**GIT (Version Control) Project**

**Project Description:**

Plan and implement GIT source control for all of our major projects. Separate LH SW updates and BeamBoss.

**Project Time Frame:**

Must be completed by December 17th

**Project Members:**

Jan, Adnan, Harold, CF, Michael, Dag Mange

**Project Steps:**

* Choose a server to store GIT main repository.
* Install GIT locally on all relevant PCs and setup access for all developers to the main repository.
* Identify which GIT processes should be used by Polewall and Team

**Timeline for Project Completion:**

*December 12th:* Present and discuss server options for remote storage.

*December 12 th:* Finalize Server selection

*December 12th:* Evaluate the recommended GIT Processes

Discuss using GIT GUI’s (same, or up to individual developer)

*December 13th:* GIT to be installed on selected Server

*December 13th:* Finalize the GIT processes that we are to follow

*December 13th:* Install GIT locally on all participating machines

*December 16th:* Start using GIT for Version Control

*December 17th:* Evaluate the Project; make changes to the GIT Processes if needed.

**GIT Server Options:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Pros** | **Cons** | **Decision** |
| Dropbox | Already installed and used by majority of team | Can cause problems in the future with revisions |  |
| Lockless server | Readily Available | Ties Polewall to Lockless. |  |
| GIT HUB | Can be purchased online, designed for GIT projects storage. Also comes with GITHUB GUI. | Only get 10 private repositories for the $25/month plan. $50/month gives us 20 private repositories |  |
| Purchase Server for Polewall | Gives Polewall Complete control over our content. | Cost, setup, management |  |
| Store with ProIsp | Already storing our websites and email. |  |  |
|  |  |  |  |

Who will be in charge of setting up GIT on the Server?

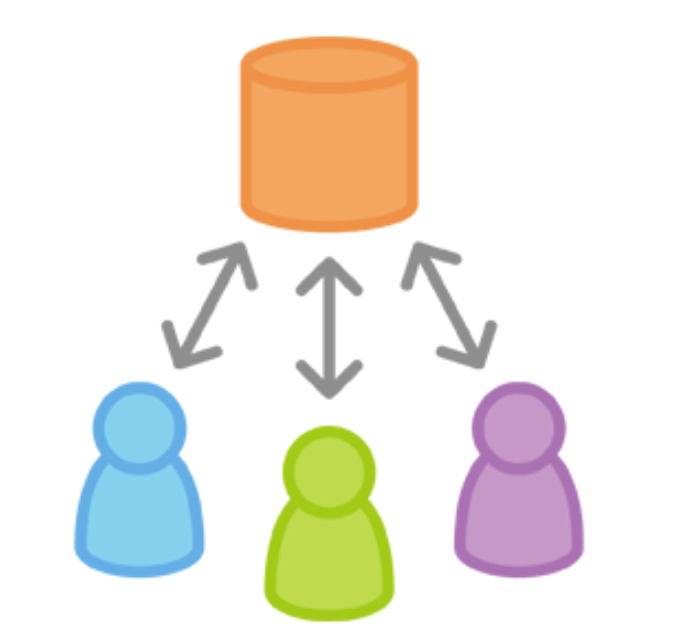
**Steps to Install GIT**

**Windows:**

**Mac:**

**Select Workflow Type:**

**Centralized:**

****

Easiest method to transition too, and will not be as difficult to set up and use. But it is not as scalable as the other workflow choices. It can be affective for the current team, but once the R&D team expands we would need to switch to a different workflow to stay organized.

I expect this method will not change much regarding how we have been keeping track of our versions. It will just implement GIT tracking and ability to write notes about each feature.

Each developer will still have a local copy of the entire project, so they can work independently of all other changes to the project. With the ability to just commit things locally until it is convenient for them to push their changes to the server.

Centralized workflow will central around one main branch, *MASTER*. With all changes being committed to this branch.

*“Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are stored locally—they’re completely isolated from the central repository. This lets developers defer synchronizing upstream until they’re at a convenient break point.*

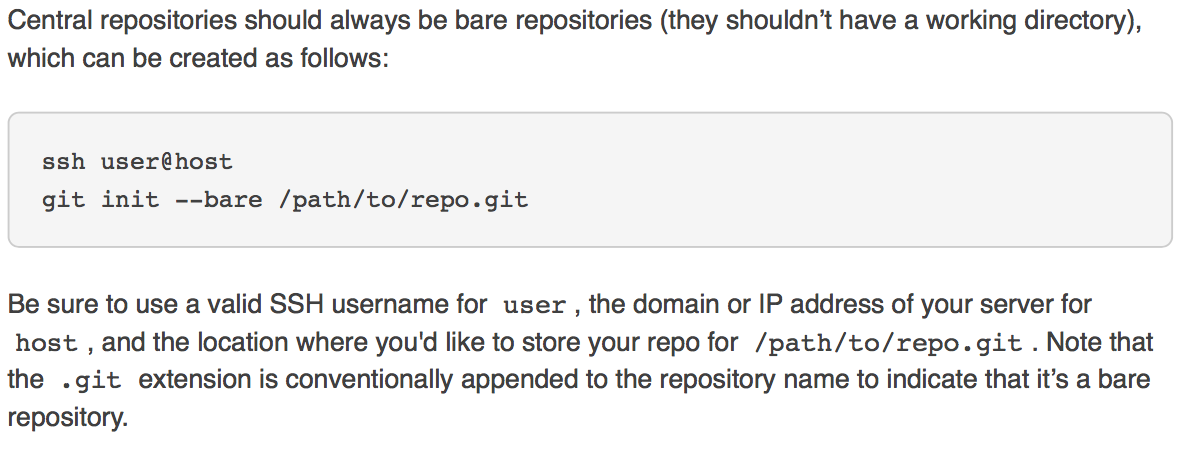
*To publish changes to the official project, developers “push” their local master branch to the central repository. This is the equivalent of svn commit, except that it adds all of the local commits that aren’t already in the central master branch.”*

Before each developer can commit their local commits to the central repository, they will need fetch the updated central commits. This process allows them to add their changes to what everyone else has already done.

If local changes conflict with upstream commits, the developer will have a chance to manually resolve the conflicts. This is supposed to make it easier for developers to handle their own merges.

Step One:

Someone will need to create the central repository. Since we will start fresh, we can implement an empty repository.



Step 2:

Everyone will need to clone the central repository to his or her local computer, creating a local copy of the entire project.

This is accomplished by using the *Git Clone* command:



Step 3:

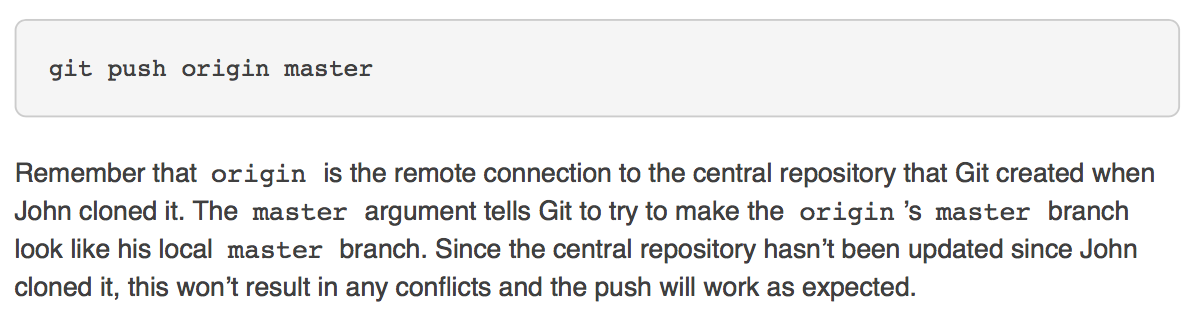
Developers begin working on their *Features* code. Using the standard git processes of *edit, stage and commit*.



Developers are able to work on their *features*, without needing to worry about what is in the central repository or what other developers are doing. Since their work is private until they choose to publish the *feature* to the central repository.

Step 4:

Publish the *feature* to the central repository, allowing other team members to access it. This is done with the *Git Push* command.

****

**Feature Branch:**

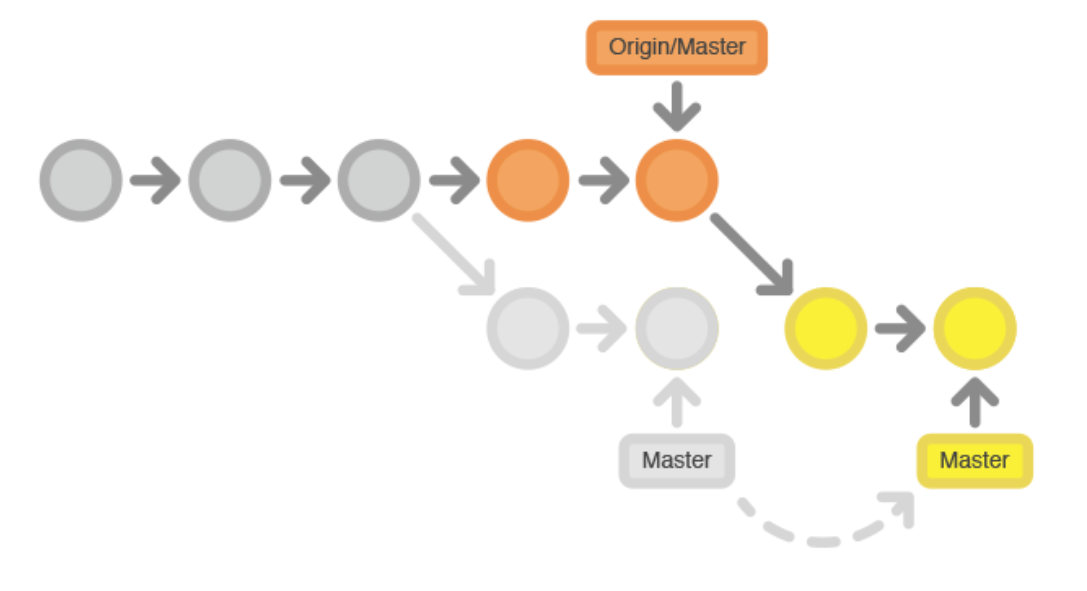
Now that developer A has commited their feature to the central repository it has become official. Developer B can still attempt to commit their work, but they will receive an error message, notifying them that they need to Pull developer A’s changes into their local repository and then integrate them with the new changes before pushing to the central repository.



Developers will use the *Git Pull* command, to incorporate upstream changes into their repository. This command will pull the entire upstream commit history into the developer’s local repository and tries to integrate it with their local commits.

****

The *–rebase* option tells Git to move all of the Developer B’s commits to the tip of the *Master* branch after sinking it with the changes in the central repository. This can be seen in the figure below.

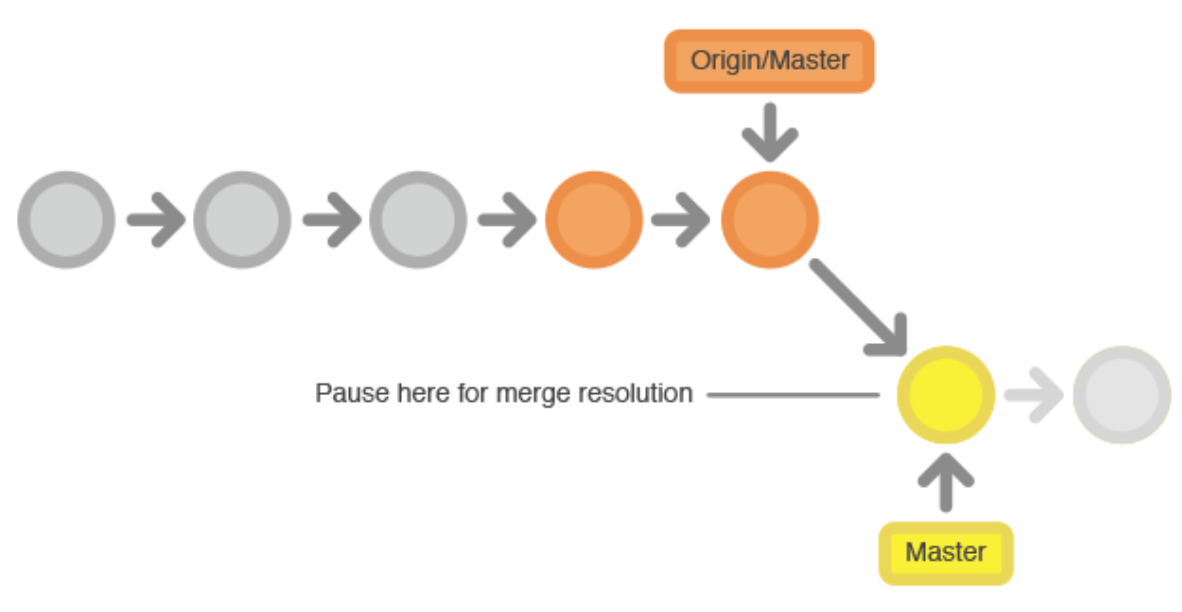


If you forget the *–rebase* option the pull will still work. But the developer would wind up with a superfluous “merge commit” every time someone needs to synchronize with the central repository. So it is better in the Central Workflow to *–rebase* instead of generating a merge commit.

*“Rebasing works by transferring each local commit to the updated master branch one at a time. This means that you catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit. This keeps your commits as focused as possible and makes for a clean project history. In turn, this makes it much easier to figure out where bugs were introduced and, if necessary, to roll back changes with minimal impact on the project.”* (atlassian)

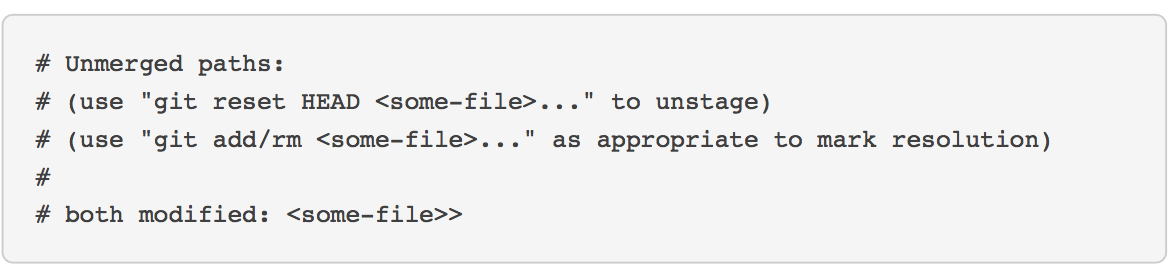
If Developer A and B are working on unrelated features, it is very unlikely that the *–rebase* process will generate conflicts. If a conflict does occur, Git pauses the rebase at the current commit and outputs the following message along with instructions:



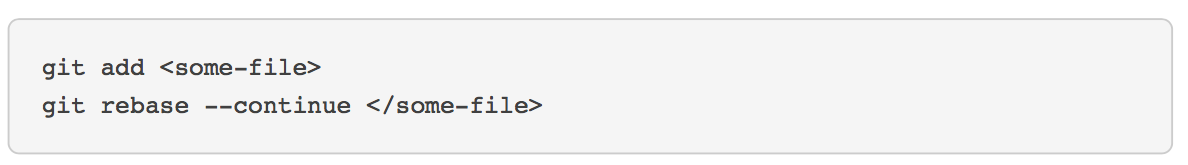


An advantage of Git is that anyone can resolve their own merge conflicts.

* Run a *git status* to see the problem
* Conflicted files will appear in the Unmerged paths section:



The developer then edits the file(s) to their liking. When they are happy with the results, they stage the new file(s) in the normal fashion and let *git rebase* do the rest:

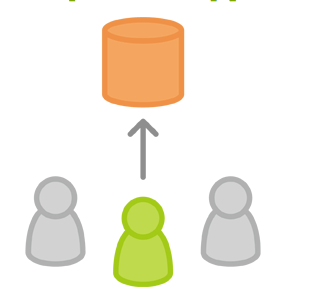
****

Git will now move on to the next commit and repeat the process for an other commits that have generated conflicts.

If the developer runs into a problem and wants to revert to the previous state, they just need to run the *git rebase -–abort* command. This will take the developer back to the point before they ran the *Git pull -–rebase* command.

After Developer B is done synchronizing with the Central Repository, they can publish their changes successfully, using the following:





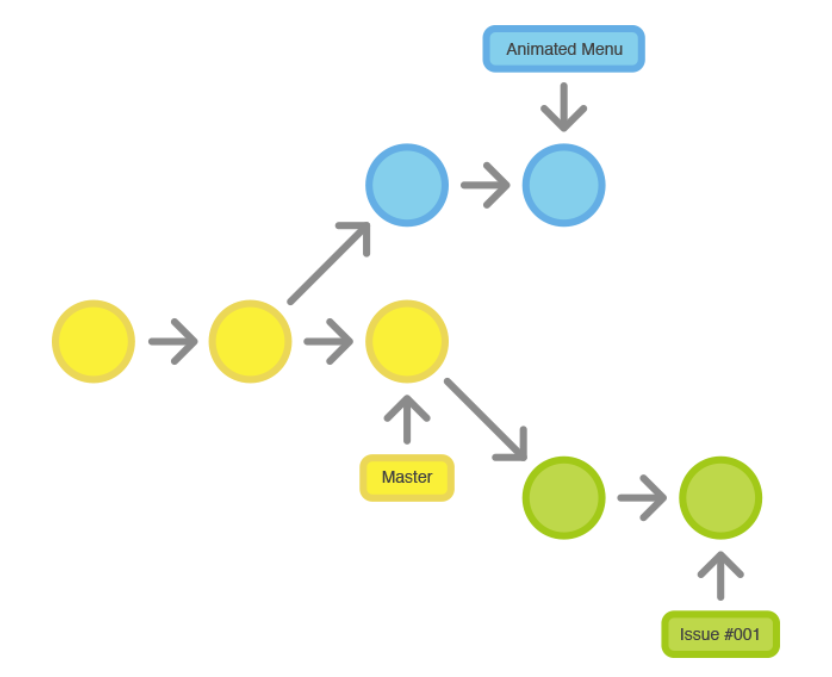
**Summary of Central Workflow:**

The Central Workflow is great for transitioning to git, but it does not leverage the distributed nature of git. If our team wants to streamline its collaboration efforts, we should explore the Feature Branch Workflow. Which, dedicates an isolated branch to every feature and allows for in-depth discussions around new additions before integrating them into the official project.

**Points to Discuss:**

* Is this option feasible for us, since we will be updating multiple codes (i.e LH 100, BeamBoss, LH 1gig)?
  + I assume
* How easy will it be for us to switch from the easier Central Workflow to a more complex Workflow when we increase the size of the team? (i.e how scalable is this solution?)
  + We would not want to implement a central workflow now if it means a substantial amount of work to scale it in the future, if that is the case we should select one of the more complex workflows from day one.
  + But if it is easy to transition to the more complex workflows, then it may be beneficial to first start with the Centralized Workflow and expand as we are ready.
* Atlassian website, makes it seem like it will be easy for us to start with a centralized workflow and then transition to a Feature Branch Workflow once we get the hang of Git.

**Feature Branch Workflow:**

****

The idea behind the Feature Branch Workflow is that all feature developments should take place in a dedicated branch instead of the *master* branch. Which makes it much easier for multiple developers to work on the particular feature without disturbing the main codebase. Also, this method means that the master branch will never contain broken code. Giving us an advantage if we are operating in a continuous integration development environment.

By encapsulating feature developments, it is possible to leverage the pull requests, which is a way to initiate discussions between developers regarding the branch. Giving other developers an opportunity to sign off before it is integrated into the official project. It is also possible to have other developers pull your work and help you, if you are stuck in the middle of a feature and need help. Making it incredibly easy for all developers to comment on one another’s work.

Like the Centralized Workflow, the Future Branch Workflow also makes use of a Central Repository, and *Master* still represents the official project history. But, developers will create new branches every time they wish to start work on a new feature under the Future Branch Workflow, instead of committing directly to the local master branch as they do under the Centralized Workflow. *Feature Branches* should have specific names, such as, *Alignment-Algorithm-Improvements* or issue-#7632. The idea behind this is to give a very clear and focused purpose to each branch.

Git does not make any technical distinctions between the master branch and the features branch, so developers can edit, stage, and commit changes to the feature branches just like they did under the centralized workflow.

*“In addition, feature branches can (and should) be pushed to the central repository. This makes it possible to share a feature with other developers without touching any official code. Since master is the only “special” branch, storing several feature branches on the central repository doesn’t pose any problems. Of course, this is also a convenient way to back up everybody’s local commits.” [[1]](#footnote-1)*

Once a developer completes a feature they don’t immediately merge it to *master*. Rather, they push the feature branch to the central server and file a pull request asking to merge their additions to the *master*. Giving the other developers on the team a chance to review the changes before they become part of the main codebase. Other developers are able to review the changes by issuing a pull request once the feature branch is available on the central server.

*“Code review is a major benefit of pull requests, but they’re actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they’ll be able to see the question right next to the relevant commits.”[[2]](#footnote-2)*

Once a pull request is accepted, publishing a feature is much the same as it is under a Centralized Workflow.

1. Make sure the local *master* is synchronized with the upstream *master*.
2. Then, merge the feature branch into *master* and push the updated *master* back to the Central Repository.

We may need to acquire 3rd party product repository management solutions to facilitate our pull requests. Atlassian mentions Bitbucket and Stash as potential solutions.

Process of starting a Feature Branch

Before the Developer begins a feature, they need an isolated branch to work on. They can request a new branch with the following command:

*Git checkout –b Alignment-feature master*

This will check out a branch called *Alignment-Feature* based on *master*, and the *–b* flag informs Git to create the branch if it does not already exist. On this branch the Developer edits, stages and commists changes in the usual fasion. Building up their feature with as many commits as necissary:

****

After the developer has added a few commits to their feature, it is a good idea to push the feature branch to the central repository. This will not only serve as a convenient backup, but it will also allow other team members access to the developers initial commits.

*Git push –u orgin Alignment-feature*

This command pushes *Alignment-Feature* to the central repository (*origin*), and the *–U* flag adds it as a remote tracking branch. After setting up the tracking branck (*-U*), the developer can call *git push* without any parameters to their push feature.

Once the Developer completes their feature they will need to file a pull request letting the rest of the team know that they are done. But first, they should make sure the Central Repository has their most recent commits:

*git push*

Then, they will file a pull reguest in their Git GUI asking to merge *Alignment-Feature* into *master*, and the rest of the team will be automatically notified. An advantage of pull requests, is that they will show comments right next to their related commits, so it is very easy for team members to ask questions about a specific changeset.

If the team members decide that changes need to be made, the developer will use the exact process they used in the first iteration of their feature. They edit, stage, commit, and push the updates to the Central Repository. With all of their activity showing up as a pull request, and other team members being able to make comments along the way.

If another team member wanted, they could also pull the *Alignment-Feature* to their local repository and work on it on their own. Any of the commits that they added would show up in the pull request.

Once the pull request for *Alignment-Feature* is accepted, someone will need to merge the feature into the stable project by completing the following comands:

*Git checkout master*

*Git pull*

*Git pull origin* *Alignment-Feature*

*Git push*

This process will often result in a merge commit, which is preferred by some developers but it is also possible to rebase the feature onto the tip of master before executing the merge, resulting in a more linear history. The process can be left to the developer teams choice.

Some GUI’s will run all of the above commands, in an automated *pull request* acceptance process if the developer simply clicks the accept button. If ours does not, it should at least be capable of automatically closing the *pull request* when the feature branch gets merged into *master*.

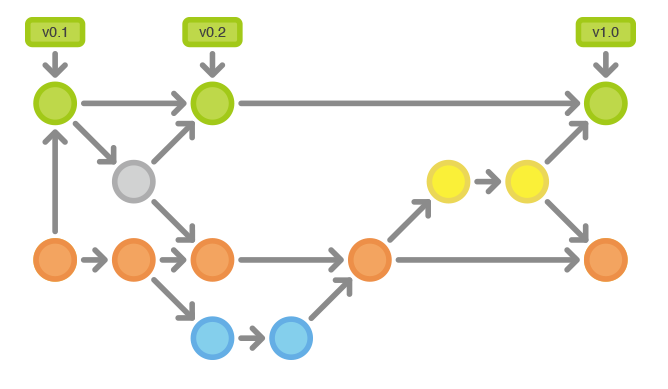
In addition to the aforementioned processes, we may also want to consider putting a specific developer in charge of final approval and merging each feature into the stable project. This is not a necessity, but it could give us more control over what features are being added to our stable project.

**Comments on the Feature Branch Workflow:**

The Feature Branch Workflow seems much more suited for Polewall’s needs then the Centralized Workflow, because it allows us to work on very specific features such as the Alignment-Algorithm. The Feature Branch Workflow becomes even more essential while working with partners like Lockless, because isolating features into separate branches allows everyone to work on independent features, simultaneously. While, also offering the ability for developers to share changes when needed.

*“The Feature Branch Workflow is an incredibly flexible way to develop a project. The problem is, sometimes it’s too flexible. For larger teams, it’s often beneficial to assign more specific roles to different branches. The Gitflow Workflow is a common pattern for managing feature development, release preparation, and maintenance.”*

**GitFlow Workflow:**

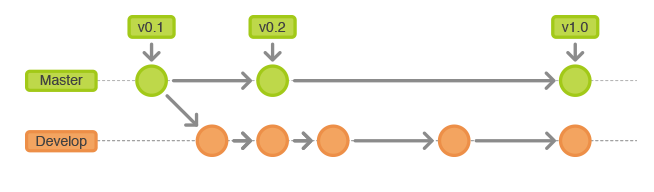
****

The Gitflow Workflow expands on the Feature Branch Workflow. It is considerably more complicated then the Feature Branch Workflow, but is also a more robust framework suited for larger projects.

Gitflow does not add any additional concepts or commands beyond what is required for the Feature Branch Workflow. Instead, it will assign very specific roles for different branches and define how and when they should interact. In addition to using feature branches, Gitflow will also use individual branches for prepping, Maintaining and recording releases.

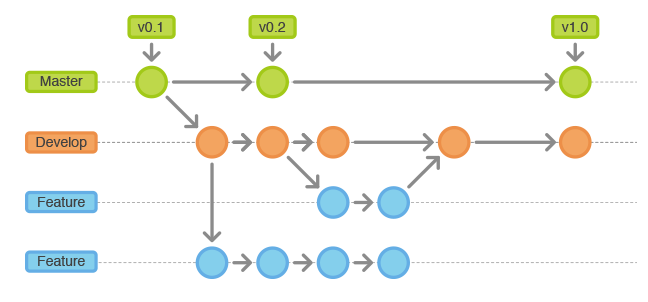
The Gitflow Workflow will work similar to the other workflows, with the only difference being the branch structures for the project.

Instead of only using the *Master* branch, Gitflow uses two branches to record the history of the project. The *Master* branch will be used to store the official release history of the project. While, the *develop* branch serves as an integration branch for features. Using this setup, it is convenient for developers to tag all commits in the *master* branch with version numbers. Please see below, for an example.

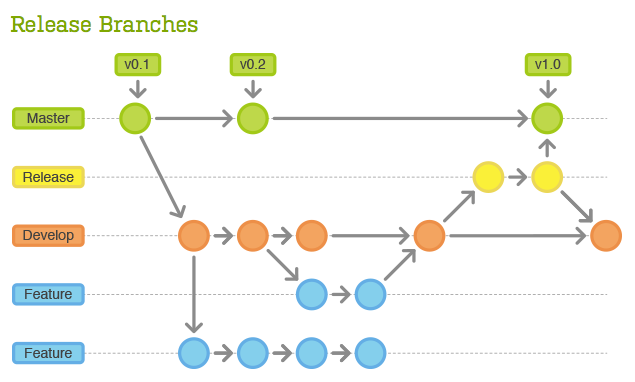


The rest of the Gitflow process resolves around the distinction between the *master* and *developer* branches.

Feature Branches will still reside in its own branch, but in Gitflow it will use the *developer* branch as its parent, rather then the *master* as it does in the Feature Branch Workflow. When a developer completes a feature branch it should be merged back into Develop. Under Gitflow, a *feature branch* should never directly interact with the *master branch*.



Note, the feature branch combined with the develop branch can be seen as the Feature Branch Workflow inside of the Gitflow Workflow.



Once develop branch has acquired enough features for a release, the developer creates a *Release* branch off of develop. Once the developer creates the Release branch they start the next release cycle, meaning they can no longer add feature after this point. Only bug fixes, documentation generation and other release-oriented tasks should be added to the release branch. Once the release is completed, it will be merged into the *master* branch and the developer should tag it with a version number. In addition, it should be merged back into develop, which may have progressed since the release was created.

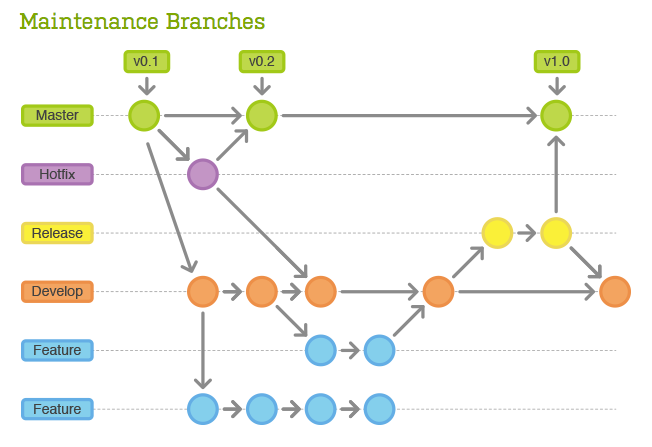
The Gitflow process makes it possible for one team to prepare releases, while the other team polishes the current release or works on features for the next release. It also does a good job of creating well-defined phases of development. The developers can easily say where they are at in the development process and have it shown in the structure of the repository.

Common Conventions in Gitflow:

Branch off: develop

Merge into: master

Naming convention: release-\* or release/\*

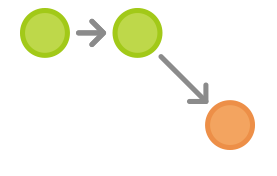


Maintenance, AKA hotfix, branches are used to quickly patch the production releases. These maintenance branches are the only ones that should fork of the master in the gitflow workflow. As soon as a developer completes a fix, it should be merged into both the *master* and the developer (or current release branch), with the *master* branch being tagged with an updated version number.

Having a dedicated line of development for bug fixes, allows our team to address issues without interrupting the rest of the workflow or waiting for the next release cycle.

**Example of Gitflow Process:**

Create a Developer branch:



A simple way to create the developer branch is to have a developer create the branch locally and then push it to the server:

*Git branch develop*

*Git push –u origin develop*

This branch will contain the complete history of the project, whereas the *master* branch will contain an abridged version.

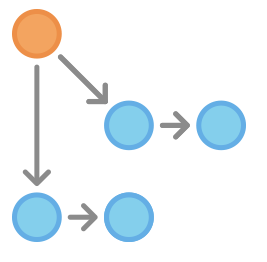
After the creating developer pushes the branch to the server, the other team members should clone the central repository and create a tracking branch for the development branch. This can be done as follows:

*Git clone ssh://user@host/path/to/repo.git*

*Git checkout –b develop origin/develop*

Everyone should now have a local copy of the historical branches.

Developer A and B begin new features



Both developer A & B will need to create separate branches for their respective features. They should both base their Feature Branches on Develop:

*Git checkout –b some-feature develop*

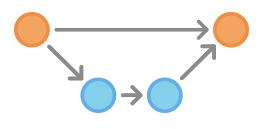
Both will add commits to the Feature Branch in the usual fashion: edit, stage, commit:

*Git status*

*Git add <some-file>*

*Git commit*

Developer A finishes his feature



Once Developer A has finished their feature, they should merge it into their local develop branch and push it to the central repository, like so:

*Git pull origin develop*

*Git checkout develop*

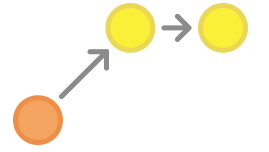
*Git merge some-feature*

*Git push*

*Git branch –d some-feature*

The first command will make sure that the develop branch is up to date before trying to merge the feature in. Note, features should always be merged to the develop branch under Gitflow, developers should never merge directly into master in this workflow.

Developer A prepares a release:



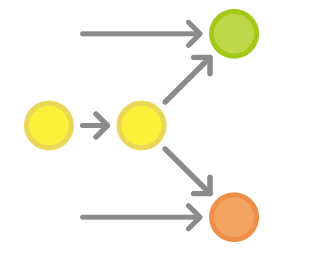
Like Feature development, the developer will use a new branch to encapsulate the release preparations. It is during this process, that the release’s version number is also established:

*Git checkout –b release-0.1 develop*

This branch will be used for everything related to this release, testing, document updates, and all other preparations that involve the release.

Once the developer creates this branch and pushes it to the central repository, the release becomes feature-frozen. Any functionality that is not already in the *develop branch* is postponed until the next release cycle.

Developer A finishes the release



When the release is ready to ship, the developer merges the release into the *master* and *develop branches*, then deletes the release branch. It is very important to merge back into *develop* because critical updates may have been added to the *release branch* and they will need to be accessible to new features. If other team members want to review the code, this would be the ideal place for a *pull request*.

*Git checkout master*

*Git merge release-0.1*

*Git push*

*Git checkout develop*

*Git merge release-0.1*

*Git push*

*Git branch –d release-0.1*

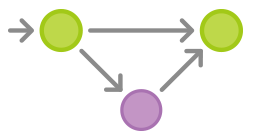
The release branches act as a buffer between feature development (*develop branch*) and public releases (*master branch*). Whenever a developer merges something to master, they should tag the commit for easy reference:

*Git tag –a 0.1 –m “Initial public release” master*

*Git push --tags*

Git comes with several hooks that are scripts, which execute whenever a particular event occurs within the repository. It is possible to automatically configure a hook to build a public release whenever you push the *master branch* to the central repository or push a tag.

Bug Is Discovered:



If bug is discovered after shipping the release, a developer will create a maintenance branch off of the master and fix the issue with as many commits as necessary. Then they will merge it directly back into *master*.

*Git checkout –b issue-#001 master*

*# fix the bug*

*git checkout master*

*git merge issue-#001*

*git push*

Maintenance, like release, branches contain important updates that need to be included in develop, so the developer needs to perform that merge as well. Afterwards, they are free to delete the branch.



**Comments on Gitflow Workflow:**

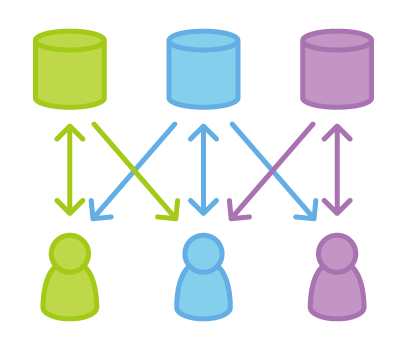
The Gitflow Workflow is more robust then the Feature Branch method, and is better suited for managing larger projects. We may decide that it is overkill for our small team, and can cause an increased burden on our developer’s workload. It may be a better idea to start with a Feature Branch workflow, and scale to a GitFlow workflow when the team becomes larger.

Overall, I feel that Gitflow will give us the best structure and documentation for showing potential investors during a due diligence. However, I am worried that it requires too much time of our developers and does not fit our current development needs for flexibility. We will need to way our options of structure vs. flexibility and make the decision on workflow that will best suit our needs both now and in the near future.

It is not necessary to take all aspects of one workflow, it is possible to mix and match to best fit our needs. This could be an option for us, where we develop a new workflow that is a hybrid of the Feature Branch Workflow and the Gitflow workflow.

**Forking Workflow:**

This Workflow will put one developer in charge of pushing all code to the single central repository. This will allow us the most control over our main code, but will also place a time burden on either Jan or Adnan. This workflow is recommended in cases where we use an untrusted third party to develop code for us. The forking workflow will mean that each contributor will have not one, but two Git repositories: a private local one and a public server-side one.

****

The Forking Workflows main advantage is that contributions can be integrated without the need for everyone to push to a single central repository. All developers will push to their own server-side repositories, while only the project maintainer can push to the official repository. This method will allow our project maintainer to accept commits form all the developers, without having to give them write access to the official codebase.

This results in a distributed workflow that provides a flexible way for us to work with 3rd parties and collaborate in a secure manner. We may not deem the Forking workflow to be necessary when working with a close partner, such as Lockless, but we should consider moving to a forking method if we work with a less trusted partner in the future.

**Forking Workflow Process:**

As with the other workflows, the Forking Workflow will begin with storing the official public repository on a server. But when a new developer wants to start working on the project, they will no longer directly clone the official repository.

Rather, they fork the official repository to create a copy of it on the server. The new copy will serve as their personal public repository. No other developers will be able to push to the personal repository, but they can pull changes from it. After they have created a server side copy, they will perform a *git clone* to get a local copy, just like in the other workflows.

When a developer is ready to publish a local commit they will push to their own public repository-Not the official one. Then the developer will file a *pull request* with the main repository, letting the project maintainer know that an update is ready to be integrated. The pull request can also serve as a convenient discussion thread, if there are any issues with the contributed code.

When the project maintainer is ready, they will pull the contributors changes into their local repository, check to make sure it does not break the project, and then proceeds to merge it into his local master branch, and then they will push to the official repository on the server. After it is pushed to the official server, the contribution becomes part of the project and other developers should pull from the official repository to synchronize their local repositories.

All of the developers’ personal public repositories are just a convenient way to share branches with other developers. All developers should still use branches to isolate individual features, like they do in the Feature Branch and Gitflow workflows. The only difference will be how those branches get shared. In the Forking Workflow, they are pulled into another developer’s local repository, while in Feature Branch and Gitflow workflows they are pushed to the local repository.

**Comments on Forking Workflow:**

We may want to implement some form of the forking workflow immediately, we do not need to follow the process exactly, but should consider having either Adnan or Jan in the role of Project maintainer when working with third parties. I believe this could help us avoid potential headaches in the future, of a 3rd party loading a code with errors that affects our main code. If Adnan or Jan acts as the project maintainer we can have a review of all code before it is merged with the official repository.

It may be overkill to follow the Forking Workflow completely, as it will add to the project maintainer’s workload. So we may want to consider having a project maintainer, but then following the Feature Branch Workflow, or the Gitflow Workflow. With the Project Maintainer merging 3rd part code as a feature, after they have approved it.

It could also be possible to make all outside developers have their own public repositories to work on our code, while the in house developers work directly with the official repository as in the other workflows. The advantage of having 3rd parties keep a public repository on our servers is that we can easily access and review their work during the development process.

**Discuss GIT GUI’s:**

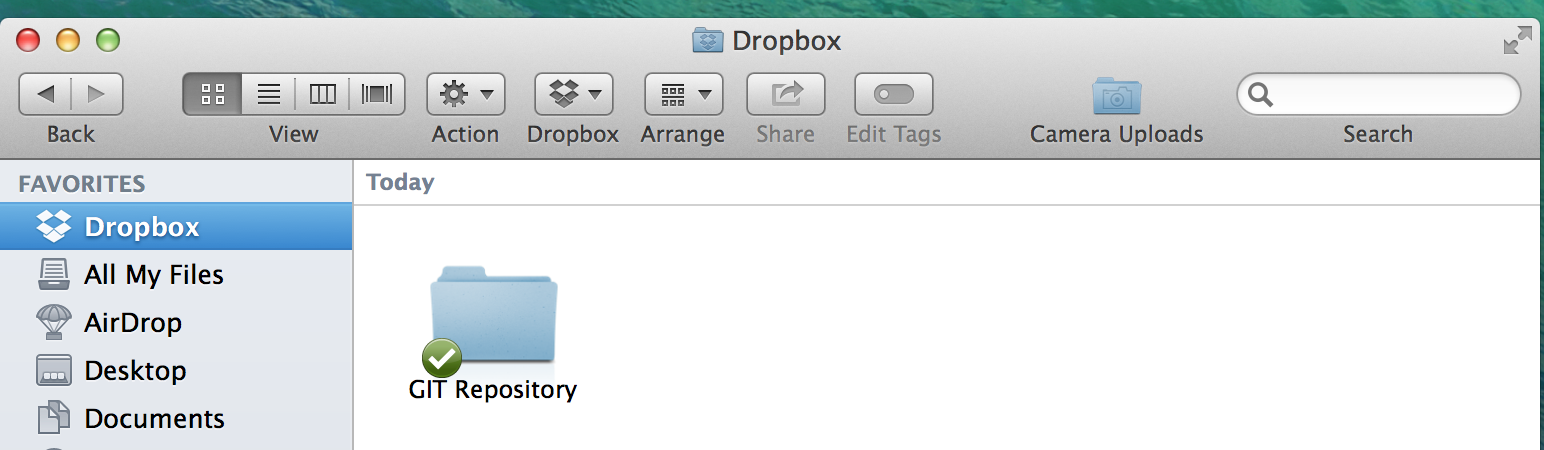
There are many options for Git GUI’s to make the git process easier on the developers. We will need to evaluate the Git GUI’s available, and determine if we should use the same one for all developer or allow each developer to select the one they prefer.

**Setting Up Git on Dropbox server:**

It is possible to create a bare git repository in a special Dropbox folder. If successful, this would be both the easiest and cost effective way for us to store our Git Repository.

A “bare” Got repository will not actually have any repository files shown. Instead, it just contains the guts and mechanics of the repository, the stuff that usually resides in the *.git* directory. This is the same type of repository that Github uses and is only meant to be a place for developers to push and pull from, it is not for doing any direct file edits or commits from.

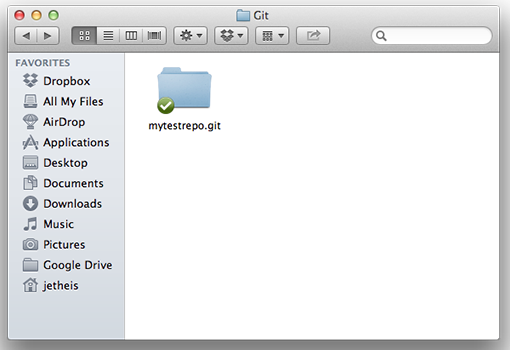
**Step 1**: Create a new folder for all of your repositories to live. (I have already done this, under the Polewall Dropbox folder GIT.



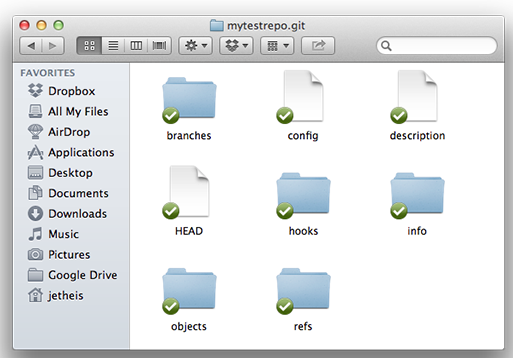
**Step 2:** in Terminal, cd into the folder and create a bare repository:



**Step 3:** Check To make sure you see a new folder inside your Git Dropbox folder; with the title you gave it.



**Step 4:** Make sure that inside the new folder, you can see the guts of the repo:



That is all we really need to do to get the bare repository set up. It is now a good idea to try to use it, so insure that it works correctly.



**Notes on Dropbox as repository:**

Many guides have said that it is a simple process to create a bare repository in Dropbox and use it as the companies Git Official Repository. While I believe this to be true, I attempted the process and failed. I do not think that it was the processes fault, but more a reflection on my lack of understanding of how to properly code. I believe the issue was with the installation of git itself on my mac, which then caused Terminal not to recognize the Git commands. In effect, I still think it may be worth wild to have Adnan or Jan attempt this process to evaluate its feasibility. Most tutorials said it would only take an experience coder 5 minutes to complete the process.

If we decide to move forward with Dropbox as our Git Server, we should set up a trial period to test its validity. Many people said they have used Dropbox with zero issues, but I have read some reports that said Dropbox messed up their repository. It is difficult for me to tell at this time if this was a result of Dropbox being a poor solution as a Git Repository or if the failure was a result of the user at the time.

I have emailed Dropbox and asked them if they have an official stance on using their folders to store our Git Repository. They have informed me that it is perfectly fine to store a local Git Repository in a Dropbox folder, but recommended we us a separate server for our public hub.

**Action Points to Discuss:**

* Which Workflow do we want to follow?
  + Do we want to follow it exactly, or create a hybrid workflow that incorporates some of the processes from the other workflows?
* Which server should we use?
* Do we want to have a Project Maintainer, who will review and merge all the code from third parties?
  + Who should be the Project Maintainer?
  + Will they only be responsible for reviewing and merging code from third parties? Or do we want them to also be responsible for reviewing and merging all of our in-house code too?
  + Do we want third parties to still set up a Public repository on our server so we can access their code? Or do we want them to just deliver us the code in the same manner that they currently use?
* What, if any, Git GUI should we use?
* Do we want to have a standard GUI for all developers? Or should each developer have their own option for selecting their GUI?

1. [https://www.atlassian.com/git/workflows - !workflow-feature-branch](https://www.atlassian.com/git/workflows#!workflow-feature-branch) [↑](#footnote-ref-1)
2. [https://www.atlassian.com/git/workflows - !workflow-feature-branch](https://www.atlassian.com/git/workflows#!workflow-feature-branch) [↑](#footnote-ref-2)