**Overview**

**Goal of the course**

By the end of this course, you will be able to know industry standards for team collaboration using Git and version control system designs, to apply continuous integration, as well as setting up test automation using Github Actions and Jenkins.

**Pre-requisites**

This course assumes you know:

* Basic Git
* Node with Express
* TDD
* MVC
* Docker

**What you'll learn in the course**

In this chapter, you'll learn to connect the following:

1. Advance Git commands
2. Centralized and Distributed Version Control System
3. GitHub
4. Push and Pull-request

<https://youtu.be/GUpYeDwOTOI>

#### Objectives:

* To review basic git commands
* To emphasize its benefits and proper usage

# Git (Revisited)

From this point, we know that *Version Control System* allows us to monitor set of files over period of time. It is a system that allows us to revert files back to a previous state, review changes made over time, see who last modified something, etc.

Git is a Version Control System, along with its competitors SVN and Mercurial. Git is best VCS to learn because of its popularity, efficiency, and branching features.

To refresh basics of git commands, we **strongly suggest** that you should revisit our Basic Git course in this link before you dive into next page:

<http://learn.village88.com/m/34/961/4543>

Just to test…

Can you figure out what’s missing in the picture? I bet you can if you know the basic git commands. ;)



Good luck! Have fun to the course and prepare to get the best learning!

#### Objectives:

* Learn branching in Git
* Know the purpose of branching
* Learn the git commands and flags for branching

# Git Branching

<https://youtu.be/Kf0NZDrp-ew>

#### Objectives:

* Learn branching in Git
* Know the purpose of branching
* Learn the git commands and flags for branching
* Apply simple commit and branch commands.

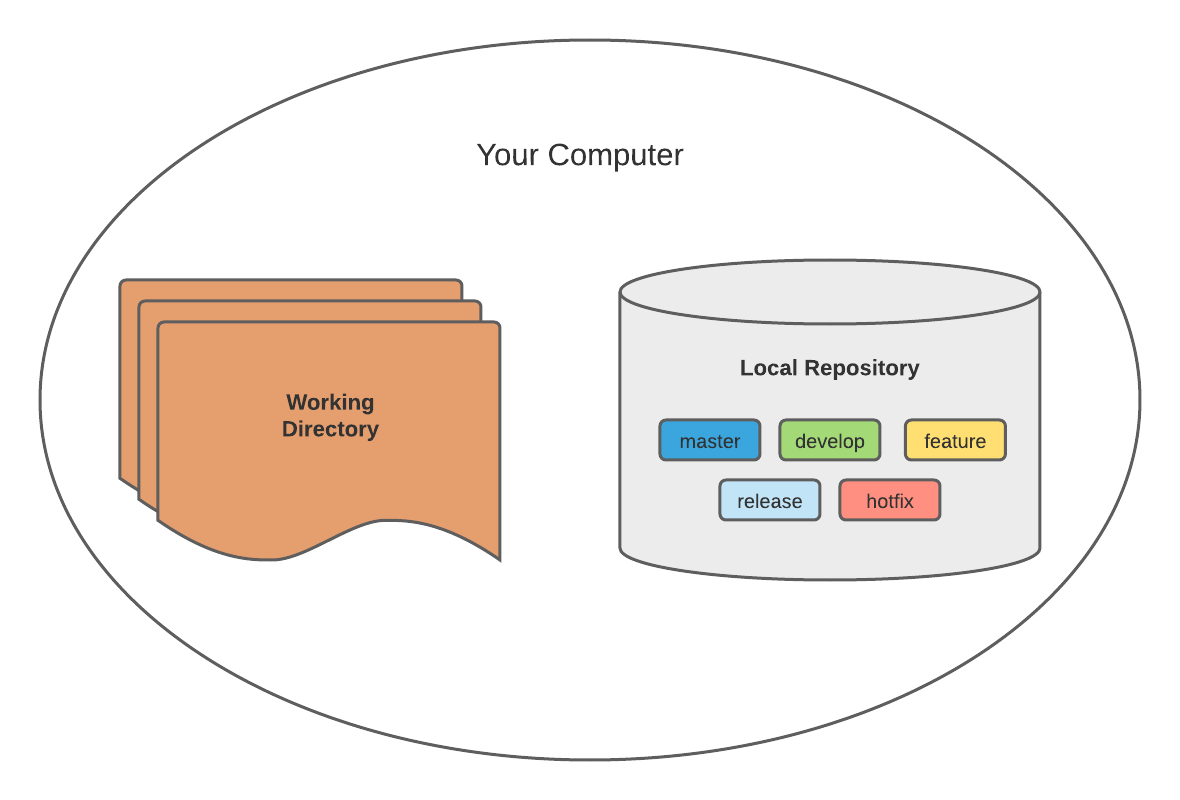
# Git Ready

Before we involve GitHub (a hosting platform that helps our codes viewable to cloud), we want you to be comfortable first about the concept of branching.

Like as we know, local repository resides in our computer. We have also working directory (can be folder inside your C: drive). The role of “branch” is to contain the versioned codes. Let’s say, in our working directory, we want to get the contents from branch “A”.

For the meantime, we will focus how to switch in different branch contents in our working directory.

Let’s see below diagram for visualization:



In this assignment, we will create, delete and list branches, as well as practice switching codebase and do simple commit.

Do you still remember the basic git commands?

To-do:

1. Initialize git
2. Create branch ‘develop’
3. Switch to ‘develop’
4. Create branch ‘feature’
5. List all branches
6. Rename ‘feature’ to ‘feature/abc’
7. List all branches
8. Remove branch ‘feature/abc’
9. List all branches
10. Create branch and switch to 'release'
11. List all branches

Attach your screenshots showing the commands you entered (in sequence).

For the final list, you should have the ff. branches:

* develop
* release

Good luck and have fun experimenting!

START WORKING ON THIS

#### Objectives:

* To debug an application
* To apply git branching
* To review in TDD
* To learn how to save changes in local repository

# Git Out, Bug!

Remember the role of “branch”? The branch’s role is to contain the *versioned* codes. Let’s say, in our working directory, we want to get the contents from branch “A”. Afterwards, we made few changes inside it (ex. write additional codes, removed a file, etc.) and then committed. When we go for branch “B”, we’ll notice in our working directory that our changes earlier are not visible now. This is due to the fact that branch “B” has different content to branch “A”, and whatever you modify is will reflect only to that specific branch. Makes sense, huh?

In this challenge, we will also incorporate TDD. By applying what you learned, you’ll need to fix the source code to make success unit tests. Hopefully the codes can help you to absorb the logic of git and simulate how commands are used to be.

**To-do:**

1. Create a new working directory and initialize git.
2. Get the contents from <https://github.com/hh-kigcasan/git-commands/> and save it to "master" branch.
3. Create and switch to branch “bugfix/git\_status“.
4. Run unit testing and check failed tests.
5. Supply the appropriate fix for the failed model*.*
6. Once the unit tests passed, commit your changes to your local repository.
7. Switch to “master” branch. (Can you see your fix there?)

*Note:*

* *For #2, use a git clone to download the contents.*
* For #4, make sure you install the dependencies first in your working directory
* *Naming convention is so important, not only in codes, but also in naming branches. This makes our branch history very beautiful, yet meaningful. In this case, we want to tell that “Hey! I fixed a bug about git status feature!” -- in indirect way. So, from now on, whenever you’re fixing a bug, do also the same format in naming a branch: bugfix/<feature or short description or bug issue id given by your team>.*

Good luck!

#### Objectives:

* To learn version control system
* To know the approach in centralized version control system
* To be aware on advantage and disadvantage of centralized version control system

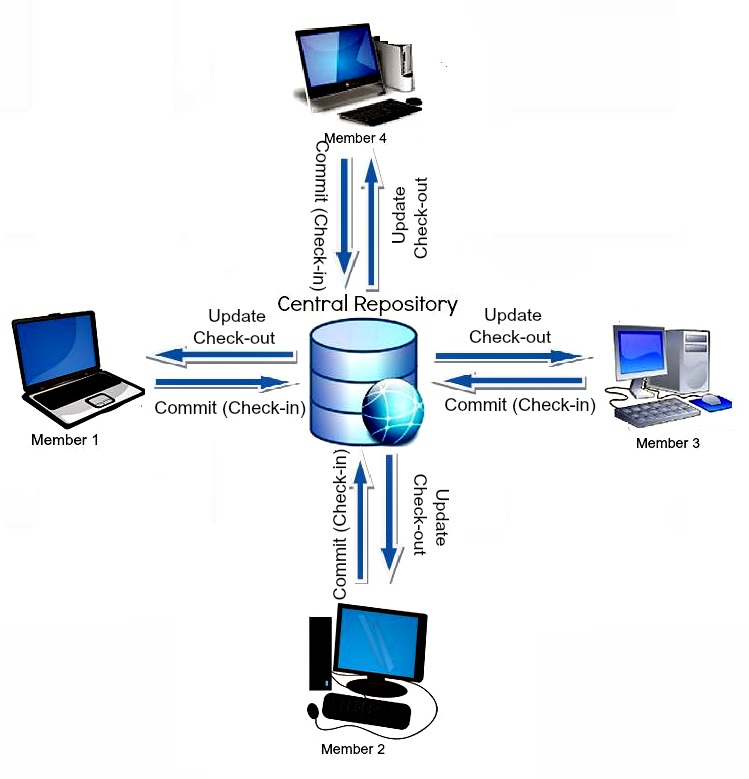
# Centralized VCS

Based on our previous discussion about “Git”, we learned that it is a Version Control System. But wait, let’s discuss first “What is Version Control System”.

Imagine you are working with a team in parallel. Meaning, not only you are working for the shared codebase. It's so time-consuming to notify co-developers every changes. Besides, you may not know that it can cause inconsistency on team’s code. Who knows? Maybe you did not apply the recent changes from co-developer, and you accidentally overwritten it! Oh no! How will the team track the changes anyway?

This is why *version control* was designed. Aside from these, team members can review codes from other members in organized way! It is what we called “pull-request” – *requesting* that a specific branch will *pull*or get your code changes. In pull-request, code reviewers can see and decide if it is acceptable, or need some improvement before making changes to destination branch.

Let’s have a look on one of the version control system design:



Can you imagine your team working in this flow?

As we can see, it is called “centralized” since there are many working copies that direct to only 1 repository. So simple, right?

Centralized version control systems are based on the idea that programmers will “commit” (apply) their changes to this central copy so others can see and update their own working copy.

This kind of VCS is popularized by the Subversion tool (SVN) – an opponent of Git. This provides easy approach in team collaboration, as well as in management because it allows administrators to fine-grained control over the central repository.

## How about the downside?

Well, the most obvious is the single point of failure that when the *centralized repository* server goes down during collaboration, saving versioned changes to there *will* *not be possible*. Your team cannot see now the most recent codebase.

Once your computer lags and your code changes were not saved, what will you do to prepare for this scenario? Will you commit your changes beforehand to be safe? Notice that in this design, “commit” is direct to the central repository. Dude, “commit” is sacred, you know!? Team wants to see final commit for your specific one task because team maintains the cleanliness of repository history. If that’s the case, maybe your favorite IDE can save you - if only has an autosave feature.

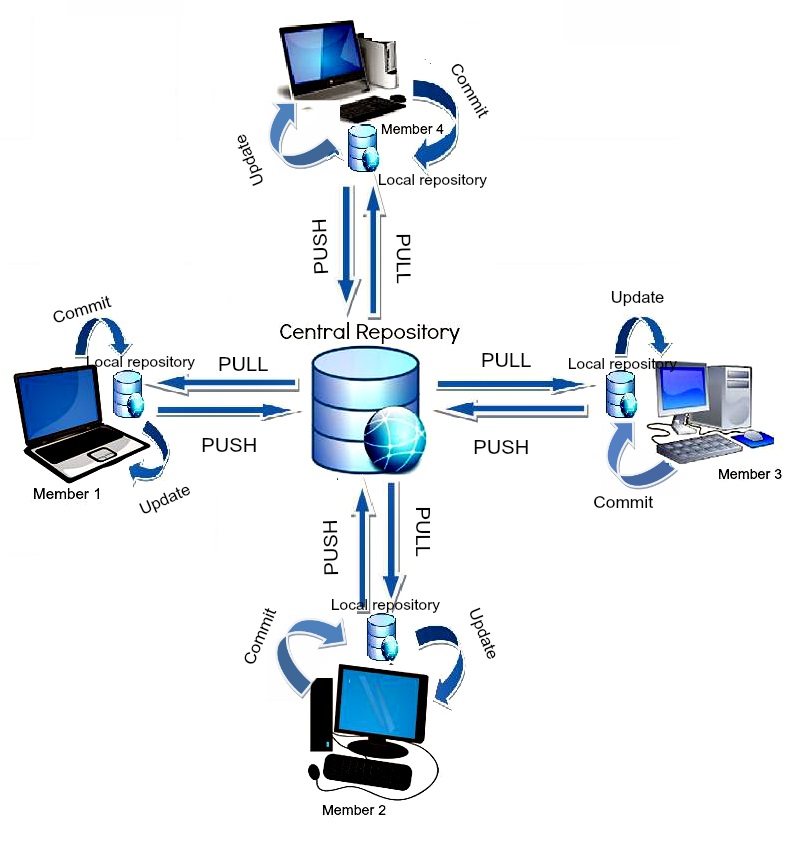
If so, do you know what’s the best alternative solution that most of the industries used up to now? We’ll find out next!

#### Objectives:

* To know the approach in distributed version control system
* To know the benefits of distributed version control system
* To compare two version control systems

# Distributed VCS

In this lesson, you will know the approach in another type of version control system design which Git belongs.



The idea of **Distributed *Version Control*** is only having own repository aside from the central repository. That's it!

Observe how the central repository is directed next to local repository, then goes down to your working directory. In **Distributed Version Control**, each developer has their own repository and working copy. Just committing your changes will not give others access to your changes -- until you are ready to push it in central repository!

Unlike ***Centralized Version Control***, having own repository in your computer is very important as it will serve as full backup of central repository in case server goes down. Also, having repository inside your computer makes confidence that you can play around in your working copy — knowing that you can go back to any save point in your local repository.

However, you might find it not so simple as Centralized Version Control, but I bet you’ll focus on greater benefits rather than the simpler with less features… am I right!?

For this course, we will focus only in **Distributed Version Control System** since it is what industry really uses now. Just keep your knowledge about Centralized Version Control and who knows, when you apply for work, you’ll be asked by interviewer about version control systems (since Git is in high demand and Git is a version control system too).

Have a good luck in your career!

# Objectives:

* To learn about pull-requests
* To learn what is merge conflict
* To know how to resolve merge conflicts

# Working with GitHub PR

Now that you learned about forking, pull-request, and merging, we’ll discuss where to host the central, personal (forked) repositories in cloud!

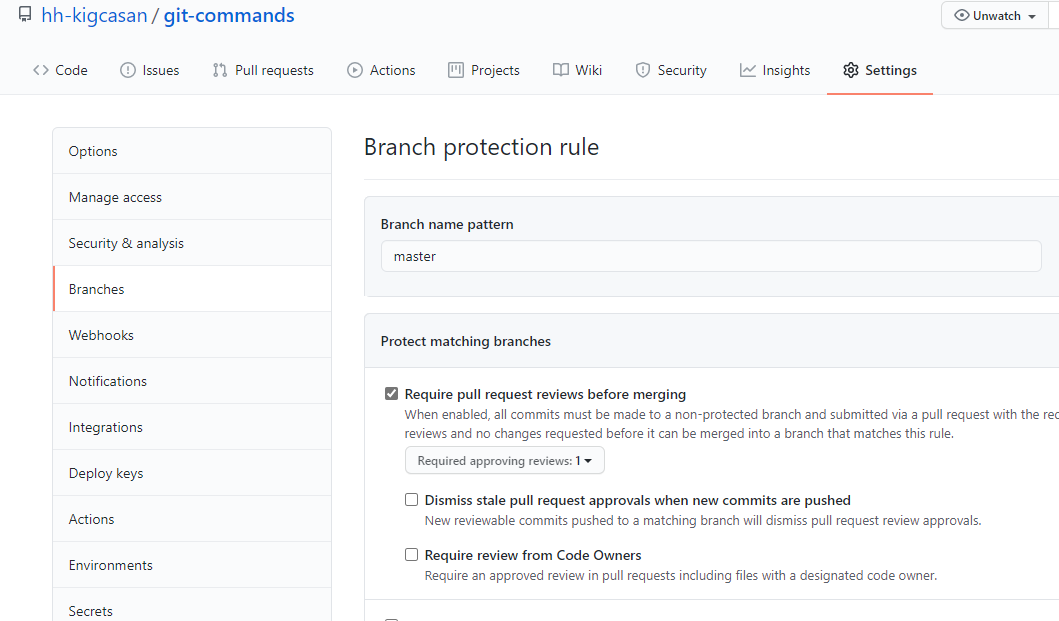
**GitHub**is a provider of Internet hosting for software development and version control using Git. Its primary goal is to offer free public repositories in cloud. GitHub is so popular and powerful not only because of this, but it also has easy support in Continuous Integration (which we will tackle in the next chapter).

If it’s your first time to know GitHub, or you have little background to it, don’t worry! Just the essential settings are good enough which we will discuss today.

## Branch Settings

When you log in to the [GitHub](https://github.com/) website, there’s a button that lets you create a repository. For the meantime, try to explore the *Settings*, specifically in *Manage access*, and *Branches*.

The ***Manage access*** is used for adding/removing collaborators for your specific repository. This is an important configuration especially when you want to give update privileges to specific users. As an admin, you still have the power to control the incoming updates to your repository (read about ***Branch Protection Rule***).

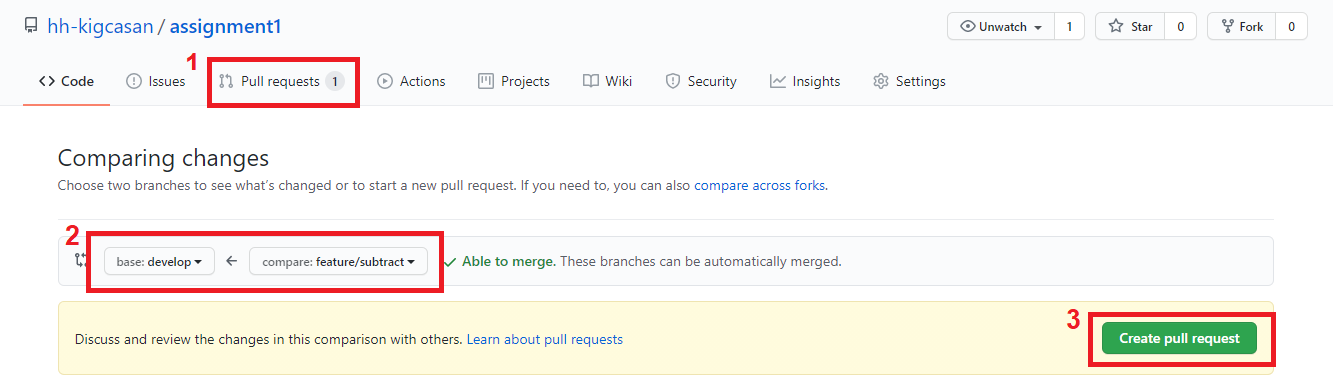


Note: Just to let you know, we are allowed to create multiple branch protection rules. That is, if you want different settings for specific branch, you can have separate branch protection rule!

In the industry, you may find that branches named “master” and “develop” are not allowed to be changed directly using push command. Let’s stick to the rule that if developer wants to push changes to either of these 2 branches, a pull-request should be submitted first. Maybe you’ll think why we should impose it. For now, you don’t need to deep dive about the *Git Flow* concept, but remember that a “master” branch is used in live production (too dangerous to modify!), and “develop” branch is the codebase used by all developers in your team (not only you and also not your own!) that’s why we should ALWAYS configure them at least with above settings.

## Merging

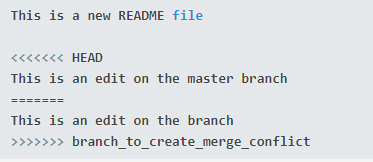
When submitting a pull-request, you will find it easy in navigating GitHub. See below image for your reference:



To add, when you’re finally in the industry, most likely you will encounter conflicts in merging codes. This usually occurs when there is another developer (not only you) has open pull-request at the same time, proposing changes with same file, same line. To address this, only one of you can be approved then the other should face merge conflict.

Learn how to resolve conflicts by knowing first the generated symbols.

For example, you have a README file:



* The "less than" characters denote the current branch's edits (in this case, "HEAD," which is another word for current destination branch's recent commit), and the equal signs denote the end of the first section.
* The second section is where the edits are from the attempted merge; it starts with the equal signs and ends with the "greater than" signs.

When doing this, if necessary, you need to communicate with the developer you had conflict with to discuss the impact of each other's fix.

Resource:

<https://www.youtube.com/watch?v=JtIX3HJKwfo>

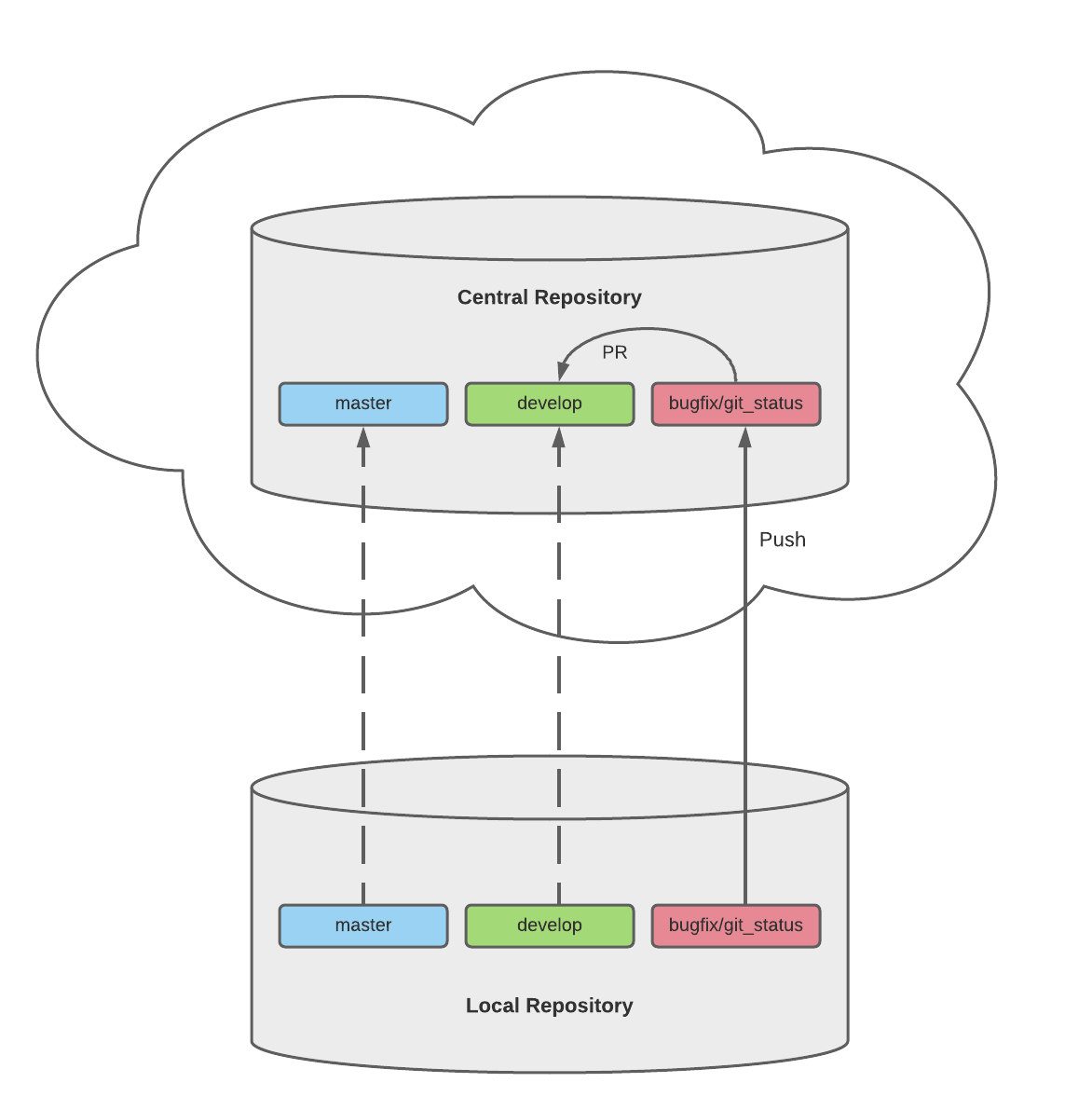
Objectives:

* To apply git commands in bash
* To create repository in cloud
* To practice pull-request from one user (for the meantime)
* To know how to configure repository settings

**Pull Request 1**

For this simple assignment, you will be required to create a central repository in GitHub. Remember your local repository in assignment #2?

Refer to the diagram below:



Since you already fixed the bug in your local repository, all you need to do is to move your codes from computer up to the cloud.

To-do:

1. From local *master*branch, create a new branch called d*evelop.*

2. Create an empty repository in GitHub. This will serve as your central repository.

3. Push your codes from local *master* to remote central *master.*Do this also to *develop*and *bugfix/git\_status*, respectively.

4. Configure your repository settings so that *master* and *develop*branches will not allow direct push.

5. Create a simple pull-request in GitHub from *bugfix/git\_status* to *develop* branch, and approve your own pull-request.

*Note: For the meantime, you are not required to have 2 GitHub accounts. Make sure that you configured your central repository to allow pull-request.*

Good luck and don't forget to upload your screenshots of git commands and pull request.

START WORKING ON THIS

#### Objectives:

* To use GitHub in simulating pull request as one user
* To simulate how conflicts are resolved

# Pull Request 2

Now that you learned how to perform git commands and pull-request, we will now do another practice.

In this assignment, you will be required to create new GitHub repository. Follow the instructions below:

**Pre-requisites:**

* Should have GitHub central repository. Clone it from <https://github.com/hh-kigcasan/git-commands-v2.git>.
* Should configured pull-request restriction in master and develop branches in your own GitHub central repository.

**To-do:**

Observe the differences of each branches, especially the feature branches. Your task is to make the 2 feature branches reflect to the develop branch using pull-request. Here are the guidelines you can follow:

1. In GitHub, open a pull-request from feature/git\_add\_all to develop.
2. Approve your own pull-request.
3. Do the same above steps for feature/git\_add\_asterisk.

All of these steps should be done only inside your created central repository. At the end of the day, make sure the codes are successfully combined to the central destination branch.

Good luck! Don't forget to upload necessary screenshots related to each requirement.

*Note: Naming convention is so important, not only in codes, but also in naming branches. This makes our branch history very beautiful, yet meaningful. In this case, we want to tell that “Hey! I made a new feature!” -- in indirect way. So, from now on, whenever you’re making new feature, do also the same format in naming a branch: feature/<feature name or id given by your team>.*

#### Objectives:

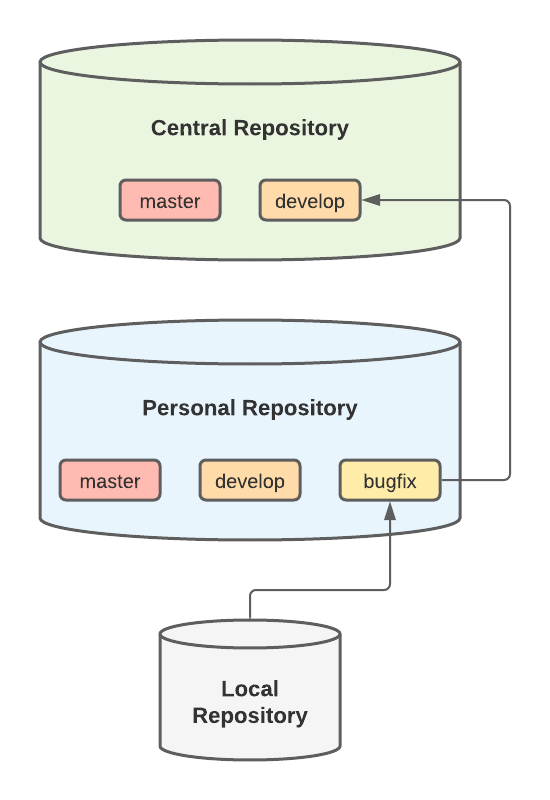
* To apply git commands in bash
* To setup personal repository using fork
* To create pull-request from personal to central repositories

# Pull Request 3

Now, we will introduce to you personal (forked) repository. As the name itself, it is used to have own full copy of central repository that serves as backup. This is a good feature in GitHub, so go and give it a try!

Imagine your team uses Distributed VCS approach and the team’s central repository is: [https://github.com/hh-kigcasan/survey-form-v1](https://github.com/hh-kigcasan/assignment2). All team members are required to have personal repository in GitHub.

Refer to the diagram below:



As you’ll notice, project’s unit tests are broken.

Supply the logic in model file to make the unit tests succeed. Remember to do local testing BEFORE introducing code changes through pull-request.

Once you’re done, send a merge request from your personal "bugfix/<title>" branch to central "develop".

Good luck! Attach screenshots of a merge request and your proposed code to complete this task.

START WORKING ON THIS