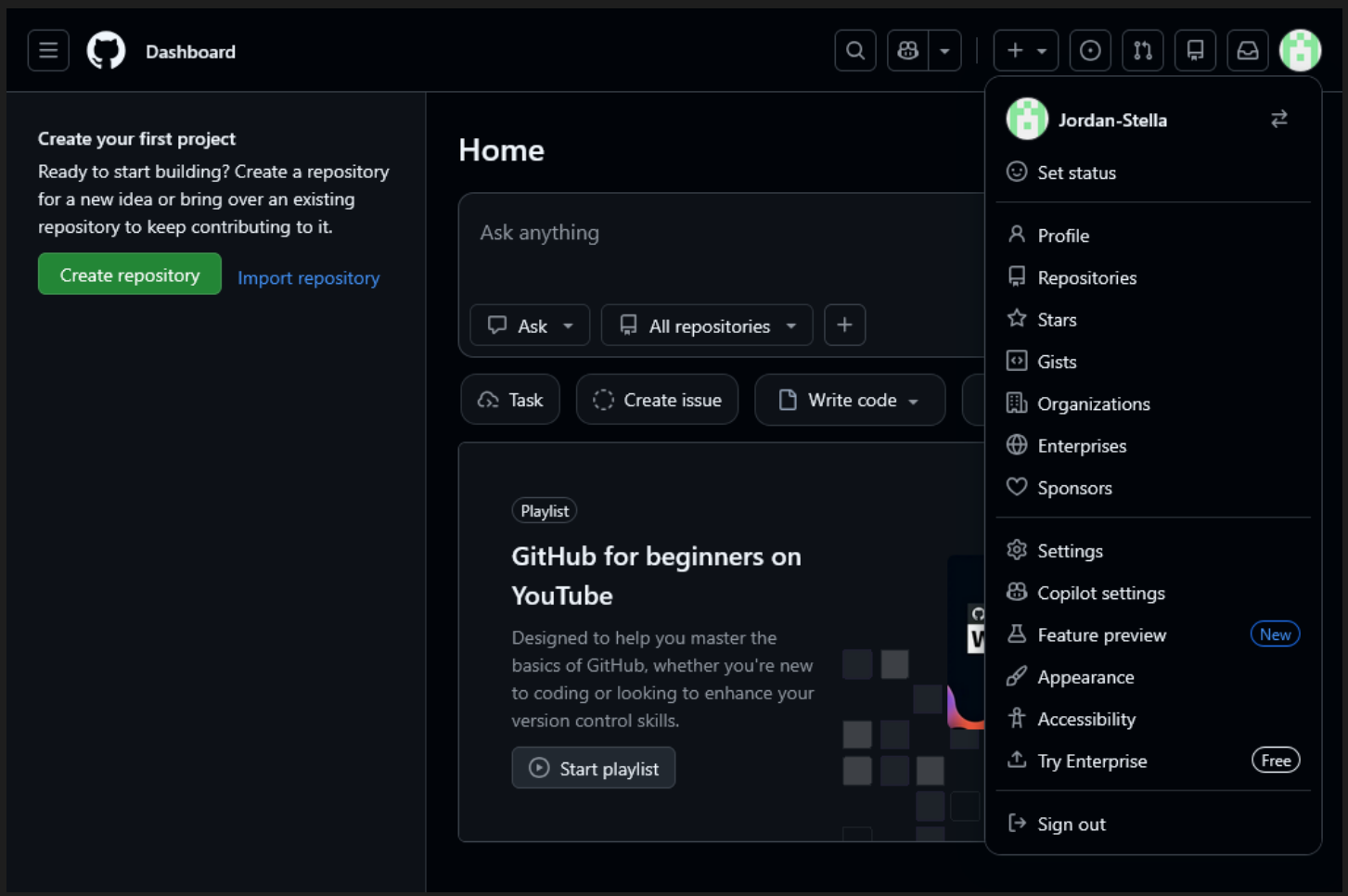
Repositories

Repositories are the main location where the code, files, and revision history is all stored. The source repository is generally the remote repository located on the cloud, and specifically in our case, stored in GitHub servers.

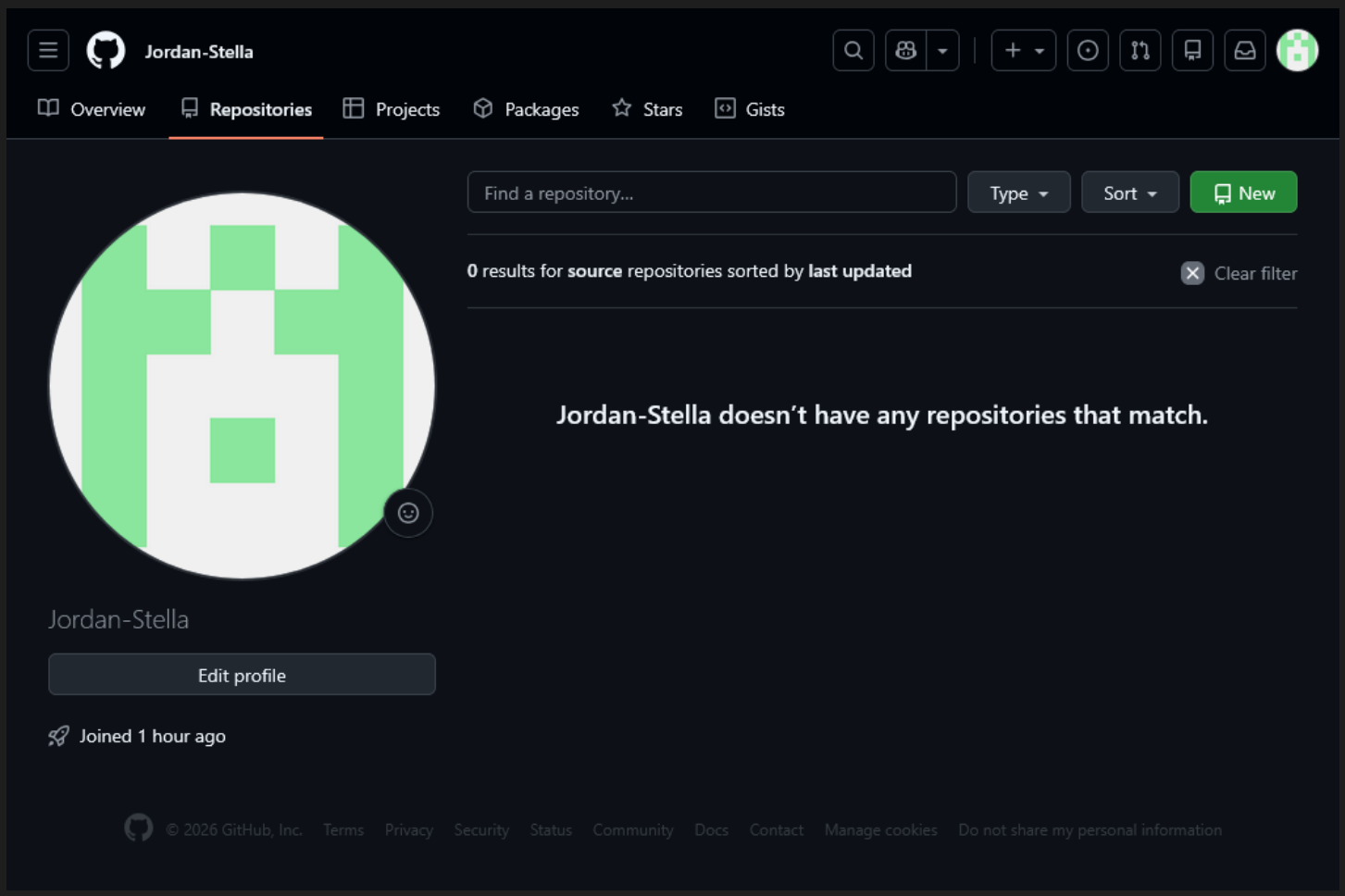
When you clone a repository, you have an identical copy of the repository where you can make your own changes and commit to your own repository without issues. You will also be able to push your local copy of the repistory up to the cloud repository you cloned it from. This is how you would push

changes to the main codebase. We will be going further into detail with the upcoming sections. First, we will go over making your own repository.

GitHub makes it easy to create your first repository.
However, we are going to follow the procedure to add a new repo that will work for the second and beyond
Simply click on your user profile picture on the top right followed by "Repositories"

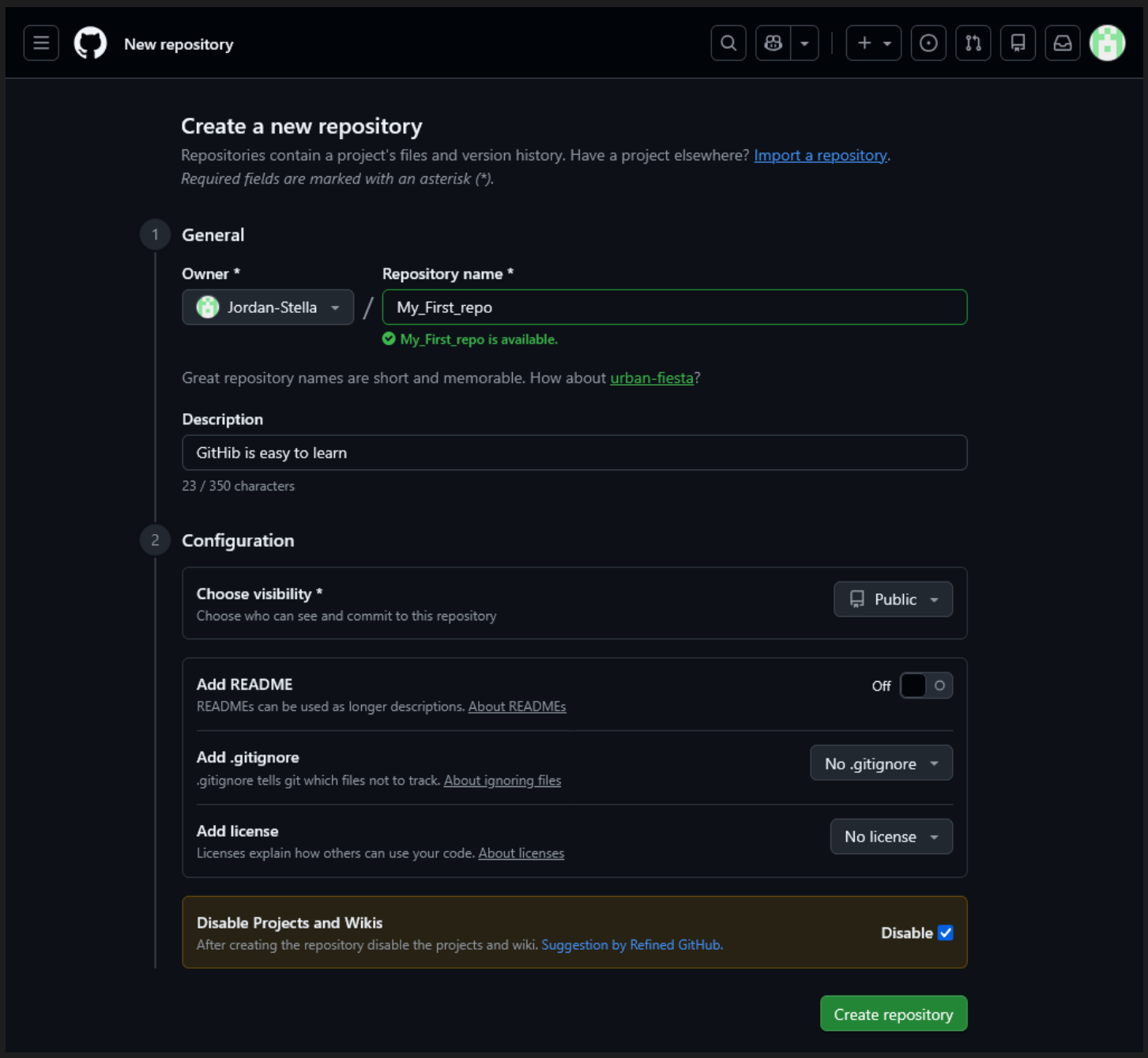


Now click the Green "New" Button on the right side of the screen under your profile picture



Here we need to give our Repository a Unique name, an optional Description, and the Visibility as Public or Private.
Optionally you may choose the license to release your files under.

Click the green "Create repository" button on the right and you are done.



For more details on REPOSITORY CREATION see the Official DOCS

Pull

The first time we are working with a repo we will be required to use the Clone function to do so execute the following:

```
git clone https://github.com/<username>/<repository>.git
```

```
  git clone https://github.com/Duru-E/Git-Github-Handoff.git
```

```
PS C:\temp\devops> git clone https://github.com/Duru-E/Git-Github-Handoff.git
Cloning into 'Git-Github-Handoff'...
remote: Enumerating objects: 88, done.
remote: Counting objects: 100% (88/88), done.
remote: Compressing objects: 100% (86/86), done.
remote: Total 88 (delta 40), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (88/88), 1.10 MiB | 6.88 MiB/s, done.
Resolving deltas: 100% (40/40), done.
PS C:\temp\devops>
```

The next time we want to work on the repo we can execute a Pull request to ensure we have the latest version

```
git pull
```

```
PS C:\temp\devops> cd .\Git-Github-Handoff\
PS C:\temp\devops\Git-Github-Handoff> git pull
Already up to date.
PS C:\temp\devops\Git-Github-Handoff>
```

We can also pull a specific branch to work on
Before we do that let's verify our current branch we are working in

```
git branch
```

```
PS C:\temp\devops\Git-Github-Handoff> git branch
* main
PS C:\temp\devops\Git-Github-Handoff>
```

As expected we are working in Main.

Next we are going to change to the branch we want to work on
git checkout
then verify with git branch we have changed to the intended branch

```
git checkout WorkInProgress
git branch
```

```
PS C:\temp\devops\Git-Github-Handoff> git branch
* main
PS C:\temp\devops\Git-Github-Handoff> git checkout WorkInProgress
branch 'WorkInProgress' set up to track 'origin/WorkInProgress'.
Switched to a new branch 'WorkInProgress'
PS C:\temp\devops\Git-Github-Handoff> git branch
* WorkInProgress
  main
PS C:\temp\devops\Git-Github-Handoff>
```

Finally we will ensure we are on the most current version with the following

```
git pull <remote> <branch>
```

As we have already cloned our repository we can replace the (https://github.com/Duru-E/Git-Github-Handoff.git) with "origin"

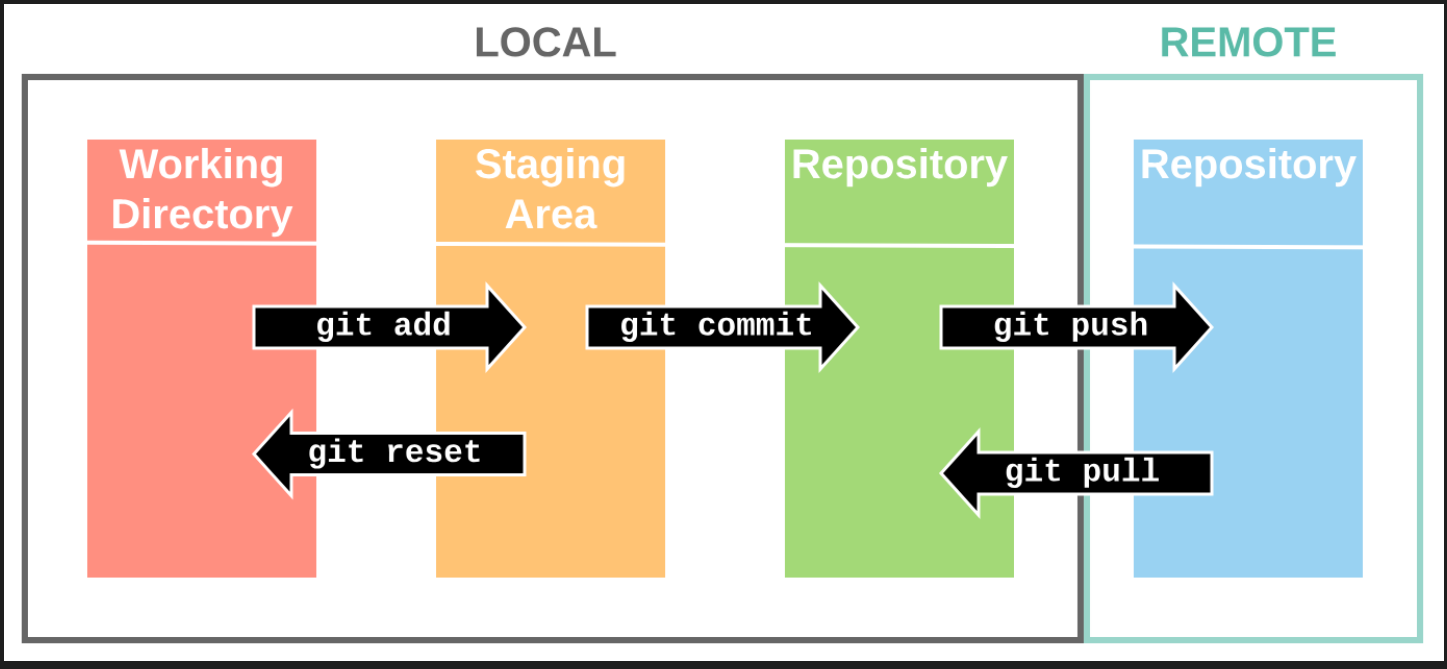
```
git pull origin <branch>
```

```
git pull origin WorkInProgress
```

```
PS C:\temp\devops\Git-Github-Handoff> git checkout WorkInProgress
Already on 'WorkInProgress'
Your branch is up to date with 'origin/WorkInProgress'.
PS C:\temp\devops\Git-Github-Handoff> git pull origin WorkInProgress
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 1.24 KiB | 423.00 KiB/s, done.
From https://github.com/Duru-E/Git-Github-Handoff
 * branch            WorkInProgress -> FETCH_HEAD
   f2c62b4..4cf2c1e  WorkInProgress -> origin/WorkInProgress
Updating f2c62b4..4cf2c1e
Fast-forward
 README.md | 26 ++++++++++++++++++++------
 1 file changed, 20 insertions(+), 6 deletions(-)
PS C:\temp\devops\Git-Github-Handoff>
```

For more details on PULL see the Official DOCS

Staging



Understanding Staging in GitHub

Staging in GitHub is the preparation of changes before committing them to a repository. We can do this easily through Git, where we use a quick and simple command to add our changes to the Staging Area. We will start with the local repository in order to add changes.

First is the local repository which we retrieve by cloning the remote cloud repository to our machine using the following command:

```
git clone <repository-url>
```

At this stage we will be making our changes to the code and files.
Now to stage our changes and set them in a prepared state, we will use the following command:

```
git add <file-name>
```

```
Ubuntu $ git checkout -b pushDemo
fatal: a branch named 'pushDemo' already exists
Ubuntu $ git branch
  main
* pushDemo
Ubuntu $ nano README.md
Ubuntu $ git add README.md
Ubuntu $
```

Or to stage all changes, use:

```
git add .
```

```
Ubuntu $ git checkout -b pushDemo
fatal: a branch named 'pushDemo' already exists
Ubuntu $ git branch
  main
* pushDemo
Ubuntu $ nano README.md
Ubuntu $ git add README.md
Ubuntu $ git add .
Ubuntu $
```

Now that all the changes are Staged, we are ready to Commit the changes to the Branch.
After this completed we will Push this Commit upstream shortly afterwards:

```
git commit -m "Your commit message"
```

```
Ubuntu $ git commit -m "Updateded README.md + images "
[pushDemo bf3448f] Updateded README.md + images
 Committer: User <user@pixie.porkbun.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 71 insertions(+), 35 deletions(-)
Ubuntu $
```

Staging is essential for Git workflow, it allows you to easily pick and choose which changes are ready to be committed and sent to the repository.

For more details on STAGING see the Official DOCS

Push

Before we can push we need to tell git the location to send the push to.
This is done with the following:

```
git remote set-url origin git@github.com:<username>/<repo>.git
```

The username is the REPO OWNERS username and may not be your own

```
git remote set-url origin git@github.com:Duru-E/Git-Github-Handoff.git
```

```
Ubuntu $ git remote set-url origin git@github.com:Duru-E/Git-Github-Handoff.git
Ubuntu $
```

After all your changes to the active branch have been made it is time to Push them to the repo
This is done with the following command:

```
git push origin <branch>
```

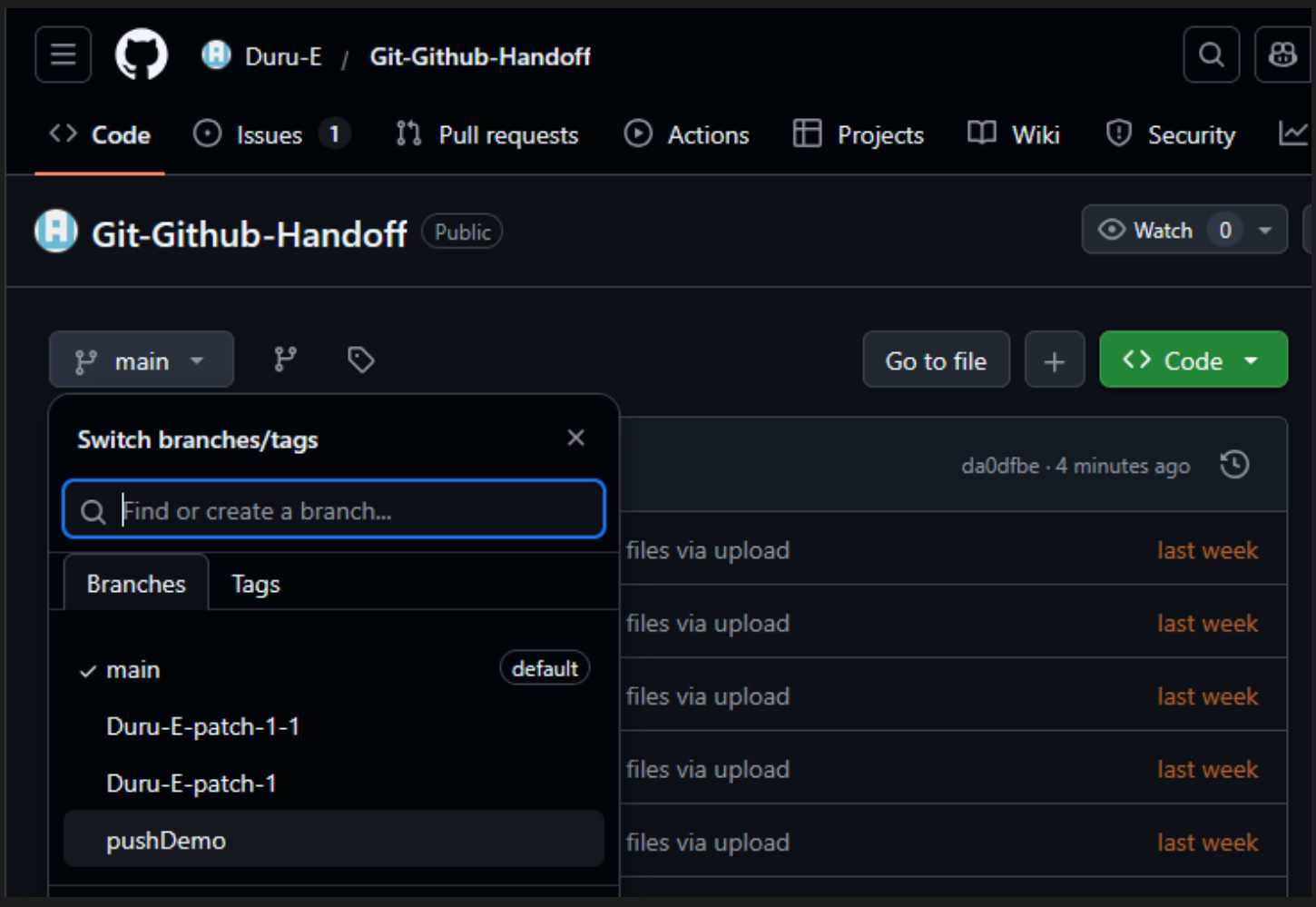
```
git push origin pushDemo
```

```
Ubuntu $ git push origin pushDemo
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 32 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 801 bytes | 801.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:Duru-E/Git-Github-Handoff.git
   da0dfbe..bf3448f  pushDemo -> pushDemo
Ubuntu $
```

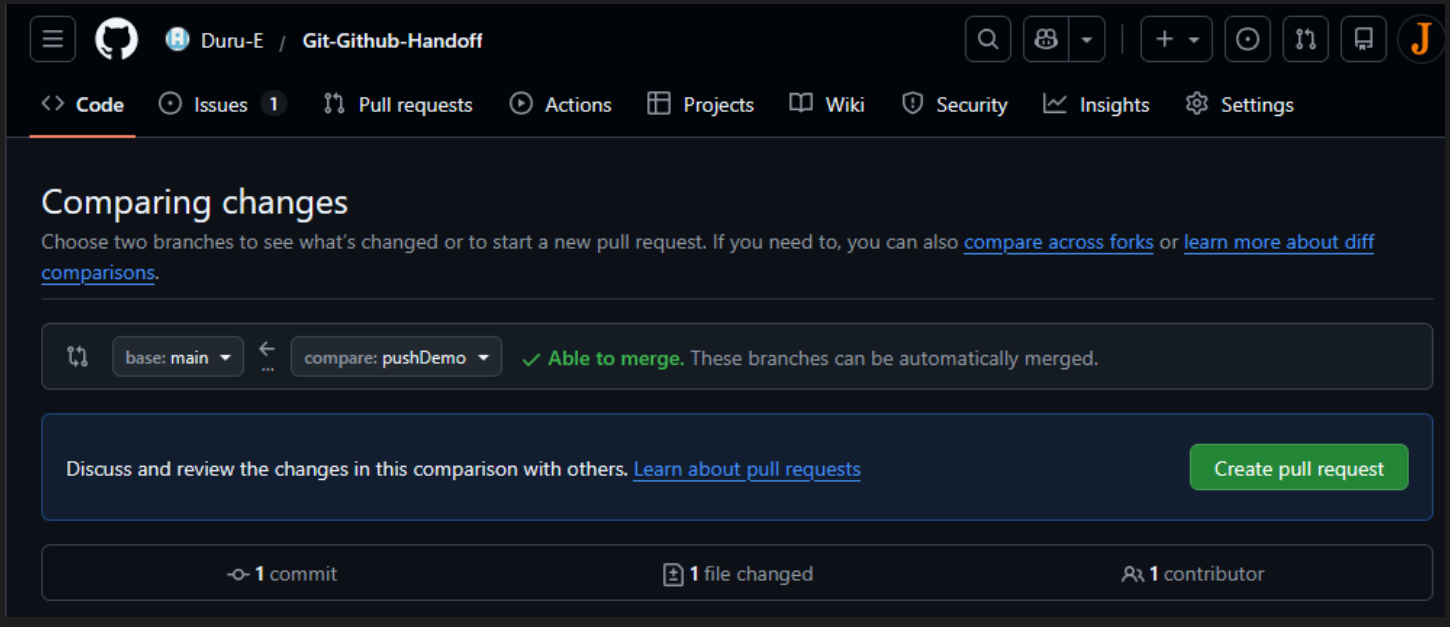
For more details on PUSH see the Official DOCS

Commits

To complete a Commit from a branch into your main Repo you must first Select the branch in GitHub



Then click on the Blue text stating X Commits ahead of main to review the changes



Once the changes have been Validated you may click -> Create pull request
 COMMIT

For more details on COMMITS see the Official DOCS

Common Issues

Git Authentication Errors

If you are coming across the error from Git that says "Author identity unknown", this is a result of Git being unable to identify the user information for commits. You can solve this issue easily by typing the following two commands:

```
git config --global user.name "Insert your username"
git config --global user.email "your.email@example.com"
```

Merge Conflicts

Merge conflicts are when two branches modify the same code and Git is unable to automatically merge the branches and forces manual input to resolve the issue. If this conflict issue is not resolved properly, it could possibly result in overwritten code, create new bugs, reintroduce fixed solutions, or remove features. This will inevitably incur more work for all of the developers involved and result in more delays for release. It will require solutions and possiblly rollbacks to resolve and will require one of the conflicting branches to be completely adapted to the conflicting branch. This issue can easily be prevented by properly discussing with fellow developers regarding the planned code changes, frequently pulling from the main branch, and following CI/CD by making small but frequent commits.

For more details on RESOLVING MERGE CONFLICTS see the Official DOCS

Committing to Main

Committing directly to main is extremely unsafe and likely to cause more issues. The code in main is the source code for the entire project and an unvalidated addition may bring errors, bugs, vulnerabilities, and drop in quality. When the code in main is corrupted in any manner, it will drastically impact every other environment in the workspace since work is pulled and updated from main. If undetected, it will potentially corrupt the rest of the work being done and cause major delays when it is discovered by all the hotfixes and patches that will be necessary to resolve the root cause. It will additionally cause further issues when it comes to end users receiving the production-ready code from main, as it will not have been validated and gone through quality assurance. Thus, be sure to always confirm what branch you are changing and where you will be committing your changes to.

Committing to Wrong Branches

By accidentally comitting to the wrong branch and not confirming what branch you are currently on, workflow will be disrupted and delayed until the necessary cleanup or potential rollback is performed. This will cause further issues not only with the person who committed the new code but also those that were already performing work on the branch that had the code changed to. It will disrupt multiple environments and require more effort for the resolution and changes to be made, even moreso if the issue is not immediately caught. As in that case, the incorrect code will be slowly built upon by correct

code that will most likely cause further issues and conflict errors. Be sure to confirm each time which branch you are using and where you are committing your changes.

Forgetting to Pull before Push

Simply forgetting to pull the latest version of the repository before releasing your version could cause many issues. For example, if an important bug-fix is released after you pulled your close and you forget to update, your version will not contain this bug-fix. This can cause major issues in the main repository and it could even cause further bugs due to possible conflicts. It is important to always pull the latest version of the repository to ensure your repository is not missing any key updates and will work efficiently with the current version. Always be certain to make sure you are using the most current version by using the command `git status` .

In this example below, branch fell behind main and needed to Merge with Main before Push was even allowed to process.

```
Ubuntu $ git pull git@github.com:Duru-E/Git-Github-Handoff.git
From github.com:Duru-E/Git-Github-Handoff
 * branch            HEAD       -> FETCH_HEAD
Already up to date.
Ubuntu $ git checkout main
M       README.md
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
Ubuntu $ git pull
From github.com:Duru-E/Git-Github-Handoff
   e5d72f5..da0dfbe  main       -> origin/main
Updating e5d72f5..da0dfbe
Fast-forward
 Push_A.png    | Bin 0 -> 29868 bytes
 Push_B.png    | Bin 0 -> 15660 bytes
 Staging_A.png | Bin 0 -> 25043 bytes
 Staging_B.png | Bin 0 -> 27433 bytes
 4 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Push_A.png
 create mode 100644 Push_B.png
 create mode 100644 Staging_A.png
 create mode 100644 Staging_B.png
Ubuntu $ git checkout pushDemo
M       README.md
Switched to branch 'pushDemo'
Ubuntu $ git push origin main
Everything up-to-date
Ubuntu $ git branch
  main
* pushDemo
Ubuntu $ git merge main
Already up to date.
Ubuntu $ git push origin pushDemo
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Duru-E/Git-Github-Handoff.git
   e5d72f5..da0dfbe  pushDemo -> pushDemo
Ubuntu $
```

Forgetting to Delete Merged Branches

Forgetting to delete branches that have already been merged may not seem like a big issue, however, it builds up more work over time. Instead of immediately clearing the branch, leaving a branch active when it is inactive will cause clutter, confusion and could result in errors and more conflicts. Developers may accidentally continue work or merge onto an unused/outdated branch and only realize when it is too late. And, it will take more effort to identify and confirm that the branch is inactive rather than immediately deleting it after it is merged.