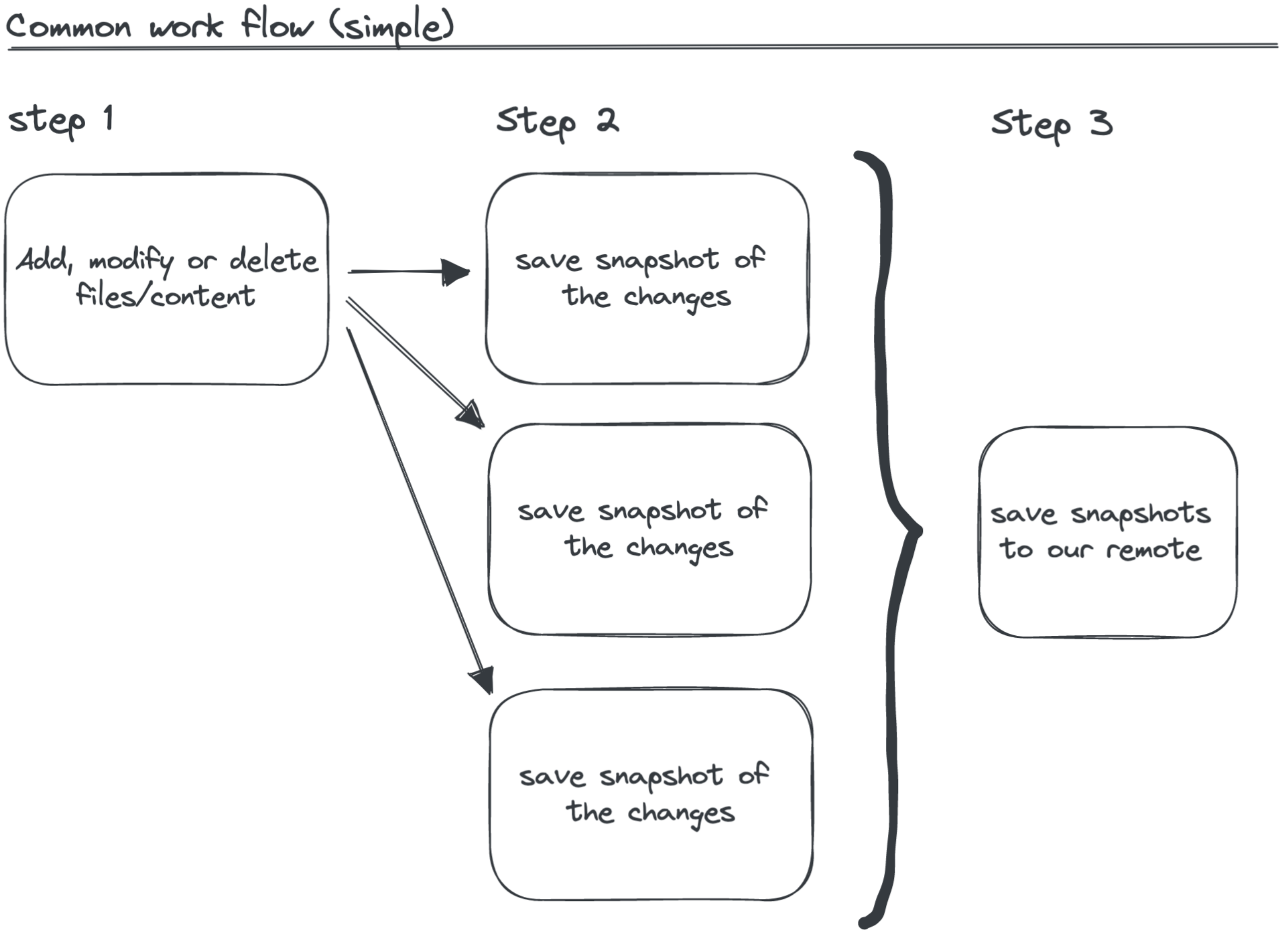
Working with Git

This document describes the fundamentals of working with git connected to an Azure DevOps Repository.

Getting started, we'll discuss a simple workflow


Common work flow (simple)
step 1
Step 2
Step 3
Add, modify or delete files/content
save snapshot of the changes
save snapshot of the changes
save snapshot of the changes
save snapshots to our remote

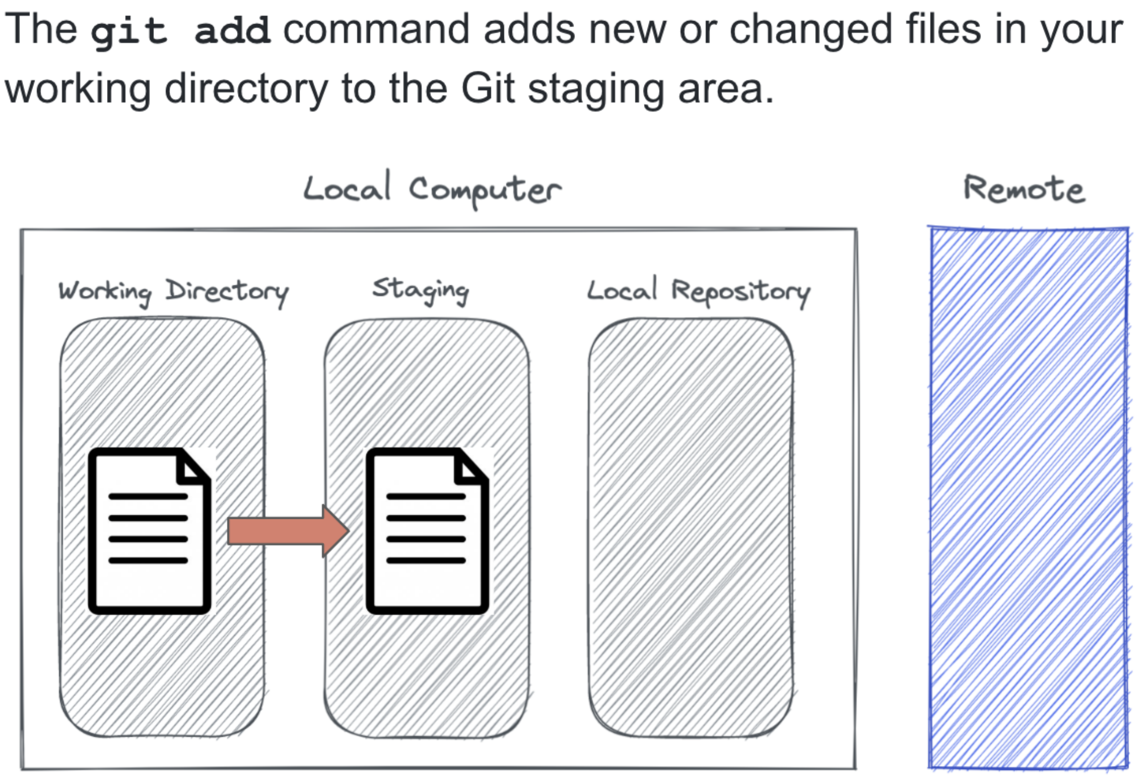
Step 1: Add, modify or delete files/content

Generally, when working on a project, we create/delete files, add/modify content and save our work. These changes are made within a working folder with or without subfolders. The working folder is the git repository and any changes to any of the files within that folder (including subfolders) are tracked; noting anything that changes.

Step 2: Save a snapshot of the changes

There are two commands (or actions) we need to execute to save a snapshot; add our files into the staging area and commit them to our local repository.

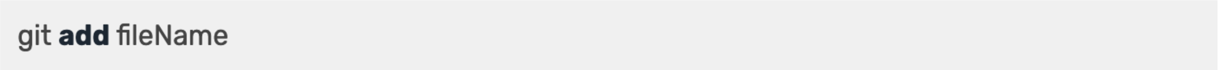
1. **Adding files into the staging area**
   Often, we reach a milestone and want to create a snapshot to save the current state of the changes we have made. Using Git terminology, this is called a commit. To do this, we first tell Git which files we want to include in our snapshot; we do this using the '**git add**' command.



The `git add` command adds new or changed files in your working directory to the Git staging area.

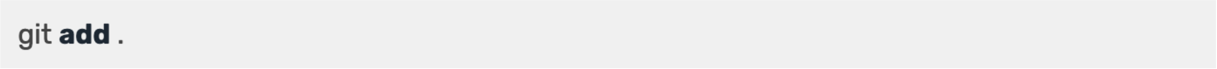
When we add files using the **git add** command, we tell git which files we want to include in our snapshot - our commit. When we add files using the **git add** command, we are adding them into a **staging area** (shown above). Anything that is added to the staging area (files, folders, deleted files, etc) are included when we **commit** (explained in Step 3 below). So, how do we do this?

To add a single file or a filename pattern (i.e. *.html)



```
git add fileName
```

To add all of the changes you have made (including file deletions)



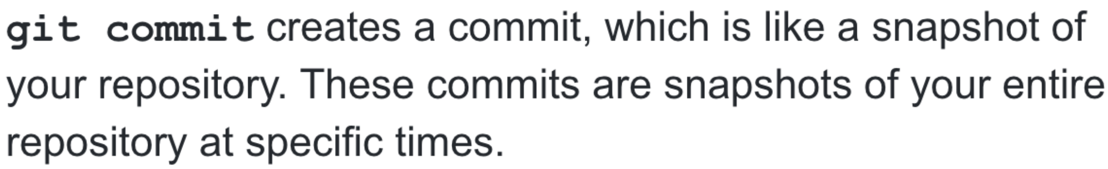
```
git add .
```

We always want to include all of our changes when we create a snapshot.

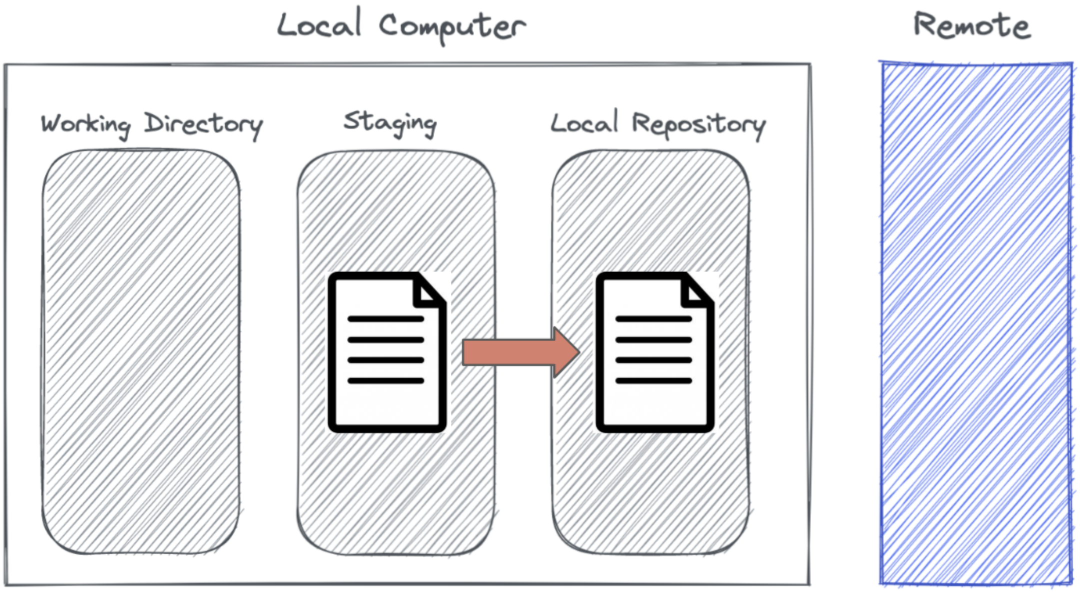
As shown in the picture, when we add files using **git add**, we place these files into the **staging area**. The staging area is what git will use when creating our snapshot.

2. Creating a snapshot - committing our changes

When we have added all of the changes we want to include in our snapshot, we are ready to use the **git commit** command.



`git commit` creates a commit, which is like a snapshot of your repository. These commits are snapshots of your entire repository at specific times.



As shown above, the **git commit** takes all files (including file deletions) stored in the staging area. It creates a snapshot of your project during a particular moment in the project's history - it creates a record of how all the files and directories appeared in a project at the time the commit is created. When we do this, we also include a message that will always be associated with this snapshot. How do we do this?

To create a commit (a snapshot):

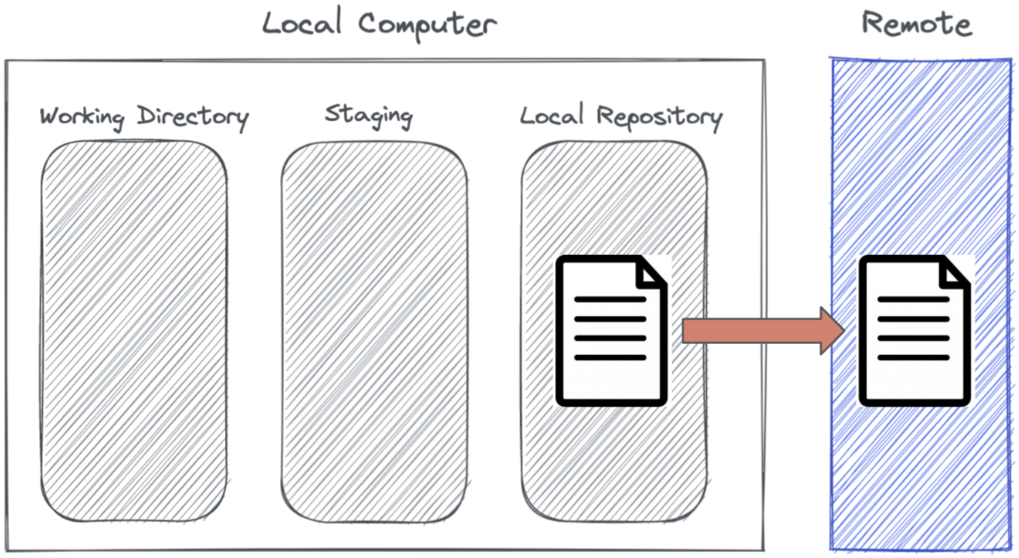
```
git commit -m "feat: commit message"
```

In this command, we specify the -m flag to indicate we want to add a commit message directly to our commit. Then, we enclose our commit message in quotation marks following the -m flag. In this case, our commit message is “feat: commit message”.

Step 3: save a snapshot(s) to our remote

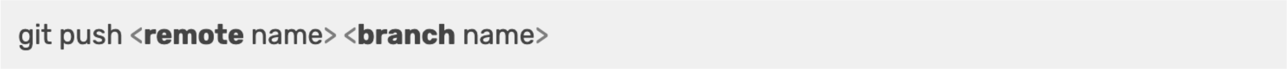
Once you have committed your code to the local repository, a record of your changes will be tracked. The **local repository** keeps track of your commits; what files were affected - basically everything that is different. It contains multiple commits - the entire history - actually - of every commit.

`git push` uploads all local branch commits to the remote branch.



When you are ready to push your snapshots - your commits - to the remote repository, you can use the **git push** command to update the remote repository. Remember, you can perform multiple commits to your local repository before pushing them to the remote repository; this is often the case.

Pushing allows you to send the commits you have made to the local version of a repository to a remote repository. For instance, you'll create a commit on your local machine if you are working on a team project. Once you're ready for everyone to see your code, you will push it to the remote repository so every collaborator can see it. To do this, we use the **git push** command as follows:

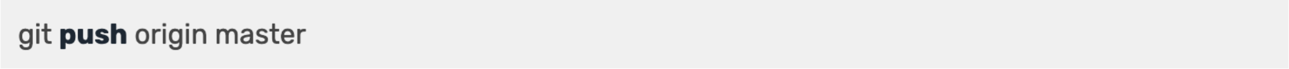


```
git push <remote name> <branch name>
```

Our "remote name" parameter refers to the remote git repository. If you have already configured git, this is most like set to "origin". This configuration is described in the *Working with Azure DevOps Repositories* document. If you want to commit to another repository, specify it using the "remote name" parameter.

The "branch" name parameter refers to the remote repository branch to which you want to push your changes.

Suppose we want to push the changes from a local repository to the "master" branch of a remote repository. We could do so using this command:



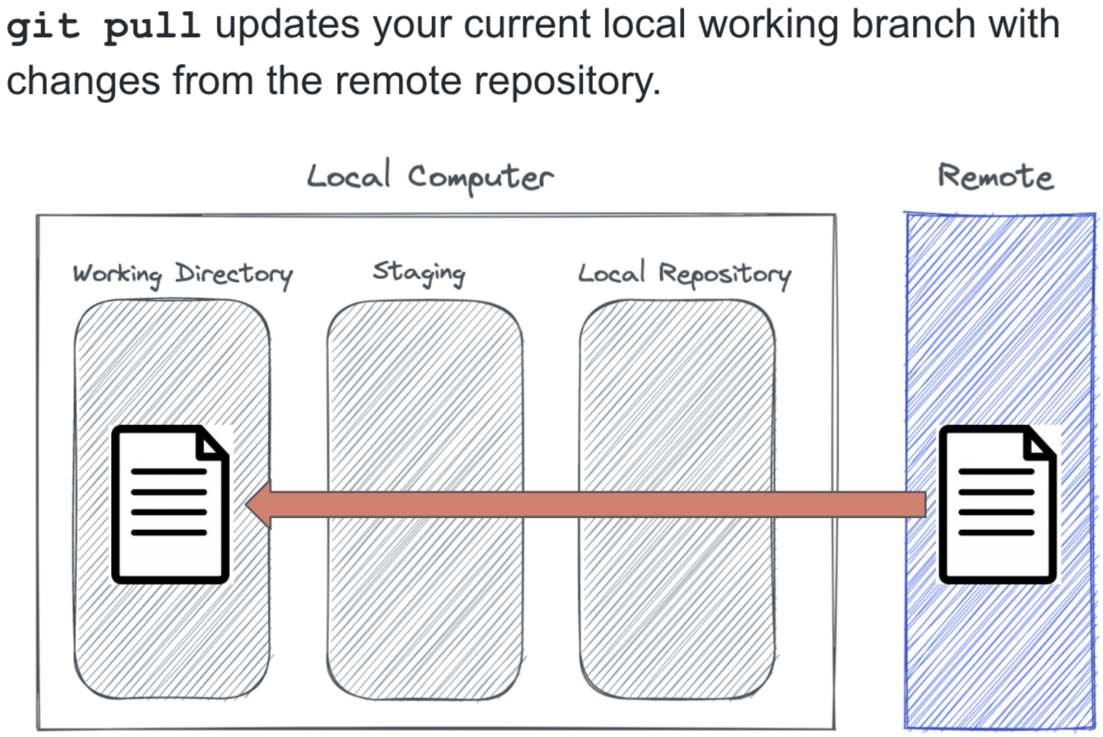
```
git push origin master
```

Updating your local repository

When you are collaborating with others, you must update your local working folder with their changes. Of course, other developers on your team need to update their working folders with the changes you have made. Using the **git pull** command is how we can do this.

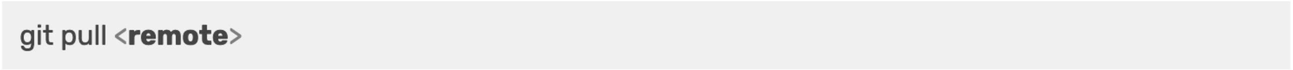
The **git pull** command is a helpful command that executes two other commands: **git fetch** and **git merge**. Let's break down how the git pull command works.

First, when you run **git pull**, the remote repository you are pulling will be downloaded. A copy of the code from the repository and the Git commits associated with the repo will be saved to your machine. Following this, a **git merge** operation is executed. This operation merges the code on your local machine with the newly-retrieved code, creating one final version of the codebase. This version will be equal to the one you have retrieved from a remote branch.



`git pull` updates your current local working branch with changes from the remote repository.

Let’s take a look at the **git pull** command:



```
git pull <remote>
```

The “remote” parameter refers to the remote repository you want to pull to your local machine. When you run this command, the remote repository will be retrieved and then merged into your local copy of the repository.

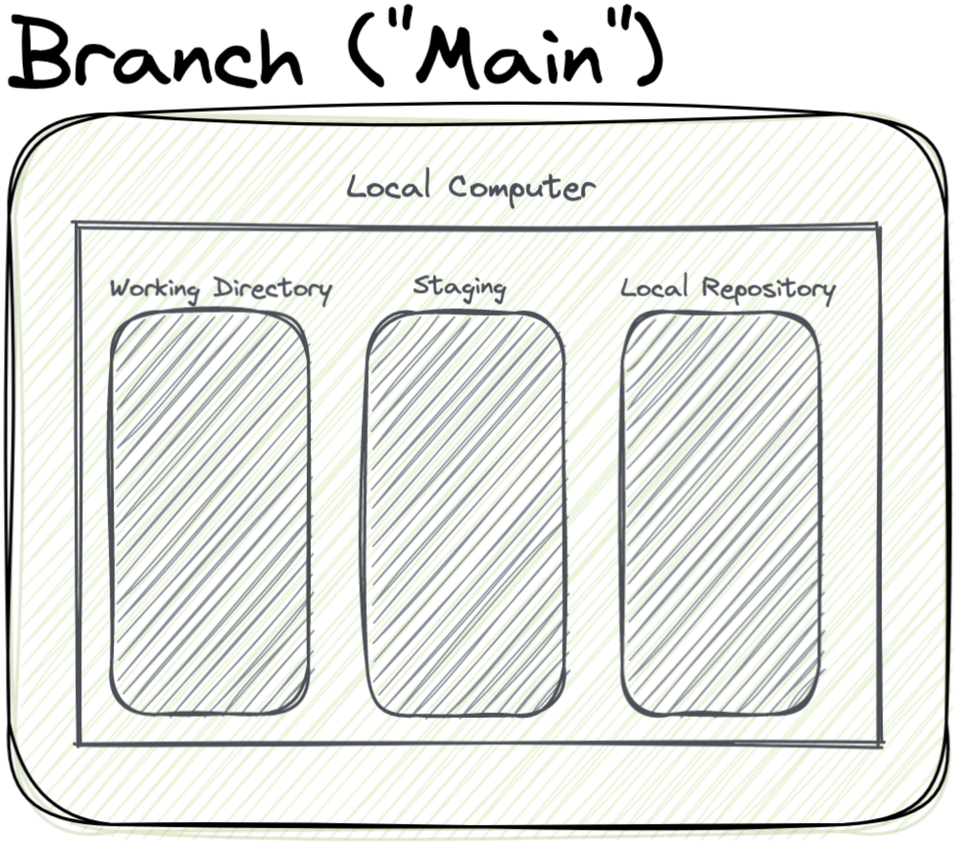
The git pull command does not affect untracked files. You will only receive changes that have been made to files on the remote branches that are being tracked by Git. These changes will be saved to your local working tree.

```
git pull origin
```

NOTE: It is an accepted practice that you pull the latest code from the remote before you push your code to the remote. This ensures that your changes are working with the latest code and that there are no compile or runtime errors.

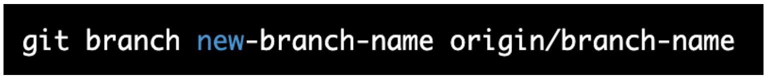
Branches

A “Branch” is an instance of your files - it includes the history of all the snapshots up to a particular date and time. Although it wasn’t explained until now, we actually have been working on a branch - it’s called ‘Main’ (or sometimes ‘Master’)



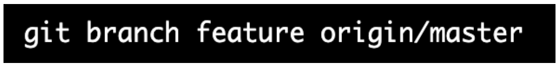
Creating a new Branch

To create a new branch based upon a specific remote (“origin”) branch, you can use the git branch command followed by the name of the new branch, like this:



```
git branch new-branch-name origin/branch-name
```

For example, to create a new branch called feature based upon the origin/master branch, you would run the following command:

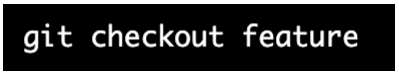


```
git branch feature origin/master
```

feature origin/master

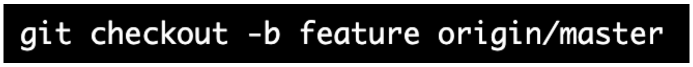
This will create a new branch called feature that is based upon the origin/master branch.

You can then switch to the new branch using the git checkout command:



```
git checkout feature
```

Alternatively, you can create and switch to the new branch in a single command by using the git checkout command with the -b flag:



```
git checkout -b feature origin/master
```

ot -b feature origin/master

This will create the feature branch and switch to it in one step.