The use of software such as Git, Docker, automated testing, and continuous integration are key for the success of any company. Especially if they’re a tech company. This is apparent when you look at the productivity of any company that uses these software's. They complement teamwork and efficiency, without it being necessary to be “together”. This is key especially in this current COVID-19 environment. Git and Docker makes it possible for any number of people to work together remotely without distractions. Git is seamless in it’s approach to teamwork and productivity. It allows you to see your teammates work and branch off to create your own work or even edit their work. As shown below, due to Git’s many features and functions, teams are allowed to work together seamlessly.

**Repository:**

A Repository is the location where files are stored on Github. If a company wanted to collaborate on a project simultaneously, they could all connect to the Github repository and download it onto their own machines. This would allow them to access the main repository on their machine while being able to make changes and push them to the main repository.

**Clone:**

A clone is a duplicate of the repository so that edits can be made without interfering with the main repository. Although changes can be made without pushing changes to the main repository, using ***git clone*** will be able to show multiple changes that people make and compare how the repository would act without major changes on the main.

**Fork**

***Git fork*** is similar to git clone, but it has one major distinction. While git clone copies the repository onto the local machine, git fork makes a duplicate onto the repository on Github. This allows changes to be made to the forked repository without having to push the changes first, and the changes done can be merged with the original repository or whatever designated location.

**Branch**

A git branch is an alteration to the path you are currently working on which is not set to be integrated with that path. If someone wanted to make a change on a project but wanted it to be worked on separate from the main path, a branch could be made and the person could continue working on it and making commits while other branches were worked on at the same time. When they wanted to commit it to a different path, they could merge the two and combine what they have done.

**Commit**

***Git commit***is publishing the changes you made into the project repository. This can be done on remote repositories or on the Github repository, but if it is done on the remote repository the changes must be pushed before they would show up on Github.

**Merge**

**A Git Merge** is the act of combining the contents of one branch integrating it with the contents of a target branch. This is ideal for making two independent separate branches into one single branch, combining multiple commits into one unified branch. A merge is done after two people are done working separately and want to combine their branches.

**Checkout**

**Git Checkout** is the command used to update the files in the branch and tell Git to store all the new commits to the new branch. For example, when one is done adding a commit to a branch, they would enter git checkout in order to officially make it part of the branch.

**Push**

**A Git Push** is the command used to upload items from the local repository into a remote repository. Git Push is essential for remote teamwork. It allows for work on separate repositories to match each other. Git push will be used by anyone who would like to edit someone else’s repository, or help them with something.

**Pull**

**A Git Pull** is the opposite command of a Git Push. Instead of uploading from a local repository to a remote repository, Git Pull downloads content from a remote repository and updates the local repository and matches the content with each other. A Git Pull is essential for any team, without it, all remote work would not be added to repositories.

**Status**

**A Git Status** is the command used to understand what has happened in the repository. It’s used to know what changes were made, what was added and removed, etc. It checks the state of each command.

**Master Branch**

A master branch in Git is the main branch as to which every commit is started from. It is the default branch given when starting a commit. All subsequent branches are made from the master.

**How to set up Git and work collaboratively with more than one person:**

**Step 1. Create a Repository**

A repository is usually used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets – anything your project needs. It is recommended to include a README, or a file with information about your project. GitHub makes it easy to add one at the same time you create your new repository. It also offers other common options such as a license file.

For example, “IS601\_Project\_1” is the repository we created for mini project. This repository can be a place to store ideas, resources, or even share and discuss things with others.

**To create a new repository**

In the upper right corner, next to your avatar or identicon, click and then select New repository.

Name the repository “IS601\_Project\_1”.

Write a short description.

Select Initialize this repository with a README.

Click Create repository.

**Step 2. Create a Branch**

Branching is the way to work on different versions of a repository at one time.

By default, repository has one branch named “main” which is the definitive branch. We use branches to experiment and make edits before committing them to “main” branch.

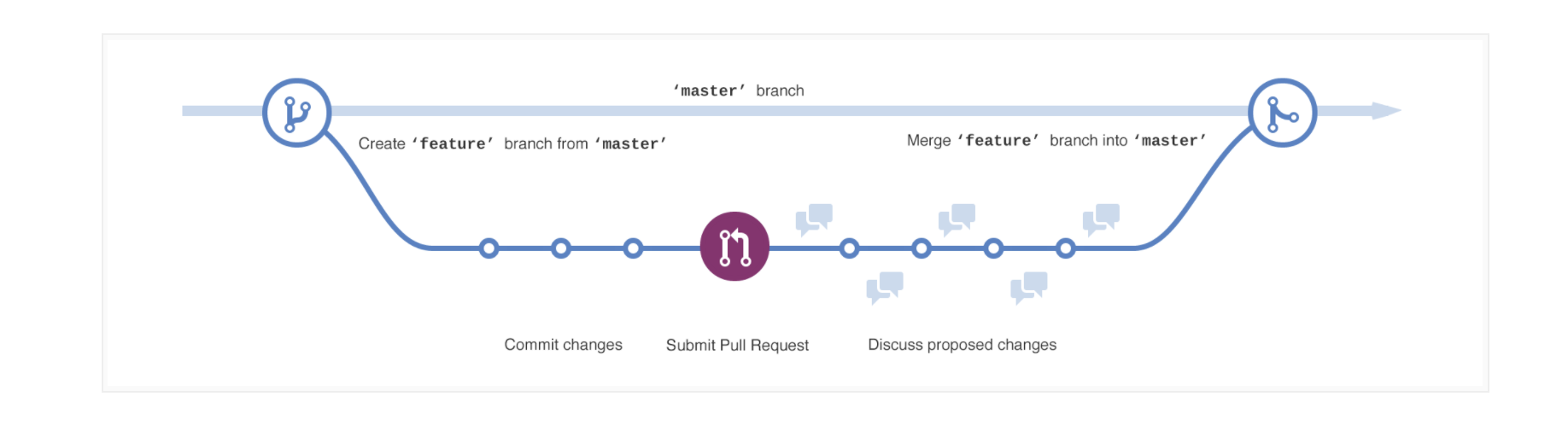
When you create a branch off the “main” branch, we are making a copy, or snapshot, of main as it was at that point in time. If someone else made changes to the “main” branch while you were working on your branch, you could pull in those updates.

Below Diagram shows:

The main branch

A new branch called “feature” (because we are doing ‘feature work’ on this branch)

The journey that “feature” takes before it’s merged into main



We can save different versions of a file. Something like:

Basic\_Git\_Commands.doc

Basic\_Git\_Commands edited.doc

Updated Basic\_Git\_Commands edited.doc

Branches accomplish similar goals in GitHub repositories.

In GitHub, developers, writers, and designers can use branches for keeping bug fixes and feature work separate from main (production) branch. When a change is ready, we can merge the branch into main.

**To create a new branch:**

1. Go to your new repository “IS601\_Project\_1”.
2. Click the drop down at the top of the file list that says **branch: main**.
3. Type a branch name, “feature” into the new branch text box.
4. Select the blue **Create branch** box or hit “Enter” on your keyboard.

Now we have two branches, main and feature. They look exactly the same, but not for long! Next, we will add our changes to the new branch.

**Step 3. Make and commit changes**

Now, we are on the code view for “feature” branch, which is a copy of main and we can make some edits.

On GitHub, saved changes are called commits. Each commit has an associated commit message, which is a description explaining why a particular change was made. Commit messages capture the history of your changes, so other contributors can understand what you have done and why.

**Make and commit changes**

Click the README.md file.

Click the pencil icon in the upper right corner of the file view to edit.

In the editor, write a bit about your project and about teammates.

Write a commit message that describes your changes.

Click Commit changes button.

These changes will be made to just the README file on your feature branch, so now this branch contains content that is different from main.

**Step 4. Open a Pull Request**

Now that we have changes in a branch off of main, we can open a pull request.

Pull Requests are the heart of **collaboration** on GitHub. When you open a pull request, you are proposing your changes and requesting that someone review and pull in your contribution and merge them into their branch. Pull requests show diffs, or differences, of the content from both branches. The changes, additions, and subtractions are shown in green and red.

As soon as you make a commit, you can open a pull request and start a discussion, even before the code is finished.

By using GitHub’s @mention system in your pull request message, you can ask for feedback from specific people or teams, whether they are down the hall or 10 time zones away.

You can even open pull requests in your own repository and merge them yourself. It is a great way to learn the GitHub flow before working on larger projects.

Open a Pull Request for changes to the README:

1. Click the Pull Request tab, then from the Pull Request page, click the green New pull request button.
2. select the branch you made, feature, to compare with main (the original).
3. Look over your changes in the diffs on the Compare page, make sure they are what you want to submit.
4. When you are satisfied that these are the changes you want to submit, click the big green Create Pull Request button.
5. Give your pull request a title and write a brief description of your changes.
6. When you are done with your message, click Create pull request.

**Step 5. Merge your Pull Request**

In this final step, it is time to bring your changes together – merging your “feature” branch into the “main” branch.

Click the green Merge pull request button to merge the changes into main.

Click Confirm merge.

Go ahead and delete the branch, since its changes have been incorporated, with the Delete branch button in the purple box.

**How to create a merge conflict and resolve the merge conflict:**

You can resolve merge conflicts using the command line and a text editor. Merge conflicts occur when competing changes are made to the same line of a file, or when one person edits a file and another person deletes the same file.

**Competing line change merge conflicts**

To resolve a merge conflict caused by competing line changes, you must choose which changes to incorporate from the different branches in a new commit.

For example, if you and another person both edited the file styleguide.md on the same lines in different branches of the same Git repository, you will get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

1. Open Git Bash.
2. Navigate into the local Git repository that has the merge conflict.

cd *REPOSITORY-NAME*

1. Generate a list of the files affected by the merge conflict. In this example, the file styleguide.md has a merge conflict.

$ git status

> # On branch branch-b

> # You have unmerged paths.

> # (fix conflicts and run "git commit")

> #

> # Unmerged paths:

> # (use "git add ..." to mark resolution)

> #

> # both modified: styleguide.md

> #

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

1. Open your favorite text editor, such as Atom, and navigate to the file that has merge conflicts.
2. To see the beginning of the merge conflict in your file, search the file for the conflict marker. When you open the file in your text editor, you will see the changes from the HEAD or base branch after the line <<<<<<< HEAD. Next, you will see =======, which divides your changes from the changes in the other branch, followed by >>>>>>> BRANCH-NAME. In this example, one person wrote "open an issue" in the base or HEAD branch and another person wrote "ask your question in IRC" in the compare branch or branch-a.

If you have questions, please

<<<<<<< HEAD

open an issue

=======

ask your question in IRC.

>>>>>>> branch-a

1. Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand-new change, which may incorporate changes from both branches. Delete the conflict markers <<<<<<<, =======, >>>>>>> and make the changes you want in the final merge. In this example, both changes are incorporated into the final merge:

If you have questions, please open an issue or ask in our IRC channel if it's more urgent.

1. Add or stage your changes.

$ git add

1. Commit your changes with a comment.
2. Commit your changes with a comment.

$ git commit -m "Resolved merge conflict by incorporating both suggestions."

You can now merge the branches on the command line or push your changes to your remote repository on GitHub and merge your changes in a pull request.

**Resolve file merge conflicts**

To resolve a merge conflict caused by competing changes to a file, where a person deletes a file in one branch and another person edits the same file, you must choose whether to delete or keep the removed file in a new commit.

For example, if you edited a file, such as README.md, and another person removed the same file in another branch in the same Git repository, you will get a merge conflict error when you try to merge these branches. You must resolve this merge conflict with a new commit before you can merge these branches.

1. Open Git Bash.
2. Navigate into the local Git repository that has the merge conflict.

cd REPOSITORY-NAME

1. Generate a list of the files affected by the merge conflict. In this example, the file README.md has a merge conflict.

$ git status

> # On branch main

> # Your branch and 'origin/main' have diverged,

> # and have 1 and 2 different commits each, respectively.

> # (use "git pull" to merge the remote branch into yours)

> # You have unmerged paths.

> # (fix conflicts and run "git commit")

> #

> # Unmerged paths:

> # (use "git add/rm ..." as appropriate to mark resolution)

> #

> # deleted by us: README.md

> #

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

1. Open your favorite text editor, such as Atom, and navigate to the file that has merge conflicts.
2. Decide if you want to keep the removed file. You may want to view the latest changes made to the removed file in your text editor.

To add the removed file back to your repository:

$ git add README.md

To remove this file from your repository:

$ git rm README.md

> README.md: needs merge

> rm 'README.md'

1. Commit your changes with a comment.

$ git commit -m "Resolved merge conflict by keeping README.md file."

> [branch-d 6f89e49] Merge branch 'branch-c' into branch-d

You can now merge the branches on the command line or push your changes to your remote repository on GitHub and merge your changes in a pull request.

1. **Forking vs Cloning**

The difference between cloning and forking a repository on GitHub:

**Forking**

A fork is a copy of a repository that allows you to freely experiment with changes without affecting the original project. A forked repository differs from a clone in that a connection exists between your fork and the original repository itself. In this way, your fork acts as a bridge between the original repository and your personal copy where you can contribute back to the original project using Pull Requests.

Forking a project is as easy as clicking the Fork button in the header of a repository. Once the process is complete, you will be taken right to the forked copy of the project so you can start collaborating.

**Cloning**

When you create a new repository on GitHub, it exists as a remote location where your project is stored. You can clone your repository to create a local copy on your computer so that you can sync between both the local and remote locations of the project.

Unlike forking, you will not be able to pull down changes from the original repository you cloned from, and if the project is owned by someone else you won’t be able to contribute back to it unless you are specifically invited as a collaborator. Cloning is ideal for instances when you need a way to quickly get your own copy of a repository where you may not be contributing to the original project.

To clone a repository, head over to the main page of a project and click the Clone or download button to get the repository’s HTTPS or SSH URL. Then, you can perform the clone using the git clone command in your command line interface of choice.