**1-Git Branching**

**Introduction**

*“Branching means you diverge from the main line of development and continue to do work without messing with that main line.”*

This means a new copy of the main repository is made, and after it is split off of the original path, it is free to be modified, without making permanent changes to the original “main” branch. There are many different branching strategies that different projects utilize, as they each provide different benefits that may fit different workflows better than others.

To make a branch, we type:

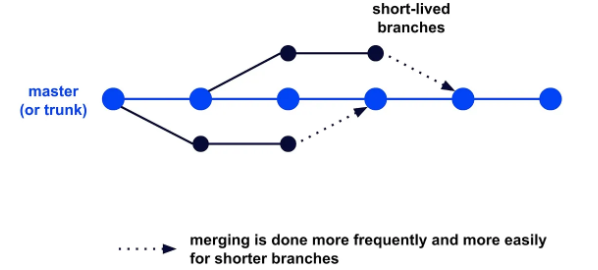
* *git branch <new-branch>*
* *git checkout <new-branch>*

This excerpt creates a new branch, and switches the path of your commits from the main branch to the newly made branch. After a change is ready to be published permanently to the main branch, we reconvene the branch onto the main, using a merge. A merge is executed by switching over to the branch you want to merge into, and then calling the merge command, like so:

* *git checkout main*
* *git merge <new-branch>*

# Trunk-Based Development

This type of development is the most straightforward method of Git collaboration. As the name implies, the development is done directly on the “trunk”, or main branch, with no other long-standing extraneous branches.

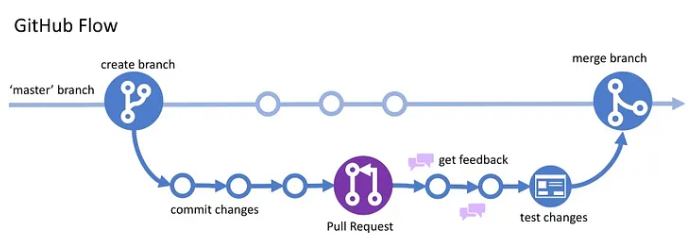


While this structure is very simple and easy to use, it requires a team that is very efficient and clean in their code. When changes are pushed directly onto the main branch, there is a much larger margin for small bugs to cause more dramatic impacts to the project than some of the more “secure” options. One way to mitigate this issue is to have very stringent testing and analysis before being deployed.

A staple of trunk-based development is a process called [**continuous deployment**](https://resources.github.com/ci-cd/). Continuous deployment is a method of automating the deployment of your application. When working with continuous deployment, after your push passes through automated test scripts, it is automatically deployed with the new change. A major benefit to this automated process is that it is efficient, stable, and allows for smaller changes more often, rather than waiting for one big change to go into effect.

**Feature Branching/Github Flow**

Feature branching is still a relatively simple strategy, but offers some more safeguards to keep the main branch safe until changes on a branch are approved through a [**pull request**](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests).

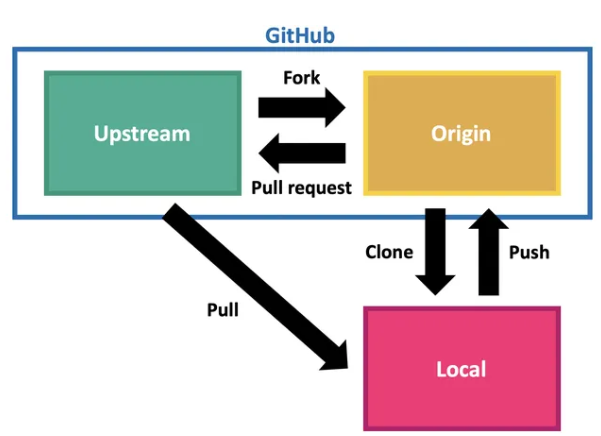


In feature branching, each new feature is made on its own branch, which is then subjected to approval and feedback through a pull request before being tested and merged back onto the main.

Typically, this method of development utilizes [**continuous delivery**](https://resources.github.com/ci-cd/)**,**which is very similar to continuous deployment, but the automation process ends before the project is pushed to production. This is a safeguard to prevent any possible bugs from slipping through the cracks in the tests. While this is a “safer” method, it also takes more time, and can lead to more delays in production compared to continuous deployment.

**Forking Strategy**

This method is very similar to the feature branching method, but instead of building out branches for each feature, individuals [**fork**](https://docs.github.com/en/get-started/quickstart/fork-a-repo) the main repository. With the forked copy, also called the origin, of the entire repository on their local environment, developers will make changes to that copy, and once the desired effect is created, send a pull request for their new copy to be merged back to the main branch.



This strategy is most common in [**open-source development**](https://www.redhat.com/en/topics/open-source/what-is-open-source), where any individual can access the code, and can offer advice or changes, without necessarily being a part of the main developers. The benefit of this method is that only the team of developers of the project are in control of the original repo, and are able to approve or decline any modifications that are proposed from outside sources. This allows for unlimited access and contributions from outside sources, yet keeps the main product completely safe from possible harm or detrimental changes.

**Create a Local Repository**

* First, open your terminal or command prompt and navigate to the directory where you want to create your project.
  + git init recipes
  + This command creates a new folder for the project called “recipes” and initializes a new Git repository in that folder. i.e. The init command tells us that it initialized an empty Git repository named  .git in the folder we told it to use.
  + The initialization process creates a  .git folder within the project folder that stores the files and data for the repository.
  + This file may be hidden by default. In Mac's Finder, use Command + Shift + . (period) to make hidden files appear. On Windows, use the File Explorer View tab and check Hidden items to display hidden files.
  + When a new repository is initialized, Git automatically creates a new primary branch. Depending on your Git settings, that primary branch is created with the name "main", "master", or something similar.
* Navigate down to the project folder:
  + #*cd recipes*
  + This working folder is the folder that Git watches to track our changes.
* verify the status of the Git repository:
  + #*git status*
  + The “***status***” command provides the status of our repository. At this point in time, we are on the main branch, we have **no** commits, and we have nothing to commit.

**Create Files**

* The primary purpose of a repository is to store a snapshot of the files from our working folder at a particular point in time. Use a text editor to create the two files: ***file1.txt*** and ***file2.txt***. Add one line of content to each file identifying the file.



// Each of these commands "echoes", or "writes" the text on the left into the file on the right. If the file doesn't exist, it creates the file.

* + Use a single greater than > with echo to replace the contents of the file with the specified text. Use a double greater than >> to append the text to the end of the file. We'll see the double greater than later in this tutorial.

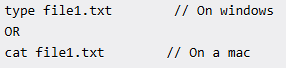
## Commit Files to the Local Repository

* Committing files to a local repository requires the following steps:
  + First, add the files to the Git staging area.
    - # git add .
    - Before committing files to the local repository, we tell Git which files we want to include in that commit by adding them to a staging area. Basically, staging allows us to selectively choose which changes to include in a commit.
    - Using the**.** (period) adds all of the files in the working folder (and its subfolders) to the staging area. If you only want to add specific files, you can list them instead.
    - Think of staging like a shopping cart. You can put items into your shopping cart or take items out of your shopping cart without committing to buying them. When you are ready, you commit to buying the items that are in your cart.
  + Now you're ready to commit the staged files to the local repository.
    - 
    - Here we commit the files in the Git staging area to the local repository.
    - The -m option is for a commit message. Follow the -m with the message, in quotes. Be sure to define a clear message describing the changes you are committing. More about commit messages in the next section below.
  + Then you can optionally verify the commit using the log command:
    - 
    - The log command outputs the commit history for the repository with the most recent commit first. Use it whenever you want to see what's happened within that repository.
    - NOTE: If your terminal window is too small to display the entire log, this command automatically opens a pager screen to page through the display. Use the PgUp, PgDn or arrow keys to page through the output. Press the Q key to exit the pager.
    - The add command doesn't output a message.
    - The commit command displays the branch we are on (main), the first few characters of the commit's id (more on that in a moment), and the commit message. It then lists the changed files. In this example, there were 2 files changed and both were insertions (new files).
    - The number after *create mode* indicates the file type and permissions. 100644 means that these are regular files, not folders, with owner read and write permissions.
    - The log command provides the commit history of the current branch. It includes a long string of numbers and letters called a Secure Hash Algorithm or SHA. The SHA is an id assigned to each commit. Technically, it's a checksum based on the files in the commit plus the log message, author information, and date. The log also displays the person that made the commit (name and email), the commit date, and the commit message.

**Create a Branch**

* Create a branch for every task or issue you work on. This ensures that your work on the task or issue is isolated from your existing codebase. That makes it easier to work on multiple parts of the project simultaneously, or try out ideas without adversely affecting your main codebase.
* We create a branch for the task following these steps.
  + First, switch to the main project branch:
    -  
    - helps to make sure we are on main project
  + Then create a branch from the current branch:
    - 
    - This branch command creates a branch with the specified name from the current branch. In this case, the branch is named style\_change.
    - But this command does *not* switch our working folder to the new branch.
  + Then switch to the new branch:
    - 
    - The checkout command checks out the branch so you can work with it. Think of it like checking a book out from the library so you can read it. Or selecting a movie from a streaming service so you can watch it.
    - As part of the checkout process, Git copies the files from the most recent commit on the specified branch into the working folder so we can work on them. When we insert, edit, or delete files in our working folder, we impact the checked out branch.
    - Alternatively, you can use a single command to create and switch to the new branch.
      * 
      * The -b option creates a new branch with the provided name and checks out that branch.
    - Optionally, view the list of all branches to confirm the creation of the branch:
      * 
      * 
      * The branch command lists all existing branches in the repository. And the current branch you are on is colored with green and marked with astrix(\*)
    - Note
      * The first checkout command tells us we are already on the main branch.
      * The second checkout command creates a new branch called style\_change and switches to that branch.
      * As part of the checkout process, the files in the working folder are changed to the files from latest style\_change branch commit. Since this is a new branch created from the main branch, the latest commit on the new branch points to the latest commit on the branch it was created from, which is main. So the files in the working folder are the files from the latest commit to the main branch.
      * The branch command lists the two branches. The current branch is indicated with an asterisk (\*).
  + Now that we have the new branch checked out, we can make changes to our working folder and those changes only impact that branch. Our main codebase remains unchanged.
    - To simulate a change to our branch, let's add a line of text to one of our files:



* + - Confirm the change by typing out the content of the file:
    - 

**Commit Changes to a Branch**

* First, stage the changes to include in the commit:



* Then commit the files that are in staging:



Or, if you've only modified or deleted files, not created any new files, you can stage and commit changes to the branch using a single command:



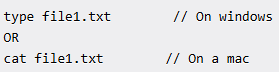
The -a option stages all modified and deleted files in the working folder, but does ***not*** include any new files. This commit command then creates a new snapshot of our working folder (except for any new files) and commits them.

* Optionally verify the commit using the log command:

 // The log command lists both of our commits with the most recent first.

**Create and Commit on Another Branch**

* Let’s create new branch to work on for login feature without affecting the original code base(main)
  + Start by switching to the main branch.
    - 
    - This checks out the main branch, replacing the code in our working folder with the code from the latest commit on our main branch.
  + Optionally, confirm the code in the working folder:

 //reads the content to the terminal

* + - Typing out our file1.txt file, we see that we don't have our style change. Our working folder now contains the original code from our main branch.
  + Next, create and switch to a new *login\_feature* branch:

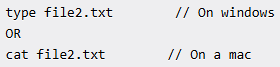


* + - The -b option signifies that we are creating a new branch with the provided name and checking out that branch.
  + Optionally confirm the new branch:



lists all the branches we created so far with the one we switched into marked with astrix(\*)

* + - I.e. the ***branch*** command lists our three branches. The asterisk (\*) indicates that we have the login\_feature branch checked out.
    - Since this is a new branch again created from the main branch, the latest commit on the new branch points to the latest commit on the branch it was created from, which is main. So the files in the working folder are the files from the latest commit to the main branch.
  + Make modifications to the current “login\_feature” branch
    - 
    - Confirm the change by typing out the content of the file:



* + To finish our changes, we commit those changes.
    - Stage the changes for the commit:



* + - Then commit the files from staging to the checked out branch:

// This creates a new commit from our staging area.

* + - Optionally verify the commit using the log command:



* + Notice in the logging above that we don't see the style change commit. That's because it's in the style\_change branch and not in this login\_issue branch.
  + While working on the login issue, if you need to go back to the style change task:
    - Check in all current changes to the login issue.
    - Use the checkout command to switch to the style\_change branch.
    - Work on the task, checking in all changes.
    - When you are ready, switch back to the login\_issue branch.
    - Use the git branch command at any time to view your list of branches and confirm which branch you have checked out.

## Merge a Branch

* When we are finished with the task or issue, we want to merge the code for that task or issue into our main codebase. We can then optionally deploy the change to the users.
* to merge our login\_issue branch into our main branch.
  + First, switch to the main branch:
    - 
  + Then merge the task or issue branch into the main branch.
    - 
    - This merges the changes from the specified branch into the current branch. If there are any conflicts, Git prompts you to resolve them before the merge can be completed.
    - The merge command displays information on the type of merge along with the files that were merged.
    - In this case, Git used a fast-forward type of merge. This is a simple merge that Git can use when there are no new changes on the current branch since the merged branch was created.
    - The commits from the merged branch are effectively incorporated into the current branch. No need for a new merge commit and there is no possibility of a merge conflict.

## Delete a Branch

* Delete any branch that you no longer need to keep your repository clean and easier to manage.
  + Let's start by confirming our current list of branches:



* + If you aren't on the main branch, switch to the main branch:



You cannot delete the branch you are on. So by switching to the main branch you can delete any of the other branches.

* + Delete the desired task or issue branch:
    - 
    - The -d option deletes the specified branch from the local repository.
    - # git branch lists remaining branches

## Merge "main" Back to Branches

* To minimize more extensive merging conflicts later, it's often a good idea to keep all task and issue branches up to date with main. That means after changes from a branch are merged into the main branch, we want to merge those changes from the main branch back to all remaining task or issue branches.
  + First, switch to one of the task or issue branches:
    - 
    - Here we switch to the style\_change branch as it's our only other branch.
    - Optionally, view the commits for that branch: # git log
  + Now merge the main branch into the current branch:
    - 
    - This merge command merges the change history from the main branch into the current branch, which in this case is the style\_change branch.
    - Optionally, view the commits for that branch: # git log
    - Notice that in this case, Git couldn't use the fast-forward type of merge. That's because the style\_change and main branches have diverged, meaning that they have both changed since the time we created the style\_change branch. Git instead used an ort strategy for merging the changes
    - Because Git couldn't use the fast-forward type of merge, the merge process created a new commit, called a *merge commit*. A merge commit represents a combination of the changes made on the separate branches.

## Finish Our Project

* Switch to the style\_change branch:



* Add the files to staging:



* And commit the changes:



* Switch to the main branch:



* Merge the style\_change branch into main:



* Delete the style\_change branch:
  + 
* Optionally, confirm that the branch was removed:
  + 