

COMP 2800 - Projects 2

Git Workflow Assignment

Introduction:

The goal of this document is to serve as a walkthrough using Git with Git workflows for version control and group collaboration. It will also help you become familiar with branching, merging, pulling from and pushing to a central GitHub repository, pull-requests and other Git functionality.

Up to this point if you have probably just used Git to keep track of you own individual files and making commits to the default branch called **master**.

This is the simplest Git workflow called The Basic Git Workflow.



Figure 1: The Basic Git Workflow - all commits are made to the one and only master branch.
(Each circle represents its own Git commit)

This was fine for individual projects, and was very useful for tracking your own changes, but for larger group projects we may need something new - branches. Here's why.

When you are working by yourself, you often work on a few small things at once and finish them before completing the next task or feature. When working in a group, multiple people are going to be working on different features at once. If a feature is still being developed or tested, you may not want to combine it with the rest of your stable and fully tested code base. If you commit your buggy untested code to **master** and it breaks everyone else's code, no one will be able to continue developing until you fix your buggy code! Everyone will hate you for this. Don't do it! Fortunately, you can avoid this, by using branches.

Git branches separate code changes or features into their own version of the code so that untested (and perhaps buggy) code can exist outside of the well-tested stable code base. There are a number of ways to use branches to organize your Git repository (repo) and they are called Git workflows.

The 2 most common Git workflows are (other than The Basic Git Workflow):

1. Feature-Branch Workflow
2. GitFlow Workflow

The Feature-Branch Workflow involves creating separate and distinct branches for each feature in progress. Commits are only made to a feature branch that the code changes are related to.

Important points of the Feature-Branch Workflow:

- Create a separate branch for each group of related code (called feature branches).
- Commits are made to feature branches.
- Commits are never made directly to **master**.
- Feature branches, once completed, tested and stable, can be *merged* back into **master**.
- The master branch contains only stable and well tested code.

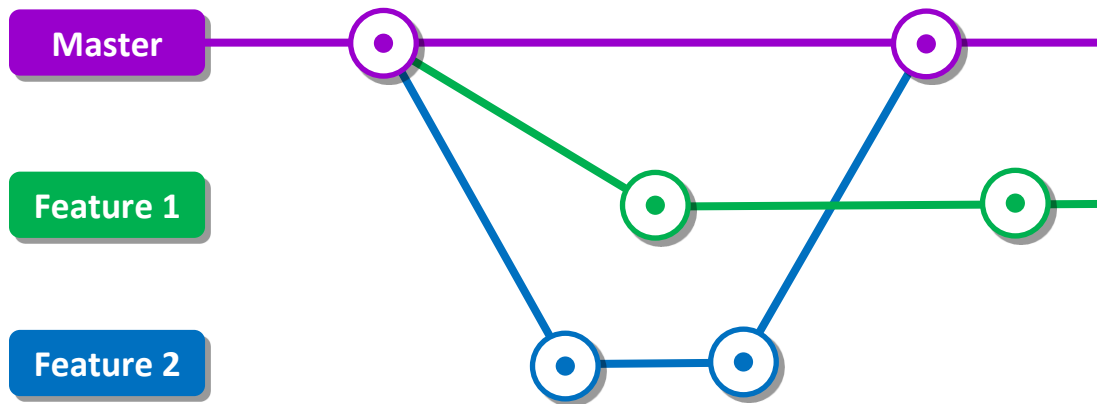


Figure 2: The Feature-Branch Git Workflow - all commits are made to separate feature branches.

Once tested and all feature code is stable, feature branches can be merged back to master.

The blue line (Feature 2) shows a feature branch that is fully tested and merged back to master (purple line).

The green line (Feature 1) shows a feature is still in progress and has not been merged back to master.

Only once a feature branch is fully tested and stable it can be merged back to master.

The GitFlow Workflow is based on the Feature-Branch Workflow, but adds one important ability. The one limitation of the Feature-Branch Workflow is that it doesn't easily allow for testing the *combination* of features. Many times you want to combine features and test them as a set - also known as integration testing. If you merge each feature to the master branch before integration testing them, you may find your code is unstable on master because the features don't co-exist well.

The GitFlow Workflow adds one more branch to the Feature-Branch Workflow often called **dev** or **testing**. This is where features are merged to, for integration testing. Once a group of features are well tested and stable, these features can be safely merged back to **master**. This way **master** always has a stable code base.

Important points of the GitFlow Workflow:

GitFlow Workflow has the same features as the Feature-Branch Workflow:

- Create a separate branch for each group of related code (called feature branches).
- Commits are made to feature branches.
- Commits are never made directly to **master**.

With the following changes:

- Feature branches, once completed, unit tested and stable, can be *merged* into **dev**.
- When *all* of the features currently on the **dev** branch are integration tested and stable, the features can be merged back to **master**.
- The master branch contains only stable and well *integration* tested code.

COMP 2800 Projects 2
Git Workflow Assignment

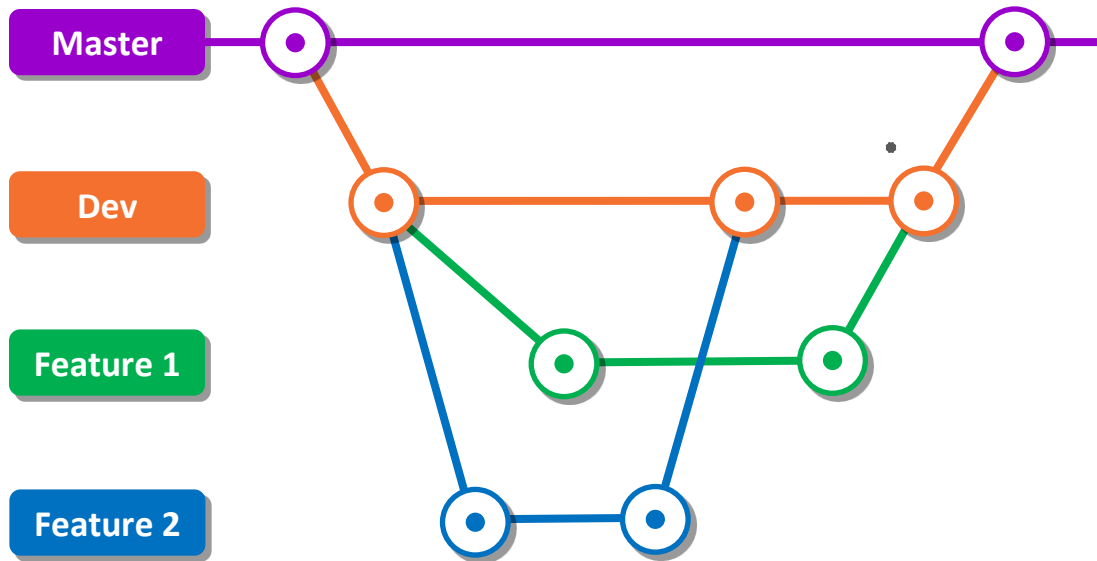


Figure 3: The GitFlow Workflow - all commits are made to separate feature branches. Once tested and all feature code is stable, feature branches can be merged into dev for integration testing. The blue line and green lines (Features 1 and 2) show feature branches that are individually unit tested and merged to dev (orange line). Once both features are committed to dev, they can be integration tested (the commit marked with a *). Only if dev is fully integration tested and stable can it be merged back to master.

REMEMBER: The goal of Git workflows is to always make sure the code in **master** is tested and stable so that all of your teammates can get the latest version of the code and continue to code without having to fix someone else's buggy, untested features.

Use Case Example:

A team of 4 developers are working together to build a website. At the moment, they only have a single page `index.html` with a very simple CSS file `style.css`.

Here are some of the tasks (features) that they are working on:

- Alvin:
 - Create an About Us Page
 - Create a Contact Page
- Mia:
 - Create a mobile friendly "hamburger" menu.
- Peng:
 - Create a "card-swipe" page functionality for the main page.
- Seeta:
 - Create a database structure to store persistent user login data.
 - Create a Login Page.

Let's Say Mia has found some sample code online for how to create a "hamburger" menu. She wants to commit her code to the repo, so she doesn't lose what she has found, but she hasn't fully tested the code yet and she knows it will likely break a few things on the site because it's still in progress. In the Basic Git Workflow (with only one master branch), if Mia committed her code to master, other developers like Peng, when they pull her changes from master, would get her half-implemented and buggy code. Peng can't properly test the CSS and JS changes he needs for his "card-swipe" pages when there are bugs in Mia's "hamburger" menu. Alvin and Seeta could also be negatively affected if a bug in the "hamburger" code prevents the site from displaying properly.

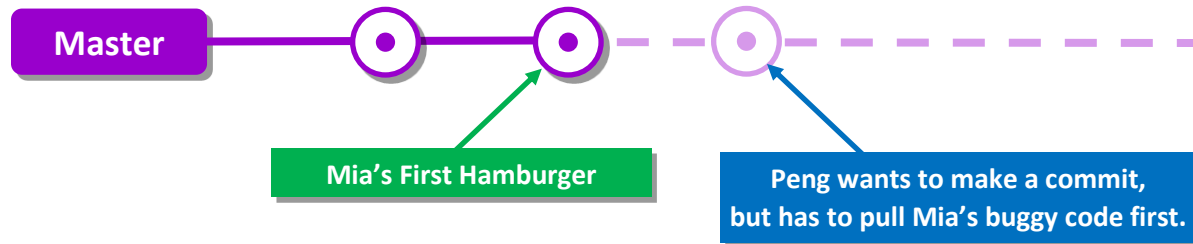


Figure 4: The Basic Git Workflow Example - Mia's buggy "Hamburger" commit blocks Peng from committing his code unless he pulls down the changes first. However, he doesn't want to pull those changes because they are untested and will break his code.

The Feature-Branch Workflow says that Mia should create a branch for her "hamburger" menu feature. All of her changes are made to commits on this branch only, until she has fully implemented and tested her "hamburger" menu on the site. Peng will not see Mia's changes because he is working his own branch which is separate from Mia's.

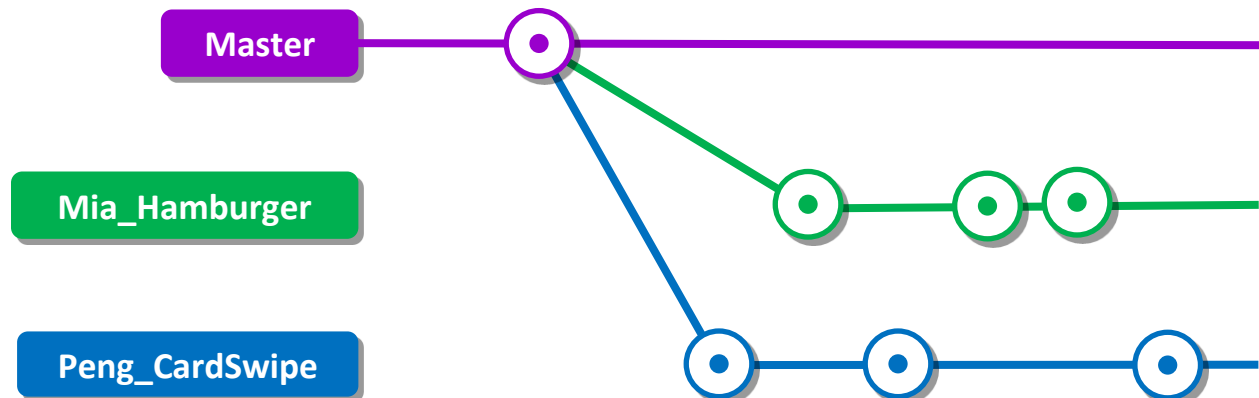


Figure 5: The Feature-Branch Git Workflow Example - Mia's "Hamburger" commits are isolated in her own feature branch call 'Mia_Hamburger'. Peng's changes are also isolated in a feature branch called 'Peng_CardSwipe'. Now their untested code won't break each other's code.

COMP 2800 Projects 2
Git Workflow Assignment

It looks like Mia has completed and unit tested her "hamburger" menu feature. Peng has completed and tested his "card-swipe" feature, too. In the Feature-Branch Workflow they both commit their tested (and what they *thought* was stable code) to **master**. What they didn't anticipate was, their *combined* changes to the CSS and JS have broken the entire site! Now their commit to master has spread to Alvin and Seeta's code and no one can continue their work until this new bug is resolved!

To solve this problem, the team decides to implement the GitFlow Workflow. The website development team adds one more branch called **dev**. And this time, instead of committing their code to **master** directly, they commit their code to **dev**. This gives both Mia and Peng a chance to integrate both their features together and do some integration testing before it gets committed to **master**. Mia and Peng, find a bug in the CSS that needs fixing, they create a commit on **dev** that fixes the bug and finish their integration testing. They merge their fully tested code from **dev** to **master**. Alvin and Seeta can safely pull from **master** without concern of bringing in untested code.

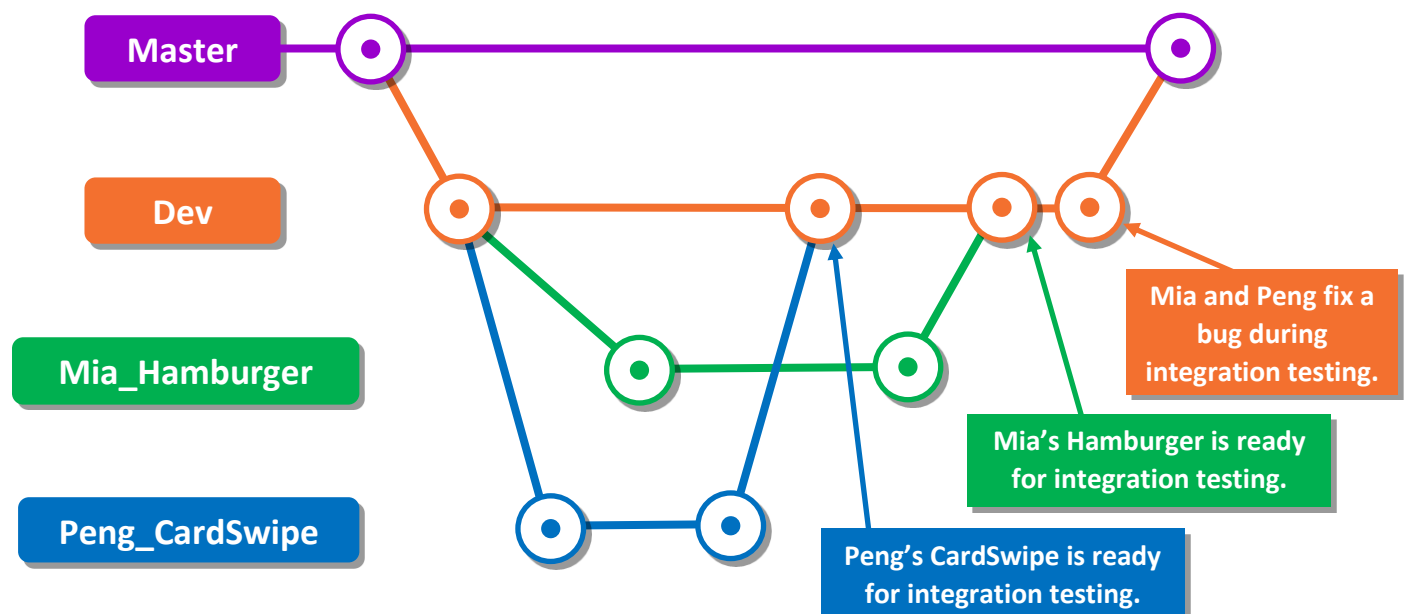


Figure 6: The GitFlow Workflow Example - Mia's "Hamburger" branch and Peng's "Card-Swipe" branch are merged into the dev branch for integration testing. During integration testing, Mia and Peng find a bug and commit the fix to the dev branch. After integration testing is complete the tested and stable code is merged to master.

As part of this assignment you will create a simple GitHub repo, collaborate with your team to create some feature branches, commit your individual changes and follow specific Git workflows. Although you will need to work with your team as Git collaborators, you will be submitting screenshots of your *individual* workflow GitHub network graph.

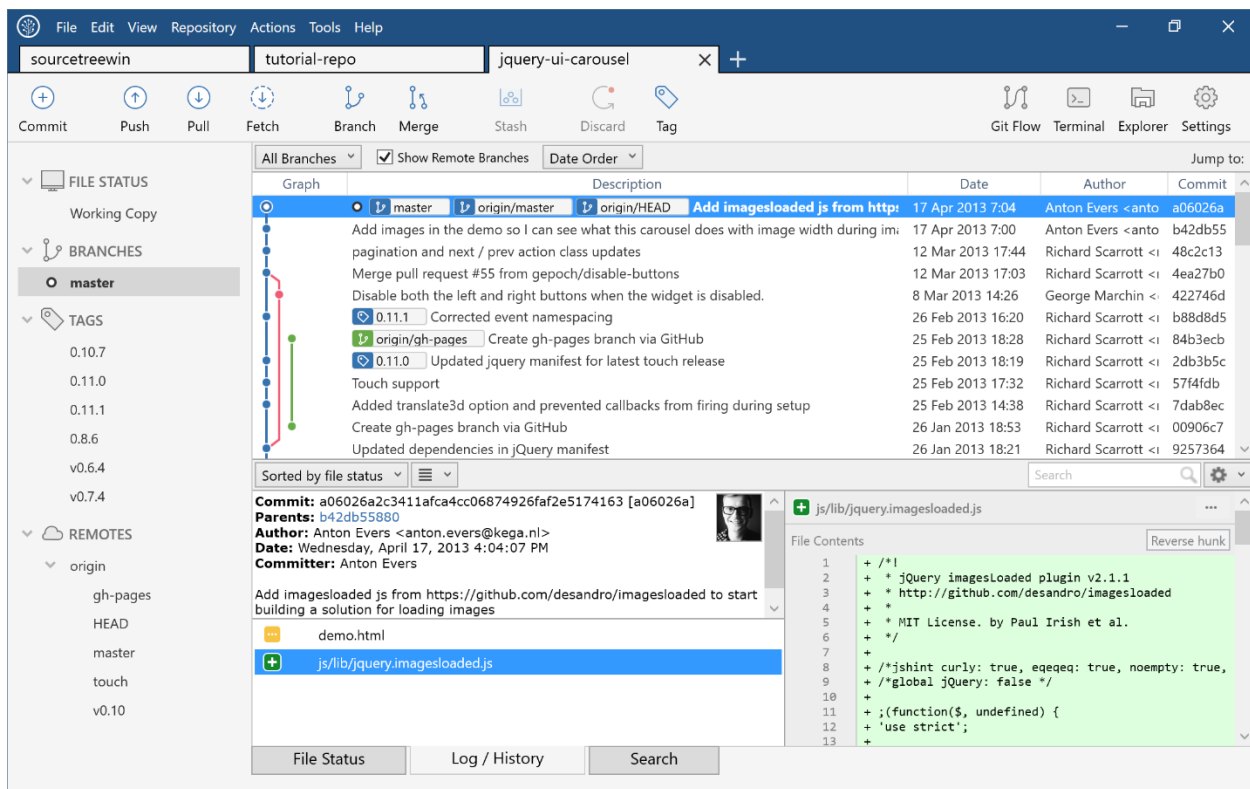
COMP 2800 Projects 2 Git Workflow Assignment

Many of the screenshots and instructions will reference a free Git GUI Software by Atlassian called Sourcetree. Sourcetree is cross-platform and works on Window and Mac. If, however, you are more comfortable with using Git on the command line, you are welcome to do so. You will also need a GitHub account (<http://github.com>) for you and each one of your teammates.

Sourcetree provides an intuitive interface for all of the common Git commands and provides visual feedback on the current status of the Git repo by showing all your uncommitted changes, which branch you are on and a network tree view showing the repo's commit history.



Sourcetree can be downloaded from: <https://www.sourcetreeapp.com/>



Source: https://blog.sourcetreeapp.com/files/2017/01/win_2_header.png

Some Git Workflow Websites and Git Resources:

Comparing Git Workflows:

<https://www.atlassian.com/git/tutorials/comparing-workflows>

Git workflows:

<https://buddy.works/blog/5-types-of-git-workflows>

Git Commands overview:

<https://rogerdudler.github.io/git-guide/>

Firebase Realtime Database Documentation:

<https://firebase.google.com/docs/database/web/read-and-write>

Git Master Guide:

<https://git-scm.com/book/en/v2>

GitFlow Branching Model:

<https://nvie.com/posts/a-successful-git-branching-model/>

GitHub Pull Requests:

<https://guides.github.com/activities/hello-world/>

Steps for Creating a Feature Branch Workflow:

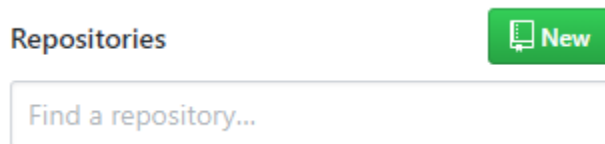
Using the Feature Branch Workflow, create a GitHub Repo with your team, create commits on feature branches and merge your feature branches back to **master**.

Step 1:

Create a GitHub Repo.

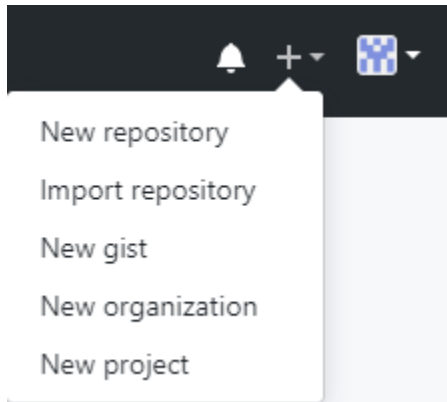
Pick one team member to be the owner of the repo. This GitHub Owner will create a new repo on GitHub (github.com).

You can either click the green 'new' button next to repositories:



COMP 2800 Projects 2 Git Workflow Assignment

Or click the '+' icon next to the account info menu:

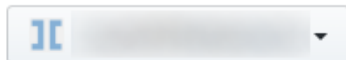


Give your repo the name: `COMP_2800_Feature_Branch_Workflow`

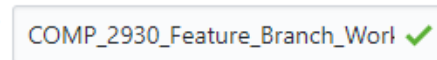
Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner



Repository name *



Great repository names are short and memorable. Need inspiration? How about [bookish-octo-doodle?](#)

Description (optional)



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.



Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▼

Add a license: **None** ▼



Create repository

COMP 2800 Projects 2

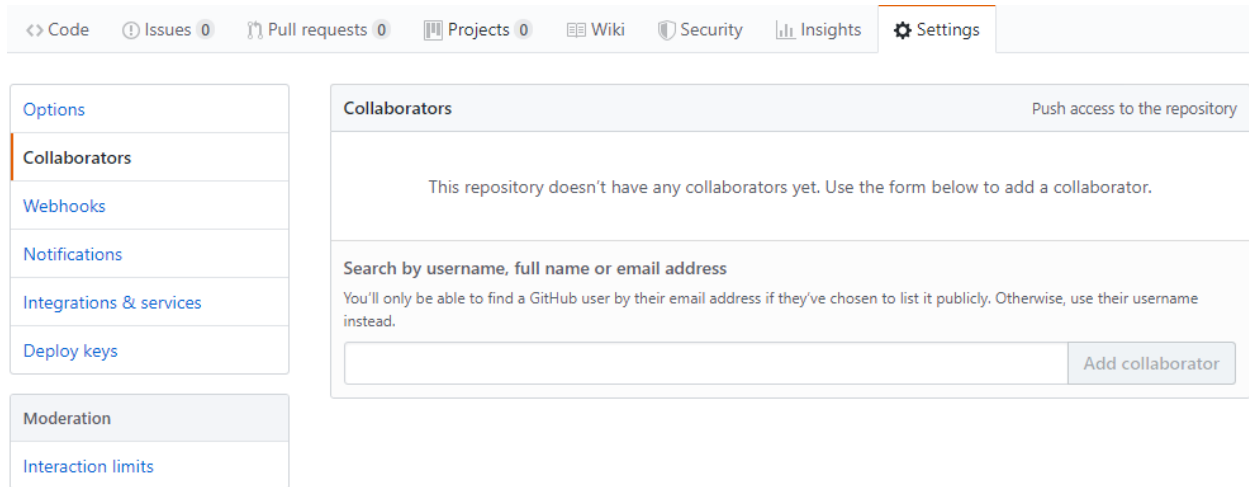
Git Workflow Assignment

Create your repo as *public*.

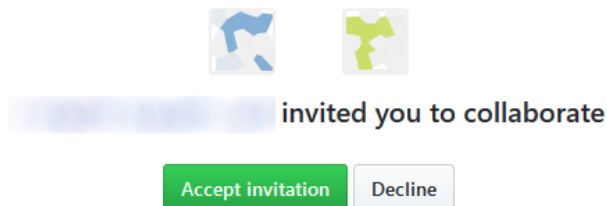
A public repo will let you and your team of 4 or 5 all be collaborators for the project.

Leave all other options as default.

Have the GitHub Owner add all of the rest of your team as collaborators.



For each collaborator, confirm the invitation that is sent to the email address associated with their GitHub accounts.



If you have a **team of 5**, you should have 1 GitHub Owner and 4 Collaborators.

You can fill in your names in the table below to remember who has which role.

Role	Name
GitHub Owner	
Collaborator #1	
Collaborator #2	
Collaborator #3	
Collaborator #4	

If you have a **team of 4** instead, The **GitHub Owner** can play the role of **Collaborator #4** as well.

If you have a **team of 3**, have the **GitHub Owner** also play the role of **Collaborator #4**; and **Collaborator #1** also play the role of **Collaborator #3**.

COMP 2800 Projects 2 Git Workflow Assignment

Step 2:

Create an initial commit on **master**.

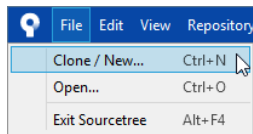
If you are the **GitHub Owner**, clone the GitHub repo onto your local machine. You can do this on the command line with the command:

```
git clone <url>
```

Example: `git clone https://github.com/<user>/<repo>.git`

This will clone the repo into your current directory, so make sure you are in the correct directory by changing directory using `cd`, you may wish to create an empty directory using `mkdir`.

In Sourcetree you can clone a GitHub repo by going to File → Clone / New ...



Clone

Cloning is even easier if you set up a [remote account](#)

Browse

Repository Type: ? No path / URL supplied

Browse

Local Folder:

⌵ Advanced Options

Clone

Enter your GitHub URL (ex: `https://github.com/<user>/<repo>.git`) and browse to the folder in which you wish to create your repo.

COMP 2800 Projects 2 Git Workflow Assignment

The **GitHub Owner** should create an initial commit to **master** to initialize the repo.

Name your commit: Initial Commit - Readme and basic HTML.

This commit should have 3 files:

Readme.txt

```
This is my readme file.
```

index.html

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 id="title">This is my simple webpage</h1>
    <div class="content">
      </div>
    </body>
  </html>
```

style.css

```
body {
  font-family: Calibri, "Segoe UI", Arial, sans-serif;
}
h1 {
  text-align: center;
}
.content {
  margin: 0 auto;
  width: 800px;
  text-align: center;
}
```

Push your changes to GitHub so that your other collaborating team members can see your changes.

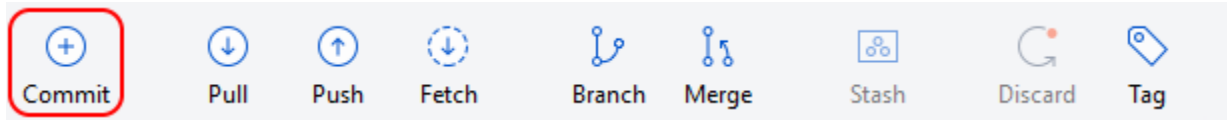
You can do this on the command line with the command:

```
git push -u origin <branch_name>
```

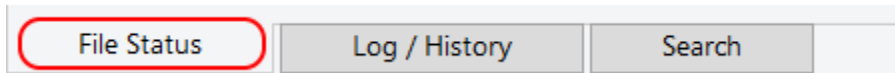
Example: `git push -u origin master`

COMP 2800 Projects 2 Git Workflow Assignment

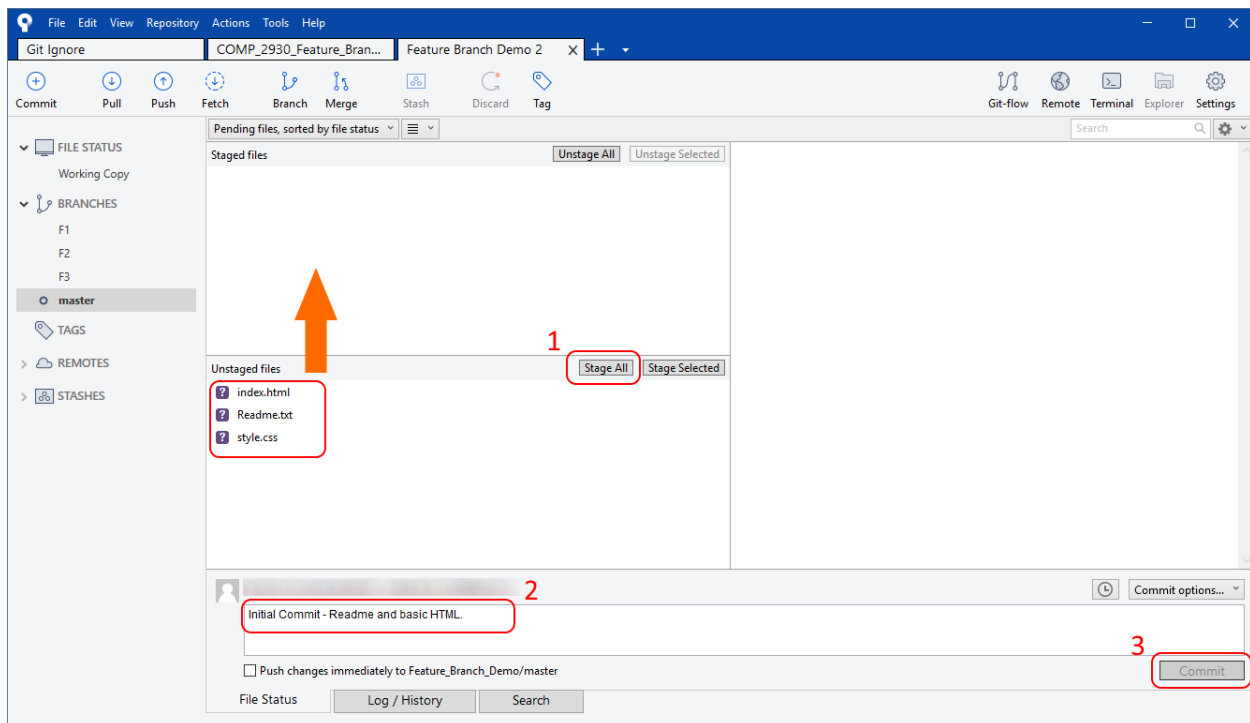
To make a commit in Sourcetree make sure you are on the commit screen (also known as the File Status tab). Click the commit button in the toolbar



Or switch to the File Status Tab by clicking the File Status on the bottom of the main window.

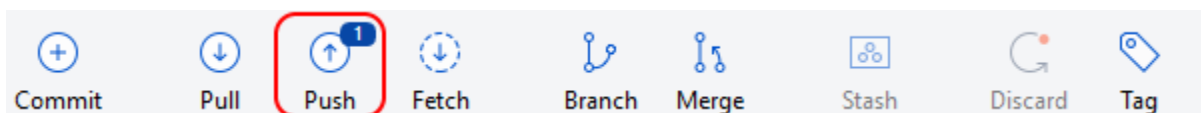


You should see all of your changes in the unstaged area.



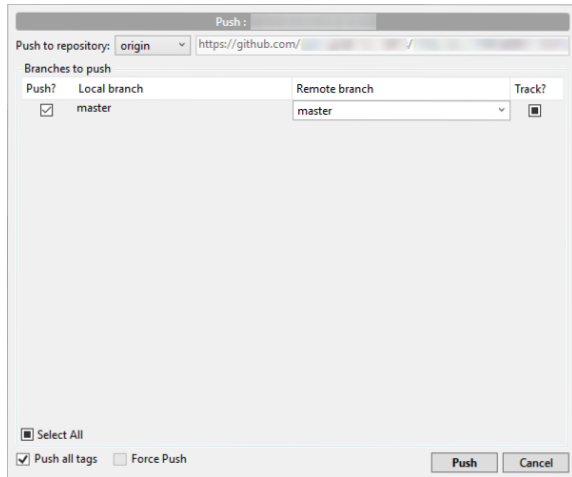
1. Click "Stage All" to add your changes to this commit.
You should see your changed files move from unstaged files to staged files.
2. Add your commit message.
3. Click "Commit".

When there are changes that haven't been pushed, you'll see a number beside the Push button in the toolbar menu. To push to GitHub click "Push" in the toolbar menu.

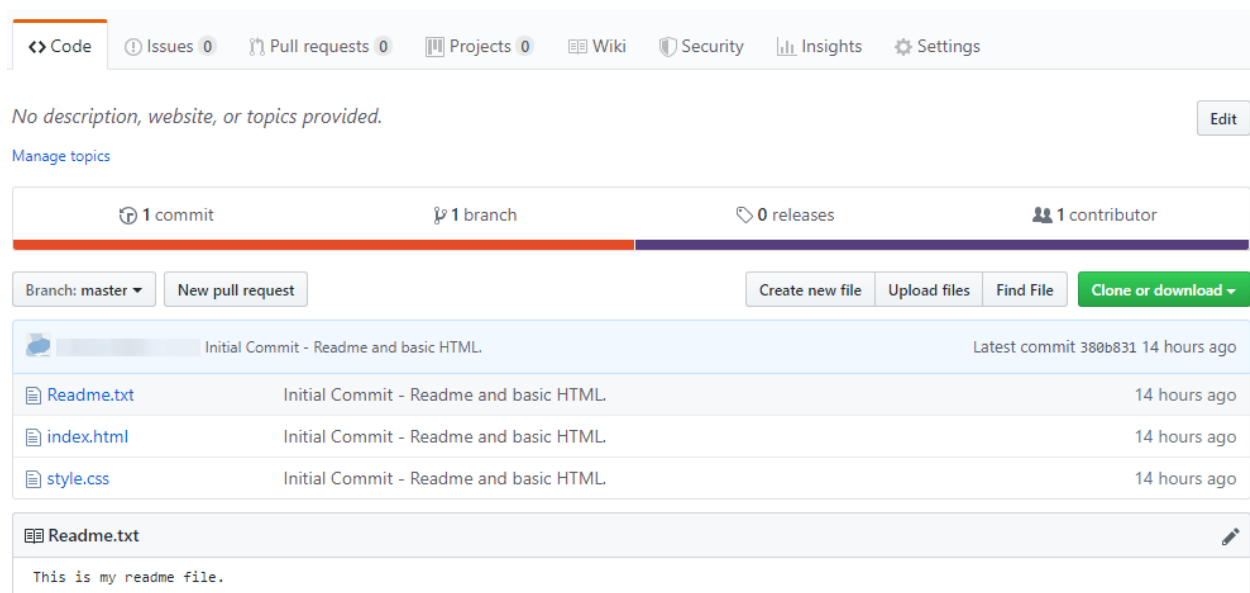


COMP 2800 Projects 2 Git Workflow Assignment

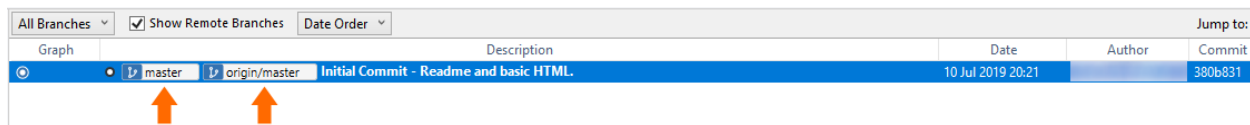
Confirm you want to push the master branch to origin (make sure the master branch is checked) and click "Push".



If your push worked, you should see this in GitHub:



And this in Sourcetree:



In Sourcetree you can see where your master branch is locally (called master) and where GitHub is (called origin/master). If there are un-pushed commits, your master and origin/master will not be on the same commit.

COMP 2800 Projects 2
Git Workflow Assignment

Step 3:

As **Collaborator #1**, create a commit into a feature branch.

If you are **Collaborator #1**, clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

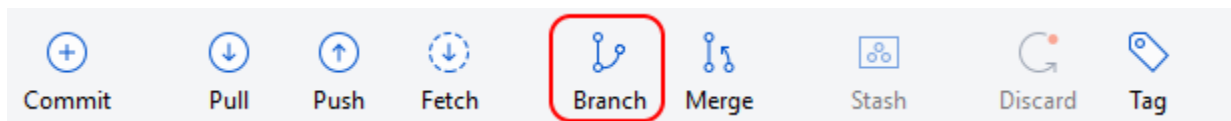
To create a new branch (and set this new branch as the current branch) on the command line:

```
git checkout -b <new_branch_name> <from_branch_name>
```

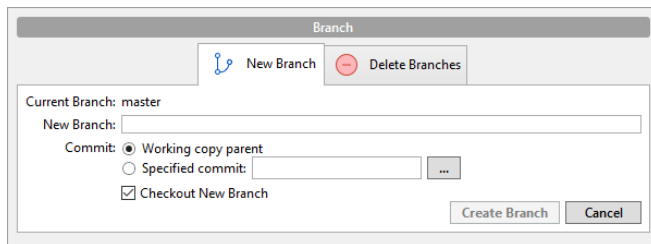
Example: `git checkout -b Alex_Smalltown_feature1 master`

You can confirm you are on the correct branch by using the: `git branch -a` command.

In Sourcetree you can create a new branch by clicking on the Branch button in the toolbar menu:



Add the name of your feature branch:



Make sure the "Checkout New Branch" is checked.

Click "Create Branch".

As **Collaborator #1** create a commit to this new feature branch.

Name your commit: Added javascript file - app.js.

This commit should have 2 file changes:

New file: app.js

```
const contentText = document.querySelector('.content');  
  
document.addEventListener('DOMContentLoaded', function() {  
    contentText.innerHTML = "This is the content";  
}, false);
```

COMP 2800 Projects 2 Git Workflow Assignment

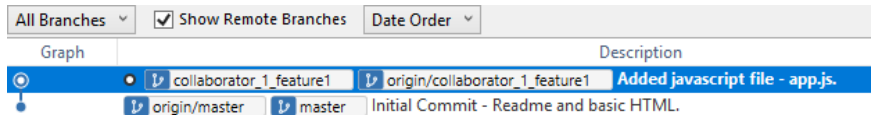
Change: index.html -

Add a `<script src="app.js"></script>` just before the closing body tag (`</body>`).

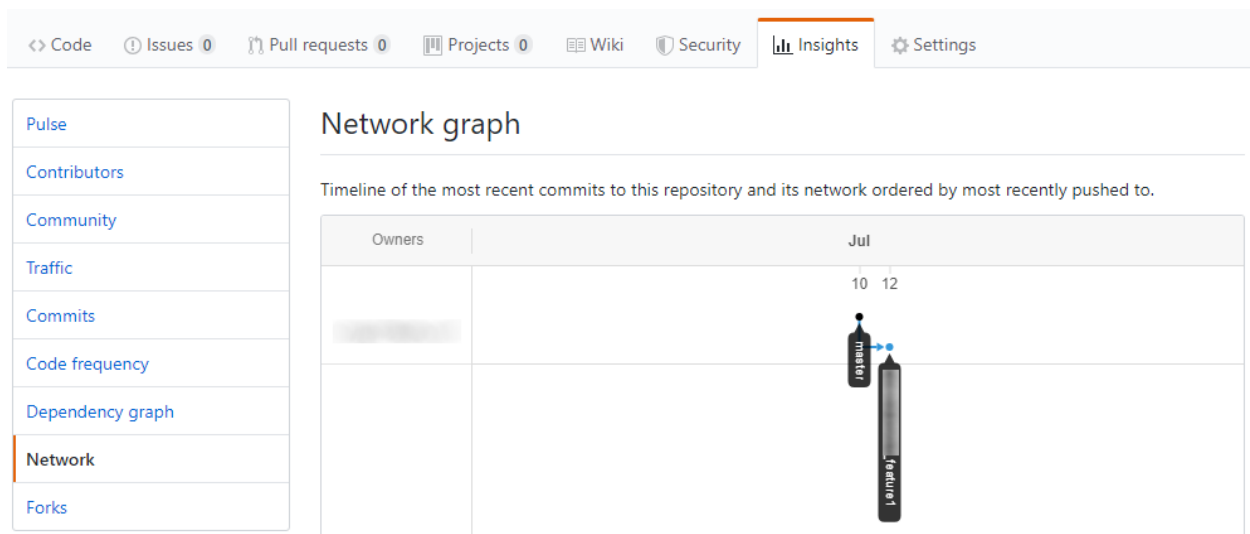
```
        <div class="content">
        </div>
        <script src="app.js"></script>
    </body>
</html>
```

Commit and Push your changes on the feature branch to GitHub.

Your Commit History in Sourcetree should look like this:



You should also be able to see your GitHub Network Graph (go to Insights → Network):



Step 4:

As **Collaborator #2**, create a commit into a feature branch.

If you are **Collaborator #2**, clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

IMPORTANT: Make sure that your new feature branch branches from **master** and not some other branch. For this Lab, DO NOT create feature branches from other feature branches.

COMP 2800 Projects 2 Git Workflow Assignment

As **Collaborator #2** create a commit to this new feature branch.

Name your commit: Changed content div class from content to container.

This commit should have 2 file changes:

Change file: index.html

Change the class for the <div> from content to container.

```
<body>
  <h1 id="title">This is my simple webpage</h1>
  <div class="container">
  </div>
</body>
</html>
```

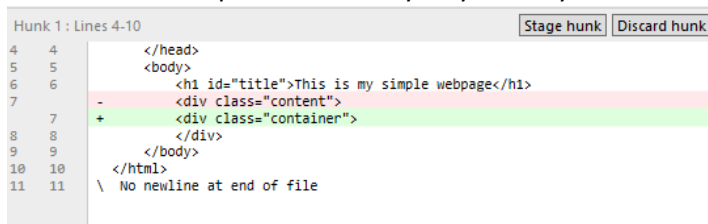
Change file: style.css

Change the class from content to container.

```
.container {
  margin: 0 auto;
  width: 800px;
  text-align: center;
}
```

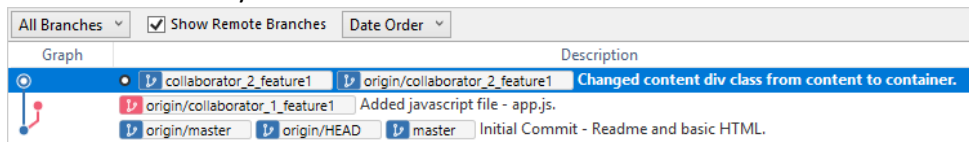
Commit and Push your changes on the feature branch to GitHub.

Note: Sourcetree provides an easy way to see your uncommitted changes to each file:



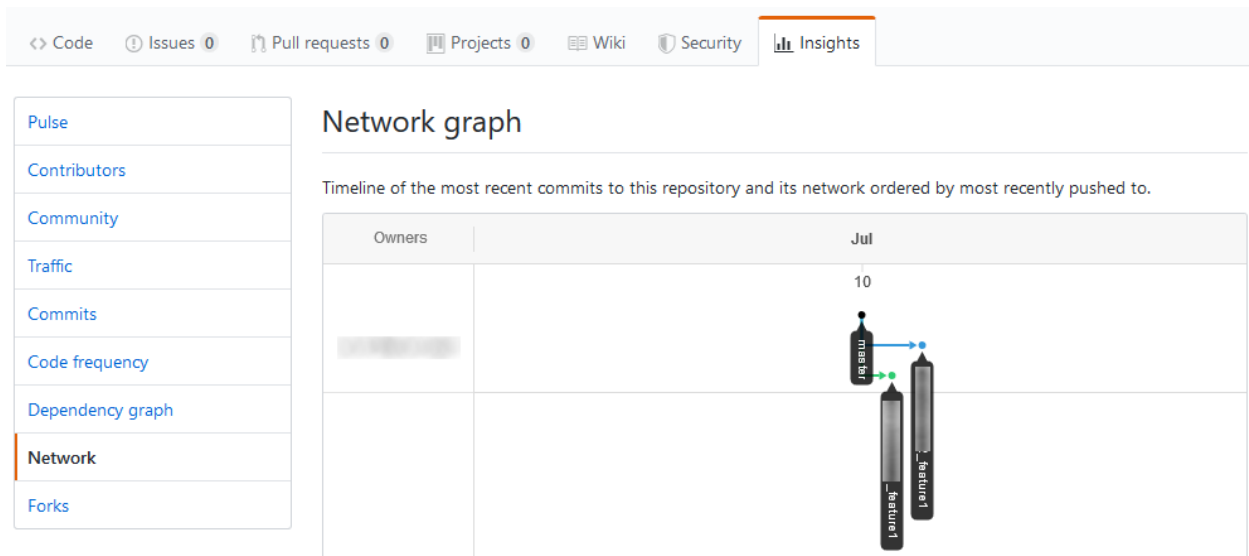
Red indicates a line removed. Green indicates a line added.

Your commit history so far should look like this in Sourcetree:



COMP 2800 Projects 2 Git Workflow Assignment

And your GitHub Network Graph should look like this:



Step 5:

As **GitHub Owner**, create a bug fix to the **master** branch.

In general, in the Feature-Branch Workflow, we want to avoid creating commits directly on the **master** branch. There is an exception to this rule, and this is when a bug fix is required on the **master** branch. Remember, our goal is to always make sure that the **master** branch has a well-tested and stable code base.

If you are the **GitHub Owner**, create a commit on **master**.

Name your commit: Added install steps to Readme.txt.

This commit should have 1 file change:

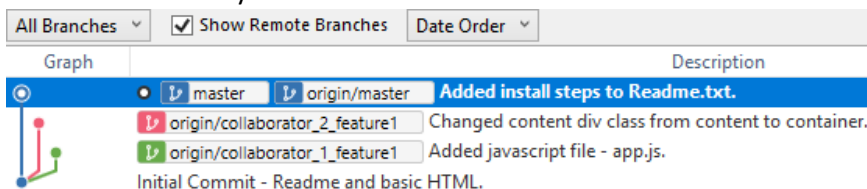
Change file: Readme.txt

Add the following 3 lines at the end.

```
This is my readme file.  
To run the website:  
1. Download the files.  
2. Open index.html in your browser
```

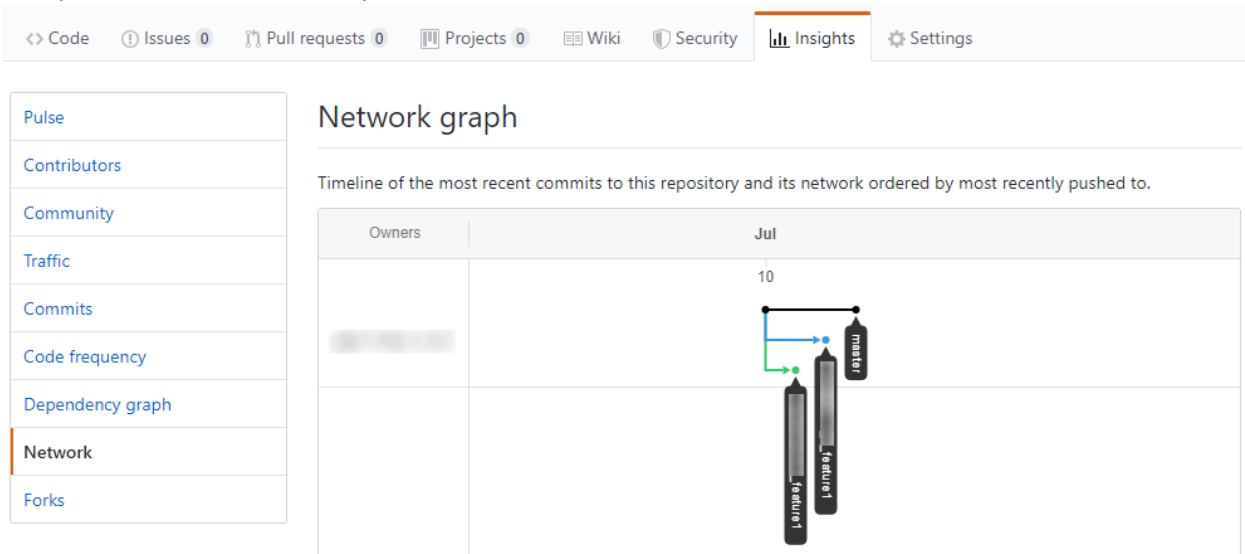
Commit and Push your changes on **master** to GitHub.

Your commit history so far should look like this in Sourcetree:



COMP 2800 Projects 2 Git Workflow Assignment

And your GitHub Network Graph should look like this:



Step 6:

As **Collaborator #3**, create a commit into a feature branch.

If you are **Collaborator #3**, clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

IMPORTANT: Make sure that your new feature branch branches from **master** and not some other branch. For this Lab, DO NOT create feature branches from other feature branches.

As **Collaborator #3** create a commit to this new feature branch.

Name your commit: Added an About Us Page.

This commit should have 1 file:

Add file: about_us.html

```
<html>
  <body> About Us
</body>
</html>
```

Commit and Push your changes on the feature branch to GitHub.

COMP 2800 Projects 2
Git Workflow Assignment

Step 7:

As **Collaborator #4**, create a commit into a feature branch.

If you are **Collaborator #4** clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

IMPORTANT: Make sure that your new feature branch branches from **master** and not some other branch. For this Lab, DO NOT create feature branches from other feature branches.

As **Collaborator #4** create a commit to this new feature branch.

Name your commit: Added a Help Page.

This commit should have 1 file:

Add file: help.html

```
<html>
  <body> Help
</body>
</html>
```

Commit and Push your changes on the feature branch to GitHub.

Step 8:

As **GitHub Owner**, create a commit into a feature branch.

If you are **GitHub Owner** create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

IMPORTANT: Make sure that your new feature branch branches from **master** and not some other branch. For this Lab, DO NOT create feature branches from other feature branches.

As **GitHub Owner** create a commit to this new feature branch.

Name your commit: Changed content to be clickable.

This commit should have 2 file changes:

Change file: index.html

Add a class for the content <div>. Call this new class **clickable**.

Add **Click me** as the text inside the <div>.

```
...
    <div class="content clickable"> Click me
    </div>
...
```

COMP 2800 Projects 2
Git Workflow Assignment

Change file: style.css

Add a `.clickable` style (with `cursor:hand;`) at the end of the file.

```
...  
  
.content {  
    margin: 0 auto;  
    width: 800px;  
    text-align: center;  
}  
  
.clickable {  
    cursor:hand;  
}
```

Commit and Push your changes on the feature branch to GitHub.

Step 9:

As **Collaborator #1**, merge your feature branch back to **master**.

For our lab, let's assume that **Collaborator #1** is finished with his/her feature and is ready for the rest of the team to include it in their work. What he/she needs to do is merge the feature branch back into the **master** branch. Merging a feature branch into **master** means now that **master** will have all the code from **master** (like it did before the merge) *plus* the code from the feature branch. Now the whole team can see the new feature and add their own improvements to it.

When merging, Git attempts to combine the file changes automatically. Sometimes when merging, the file changes are on very different files (you only changed index.html and I only changed app.js) or different locations within the same file (you only changed the <head> tag of index.html and I only changed the <body> tag of index.html). In these cases, since the changes are in unrelated locations, Git can do the merge automatically - without any interaction from you.

However, Git can't merge a file automatically when 2 people modify the same (or overlapping) areas of the same file. When this happens, Git marks the file as having a **merge conflict**. It's up to the team to decide what to do in this situation and there are a number of possible solutions - you could:

1. Take **all** of the 1st person's changes and **none** of the 2nd person's changes.
2. Take **none** of the 1st person's changes and **all** of the 2nd person's changes.
3. Take **neither** of the 1st person's changes **nor** the 2nd person's changes (this *very rarely* happens).
4. Take **both** the 1st person's changes **and** the 2nd person's changes.
5. Take **both** the 1st person's changes **and** the 2nd person's changes **plus** something more.
6. Create **something new**, which is neither *directly* the 1st person's nor the 2nd person's changes.

Since every merge conflict is different, it's best to discuss with your team (or at least the 2 people with the conflicting code changes) as to which solution is best. Whatever you and your team decide on to resolve the merge conflict, you need to make your change(s) to the affected file(s) before you can complete the merge.

We will discuss how to resolve a conflict when it happens later in this Lab.

COMP 2800 Projects 2
Git Workflow Assignment

In order to merge a feature branch into **master**, you first need to be in the **master** branch.

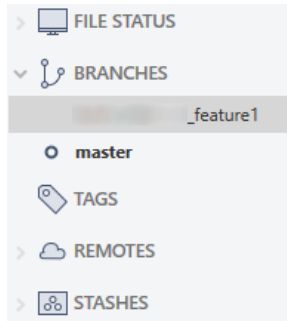
To change branches back to **master** on the command line:

```
git checkout <branch_name>
```

Example: `git checkout master`

You can confirm you are on the correct branch by using the: `git branch -a` command.

In Sourcetree you can change branches by double clicking on the branch in the branches section of the left tree menu:



The current branch is shown in **bold** - in the picture above the **master** branch is the current branch.

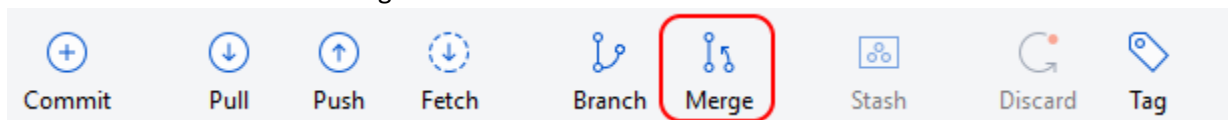
Once you are in the **master** branch you can merge into **master** your (**Collaborator #1**) feature branch.

On the command line use the git merge command:

```
git merge <feature_branch_name>
```

Example: `git merge Alex_Smalltown_feature1`

In Sourcetree click on the "Merge" button in the toolbar menu:



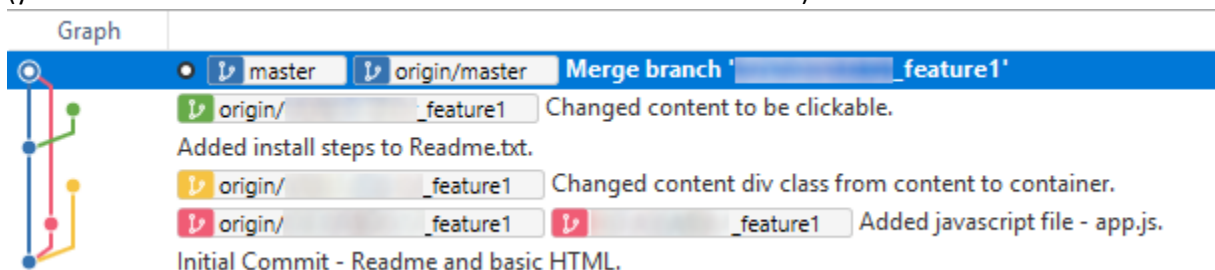
Select your (**Collaborator #1**) feature branch and click OK.



Push your changes back to GitHub (origin).

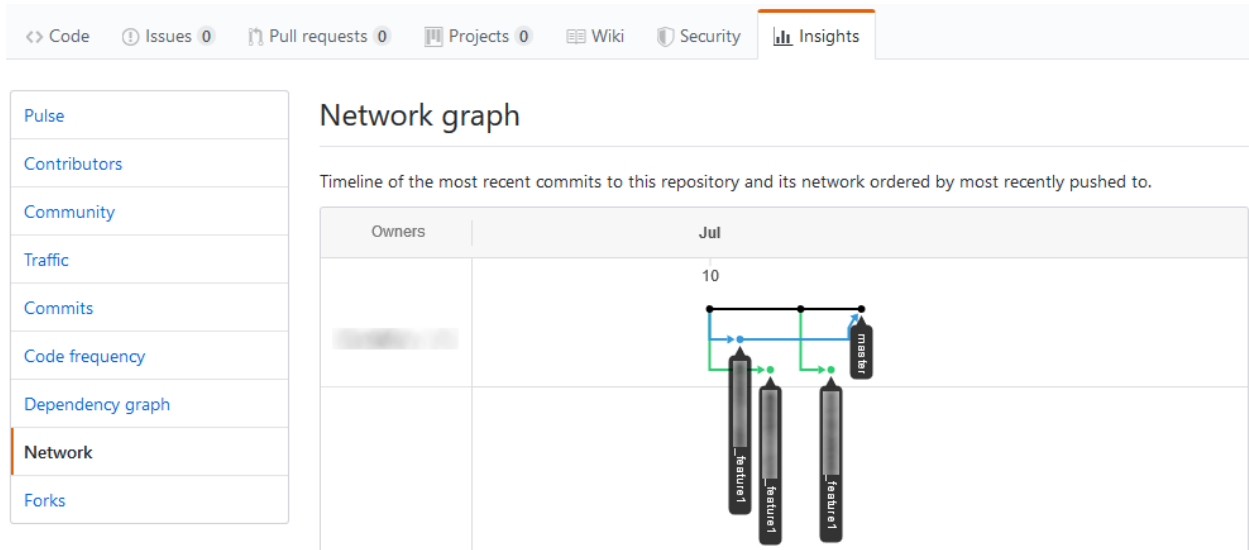
Your commit history in Sourcetree should look similar to this

(yours should have a few additional commits and feature branches):



COMP 2800 Projects 2 Git Workflow Assignment

And your GitHub Network Graph should look similar to this
(yours should have a few additional commits and feature branches):



Step 10:

As **Collaborator #2**, merge your feature branch back to **master**.

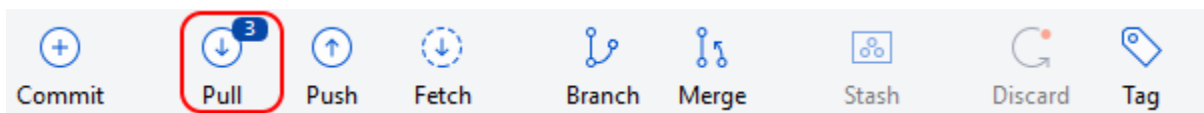
Confirm you are on the **master** branch before merging.

Pull from **master** to ensure you are current with the changes from GitHub (origin).

IMPORTANT: In order to be able to push to GitHub, you must be current with the changes on GitHub. If GitHub has changes that you haven't pulled yet, you won't be able to push your changes.

If you type `git status` on the command line and it gives you a message saying something like:
Your branch is behind 'origin/master' by 3 commits
you should do a `git pull` to make sure you have all the changes from GitHub.

In Sourcetree you can see that you are not up-to-date and that you are behind the origin when the pull icon in the toolbar menu has a number beside it.



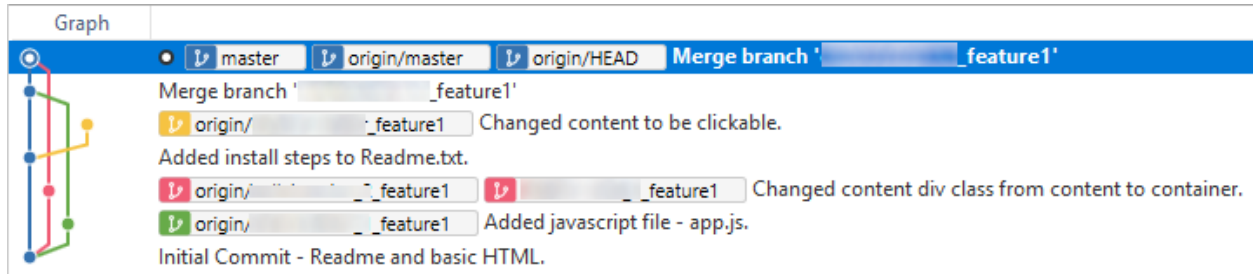
Click Pull to pull all the changes from GitHub.

Merge into **master** your feature branch.

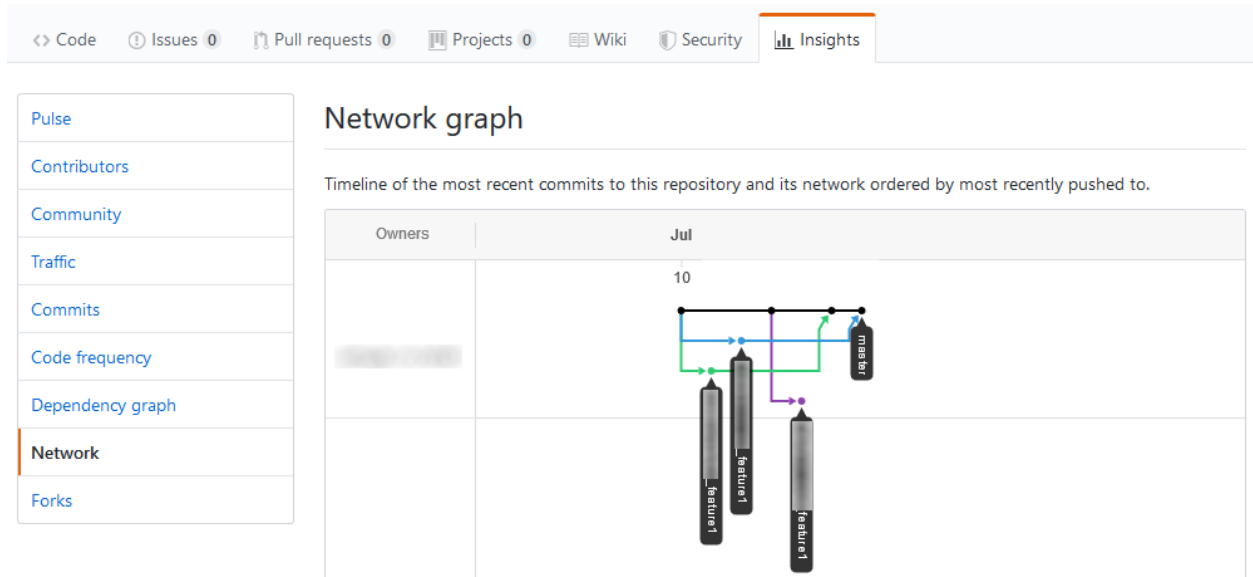
Push your (**Collaborator #2**) changes back to GitHub (origin).

COMP 2800 Projects 2 Git Workflow Assignment

Your commit history in Sourcetree should look similar to this
(yours should have a few additional commits and feature branches):



And your GitHub Network Graph should look similar to this
(yours should have a few additional commits and feature branches):



Step 11:

As **Collaborator #3**, merge your feature branch back to **master**.

Confirm you are on the **master** branch before merging.

Pull from **master** to ensure you are current with the changes from GitHub (origin).

Merge into **master** your feature branch.

Push your (**Collaborator #3**) changes back to GitHub (origin).

COMP 2800 Projects 2
Git Workflow Assignment

Step 12:

As **Collaborator #4**, merge your feature branch back to **master**.

Confirm you are on the **master** branch before merging.

Pull from **master** to ensure you are current with the changes from GitHub (origin).

Merge into master your feature branch.

Push your (**Collaborator #4**) changes back to GitHub (origin).

Step 13:

As **GitHub Owner**, merge your feature branch back to **master**.

Confirm you are on the **master** branch before merging.

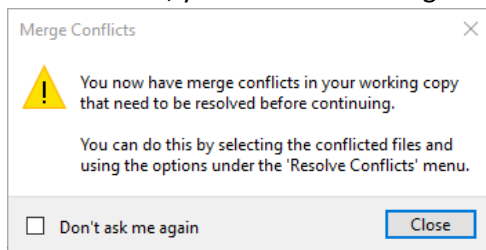
Pull from **master** to ensure you are current with the changes from GitHub (origin).

Merge into **master** your feature branch. This time when we do the merge Git won't be able to merge the changes automatically. Git will notice that index.html has been changed by you and by **Collaborator #2**.

Git on the command line, will show an error message like this:

```
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

In Sourcetree, you will see a message like this:



If you open index.html, you may notice now that index.html contains some interesting code:

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 id="title">This is my simple webpage</h1>
    <<<<<<< HEAD
      <div class="container">
        =====
        <div class="content clickable"> Click me
      >>>>>>> *****_feature1
      </div>
    <script src="app.js"></script>
  </body>
</html>
```

COMP 2800 Projects 2 Git Workflow Assignment

Git has combined both your changes (as **GitHub Owner**) and **Collaborator #2**'s changes.

Collaborator #2's changes are the changes highlighted in **green** above and marked with **<<<<<< HEAD**.

Your changes (as **GitHub Owner**) are highlighted in **magenta** and marked with **>>>>>> *****_feature1** - in your case this will be the name of your feature branch.

Both of the changes, are separated by **=====** (which is highlighted in **yellow**).

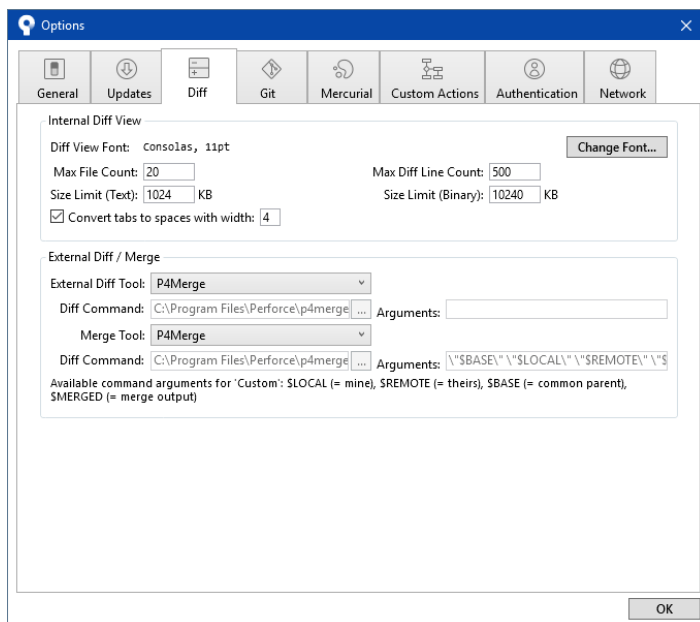
If you wish, you can open any text editor, open index.html and make the changes manually (choosing any of the 6 options mentioned earlier in the section for solving merge conflicts). Make sure to remove the Git markings:

- **<<<<<< HEAD**
- **=====**
- **>>>>>> *****_feature1**

Sourcetree, when it recognizes a Git merge conflict, will let you compare the 3 files to help you resolve the conflict. There are a number of great 3-way merge tools out there. Here are a few 3-way merge tools that Sourcetree supports directly:

- P4Merge (also known as Helix Merge and Diff Tools)
- KDiff3
- DiffMerge
- Beyond Compare
- TortoiseMerge
- WinMerge
- Araxis Merge

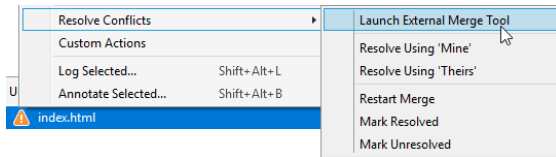
You can also configure Sourcetree to use another 3-way merge tool of your choice.



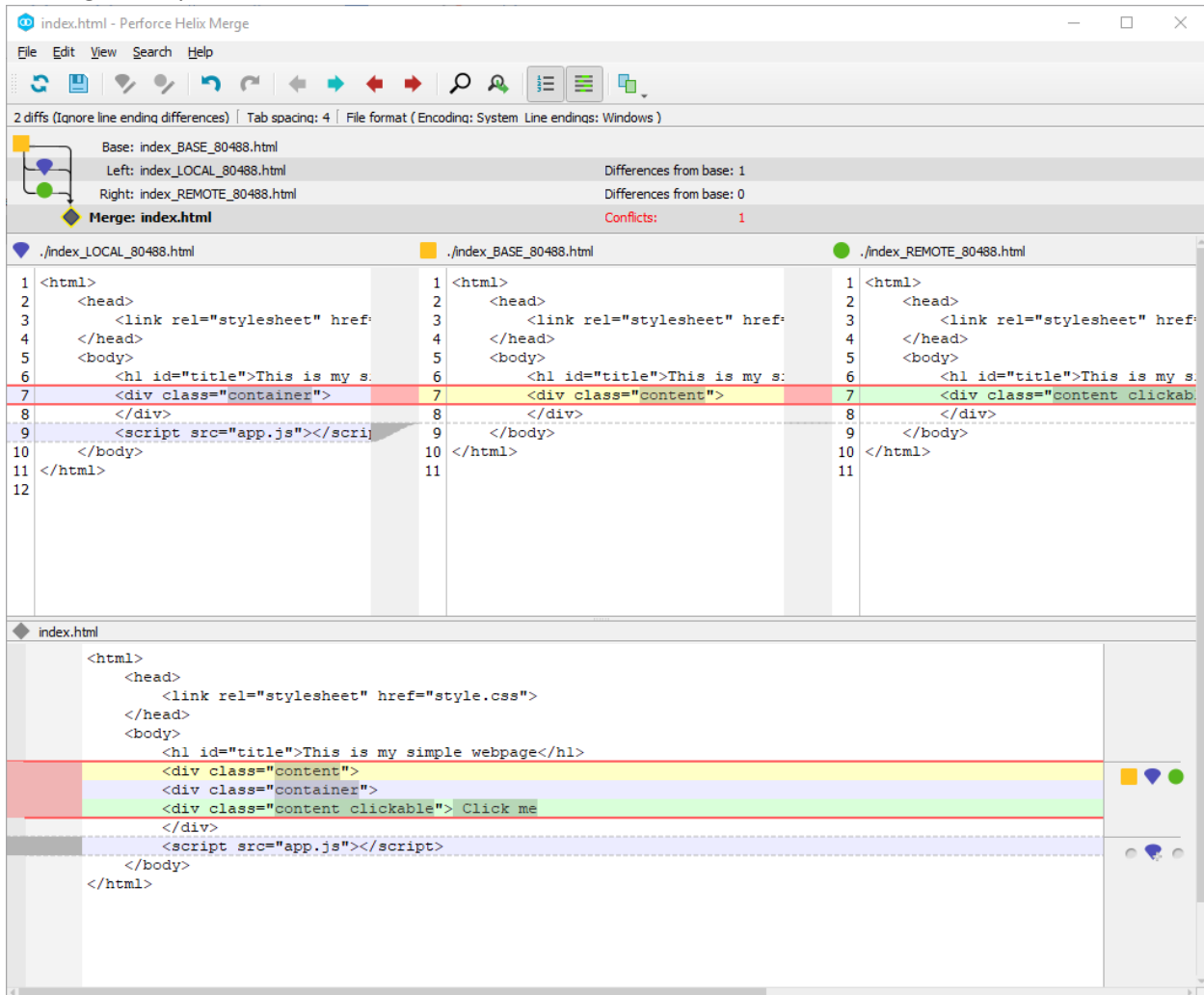
COMP 2800 Projects 2 Git Workflow Assignment


This Lab will use P4Merge (Helix Merge) to resolve the Git conflict on index.html.

To start P4Merge from within Sourcetree, right click on index.html, click "Resolve Conflicts" and then click "Launch External Merge Tool".




P4Merge will open and show a window like this:

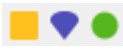


Your changes (as **GitHub Owner**) are on the left with the blue icon .

The changes from and **Collaborator #2** are on the right with the green icon .

And the base (which has neither of the changes is shown in the middle) with the yellow icon .

COMP 2800 Projects 2 Git Workflow Assignment

You can use the editor on the bottom to select which changes you wish to keep by clicking the icons on the bottom right .

If you want to select multiple changes, hold down the SHIFT key on your keyboard while clicking the icons.

For this lab, we want to keep a combination of both changes. (Remember that **GitHub Owner** wanted to add a `clickable` class to the `div` and that **Collaborator #2** wanted to change the content content to container)

Your bottom window should match the following when complete:

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 id="title">This is my simple webpage</h1>
    <div class="container clickable"> Click me
  </div>
    <script src="app.js"></script>
  </body>
</html>
```

Click the Save icon  when you are done.

Close P4Merge.

When you are back in Sourcetree you should see that `index.html` is staged and ready to commit with our merged changes.

If you see a file `index.html.orig`, you can safely delete this file. P4Merge created this file as a backup because it doesn't want to override your files in case something goes wrong. To delete this file from within Sourcetree, right click `index.html.orig` and click "Remove".

If you don't have Sourcetree or P4Merge and want to use a basic text editor instead to make the changes, you can open `index.html` and make the changes yourself.

With which ever tool you use to resolve the git merge conflict `index.html` should look like this once complete:

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 id="title">This is my simple webpage</h1>
    <div class="container clickable"> Click me
  </div>
    <script src="app.js"></script>
  </body>
</html>
```

COMP 2800 Projects 2 Git Workflow Assignment

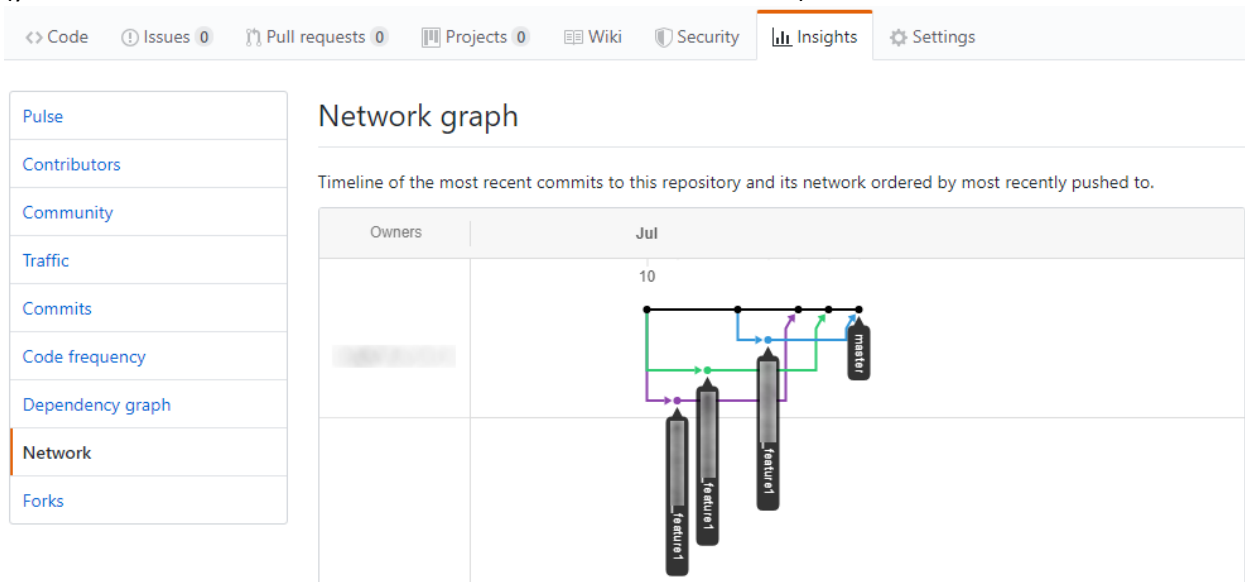
Commit your merge changes.

Push your merged changes back to GitHub (origin).

Step 14:

Each student will submit a screenshot of their GitHub Network Graph for the Feature-Branch Git Workflow.

Your Feature-Branch GitHub Network Graph should look like this (yours should have a few additional commits and feature branches):



Steps for Creating a GitFlow Workflow:

Using the GitFlow Workflow, create a GitHub Repo with your team, create a **dev** branch, create commits on feature branches (which branch from **dev**), merge all your feature branches back to **dev**, integration test and commit changes to **dev** before merging **dev** back to **master**.

Step 15:

Create a GitHub Repo.

Give your repo the name: `COMP_2800_GitFlow_Workflow`

Create your repo as *public*.

Have the **GitHub Owner** add all of the rest of your team as collaborators.

Step 16:

Create an initial commit on **master**.

If you are the **GitHub Owner**, clone the GitHub repo onto your local machine.

COMP 2800 Projects 2
Git Workflow Assignment

The **GitHub Owner** should create an initial commit to **master** to initialize the repo.

Name your commit: `Initial Commit - Readme.`

This commit should have 1 file:

New file: Readme.txt

```
This is my readme file.
```

Push your changes to GitHub so that your other collaborating team members can see your changes.

Step 17:

Create a **dev** branch with an initial commit on **dev**.

If you are the **GitHub Owner**, create a new branch called **dev**.

The **GitHub Owner** should create an initial commit to **dev** to initialize the branch.

Name your commit: `Initial Commit on dev.`

This commit should have 1 file:

New File: dev.txt

```
This is dev.
```

Step 18:

As **Collaborator #1**, create a commit into a feature branch.

If you are **Collaborator #1**, clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

IMPORTANT: Make sure that your new feature branch branches from **dev** and not some other branch.

For this Lab, DO NOT create feature branches from other feature branches.

As **Collaborator #1** create a commit to this new feature branch.

Name your commit: `Feature 1.`

This commit should have 1 file:

New file: F1.txt

```
Feature 1
```

Commit and Push your changes on the feature branch to GitHub.

Step 19:

As **Collaborator #2**, create a commit into a feature branch.

If you are **Collaborator #2**, clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

IMPORTANT: Make sure that your new feature branch branches from **dev** and not some other branch.
For this Lab, DO NOT create feature branches from other feature branches.

As **Collaborator #2** create a commit to this new feature branch.

Name your commit: **Feature 2**.

This commit should have 1 file:

New file: F2.txt

```
Feature 2.
```

Commit and Push your changes on the feature branch to GitHub.

Step 20:

As **GitHub Owner**, create a bug fix to the **master** branch.

In general, in the Feature-Branch Workflow, we want to avoid creating commits directly on the **master** branch. There is an exception to this rule, and this is when a bug fix is required on the **master** branch. Remember, our goal is to always make sure that the **master** branch has a well-tested and stable code base.

If you are the **GitHub Owner**, create a commit on **master**.

Name your commit: **Bug Fix**.

This commit should have 1 file change:

Change file: Readme.txt

Add the following line at the end.

```
This is my readme file.  
Bug Fix.
```

Commit and Push your changes on **master** to GitHub.

Step 21:

As **Collaborator #3**, create a commit into a feature branch.

If you are **Collaborator #3**, clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

COMP 2800 Projects 2
Git Workflow Assignment

IMPORTANT: Make sure that your new feature branch branches from **dev** and not some other branch.
For this Lab, DO NOT create feature branches from other feature branches.

As **Collaborator #3** create a commit to this new feature branch.

Name your commit: `Feature 3.`

This commit should have 1 file:

New file: F3.txt

```
Feature 3.
```

Commit and Push your changes on the feature branch to GitHub.

Step 22:

As **Collaborator #4**, create a commit into a feature branch.

If you are **Collaborator #4** clone the GitHub repo onto your local machine.

Create a feature branch in the following name format:

firstName_lastName_featureName

ex: **Alex_Smalltown_feature1**

IMPORTANT: Make sure that your new feature branch branches from **dev** and not some other branch.
For this Lab, DO NOT create feature branches from other feature branches.

As **Collaborator #4** create a commit to this new feature branch.

Name your commit: `Feature 4.`

This commit should have 1 file:

New file: F4.txt

```
Feature 4.
```

Commit and Push your changes on the feature branch to GitHub.

Step 23:

As **Collaborator #1**, merge your feature branch back to **dev**.

Confirm you are on the **dev** branch before merging.

Push your changes back to GitHub (origin).

COMP 2800 Projects 2
Git Workflow Assignment

Step 24:

As **Collaborator #2**, merge your feature branch back to **dev**.

Confirm you are on the **dev** branch before merging.

Pull from **dev** to ensure you are current with the changes from GitHub (origin).

Merge into **dev** your feature branch.

Push your (**Collaborator #2**) changes back to GitHub (origin).

Step 25:

As **Collaborator #3**, merge your feature branch back to **dev**.

Confirm you are on the **dev** branch before merging.

Pull from **dev** to ensure you are current with the changes from GitHub (origin).

Merge into **dev** your feature branch.

Push your (**Collaborator #3**) changes back to GitHub (origin).

Step 26:

As **Collaborator #4**, merge your feature branch back to **dev**.

Confirm you are on the **dev** branch before merging.

Pull from **dev** to ensure you are current with the changes from GitHub (origin).

Merge into master your feature branch.

Push your (**Collaborator #4**) changes back to GitHub (origin).

Step 27:

Integration test your code on **dev** before merging to **master**.

After you and your team have merged all your features into dev, you start to do integration testing on the combination of all the features. You discover a few bugs and changes you want to do before considering the code fully tested and stable.

As **Collaborator #1**, create a commit to the **dev** branch.

Name your commit: `Integration Testing`.

This commit should have 1 file change:

New file: all.txt

Features 1 to 4 complete.

COMP 2800 Projects 2 Git Workflow Assignment

Commit and Push your changes on the **dev** branch to GitHub.

Merge into **master** the integration tested **dev** branch.

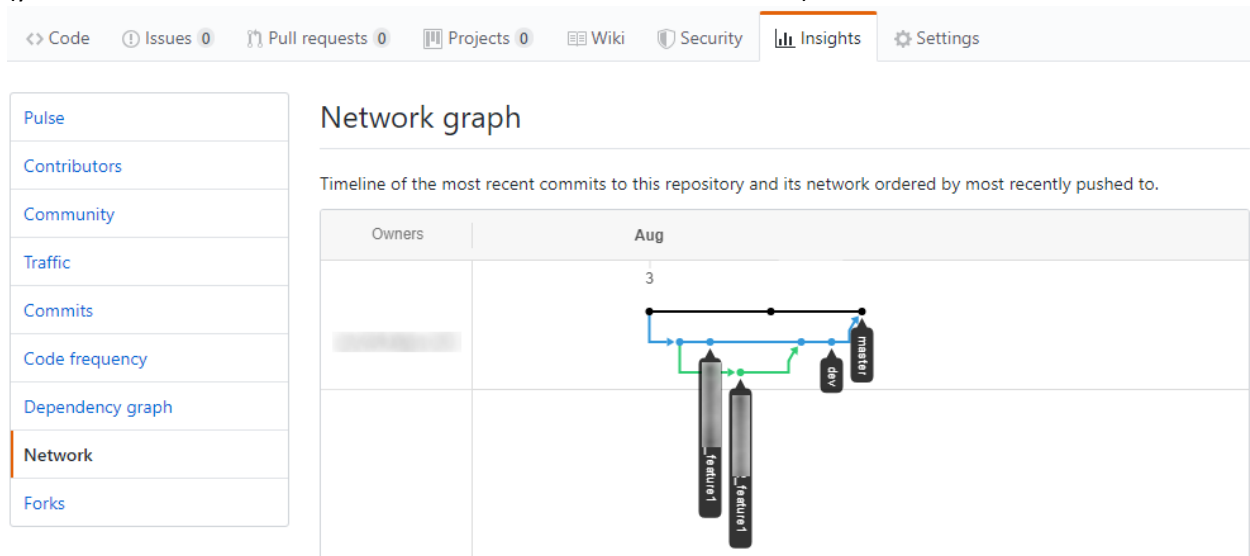
Push your (Collaborator #1) changes back to GitHub (origin).

Step 28:

Each student will submit a screenshot of their GitHub Network Graph for the GitFlow Git Workflow.

Your GitFlow GitHub Network Graph should look like this

(yours should have a few additional commits and feature branches):



COMP 2800 Projects 2
Git Workflow Assignment

Marking Criteria:

Criteria	Marks
<p>Complete and submit a GitHub network graph (from github.com) of the Git Workflows described in the instructions:</p> <ul style="list-style-type: none">• Feature-Branch Git Workflow• GitFlow Workflow <p>For each of the completed Git Workflows:</p> <ul style="list-style-type: none">• Each team member has at least:<ul style="list-style-type: none">○ 1 feature branch with their name on it in the format firstName_lastName_featureName ex: Alex_Smalltown_feature1○ 1 commit in each feature branch before merging back• All commits should be committed to the feature branches. There should be no commits on the master with the exception of the one bug fix on master and the merges from feature branches.• Feature branches, once complete and tested, must be merged back into master (or to dev, integration tested, and then to master, in the case of the GitFlow workflow). Do not delete the branches once they are merged back to master.• All merge conflicts must be properly handled and resolved before committing.	<p>5 marks / workflow</p>
Total:	10 marks

Submission Requirements:

Submission:	File name:
<p>Although you will be working with your group to create the correct commits in the Git Workflows, your submission will be individual.</p> <p>Submit the following screenshots of your completed network graph (from github.com) for the 2 Git Workflows:</p> <ul style="list-style-type: none">• Feature-Branch Git Workflow• GitFlow Workflow	<p>Feature-Branch_Workflow.jpg GitFlow_Workflow.jpg</p>
<p>A text file containing your GitHub URL.</p> <ul style="list-style-type: none">• Example: <a href="https://github.com/<user>/<repo>.git">https://github.com/<user>/<repo>.git <p>Make sure that your GitHub Repo is public.</p>	<p>GitHub_URL.txt</p>