Week 10: Git II

Introduction

This week, we will look further into git and how this is used. As you already know from our previous session on git (week 4), git is an open-source version control system that is widely used by programmers.

This session, will start with a recap of the basics and then move into some more interesting ways git can be used when working as part of a team.

Before we begin, make sure you have the latest version of git (feel free to skip this if you already have git installed).

Task 1: Git Basics

The following are the basic git commands that you should be aware from our last session:

1. Start:
    - `clone`: Clone a remote repository
    - `init`: Create a new local repository
2. Main Workflow:
    - `pull`: Get and integrate the changes from a remote repository
    - `add`: Take a snapshot of selected (or all) files
    - `commit`: Record changes to the local repository
    - `push`: Update the remote repository with the changes in your local repository

TO DO: If you are unfamiliar with git or the above commands, then I recommend that you spent the beginning of the session completing this tutorial. Otherwise if you feel confident with git, please continue with the following tasks.

Task 2: Additional Commands

The following are some additional commands that are extremely useful:

- `status`: Show a summary of the current situation (see what you are doing)
- `log`: View history of changes (see what happened)
- `branch`: List, create, or delete branches
- `merge`: Join two or more branches together
- `switch`: Create or switch branches
- `checkout`: Can do the same things as `switch`, but can also be used to check previous commits

The above commands are just the tip of the iceberg, git provides us with a lot more functionality. To check all available commands use `git help -a`. Additionally, to learn more about a particular command and see all available options you can use `git [command] -h`. Finally, to see a list of official guides, use `git help -g`.

The following recourses are also useful:

- Git Visual Cheatsheet.
- GitHub Cheatsheet

Task 3: Basic Workflow

In this task, we will simulate a basic git workflow between two programmers to remind ourselves of the basic git usage. I have designed the workshop in a way that can be completed by a single person, but if you prefer, you can **pair up with the person next to you to complete the following tasks**.

TO DO:

1. First create a new remote repository (feel free to use any platform, e.g. GitHub). Add a readme file to your repository.
2. Then clone this repository twice in two different folders. We will use one folder as the workspace of team member A and the other folder as the workspace of team member B.
3. Add a line (something like "added by A") to the readme file from A's workspace, and then publish this to the remote repository.
4. Then move to B's workspace, make sure you are up-to-date with the remote repository and add a similar line to the readme file. Again publish your changes to the remote repository.

Task 4: Resolve Conflicts

The previous example is a bit unrealistic. Usually teams don't work in sequence but work in parallel. Unfortunately, this sometimes produces conflicts.

TO DO:

1. Let's start with a simple one. Pull from A and/or B to make sure both A's and B's workspaces are looking the same. Then create a file called "A.txt" on A's workspace and a file called "B.txt" on B's workspace at the same time (i.e. if you are simulating A first and then B, don't pull before creating the file for B). Then as A, update the remote repository. Afterwards, try to do the same as B. Fix the issues that arise using a merge.
2. Check the timeline of your commits to visually verify what happened using:

   ```
   git log --pretty=%s --graph
   ```

   or online (if you are using GitHub, check under Insights then Network).
3. Again this is not 100% realistic as A and B were working on separated files. Let's try to resolve the conflict when the users change the same file. First make sure both workspaces look the same. Then create a new file in the repository called "AB.txt". It doesn't matter which user does that, just make sure that the file is in both workspaces (e.g. create it using A, then publish it, and pull using B). Then perform a similar procedure as above but this time both A and B add a line to the file "AB.txt". How can you merge in this case?
4. Check again the timeline of your commits to visually verify what happened.

Bonus Task 1: Git Branching Strategies

Collaborative programming is hard, and over the years a number of strategies have been developed to help programmers deal with it. These are usually called branching strategies or flows. We will take a look at three popular approaches but many more exist. The important thing to note, is that you shouldn't adopt a branching strategy just for the sake of it but mold one to fit the needs of your team or project.

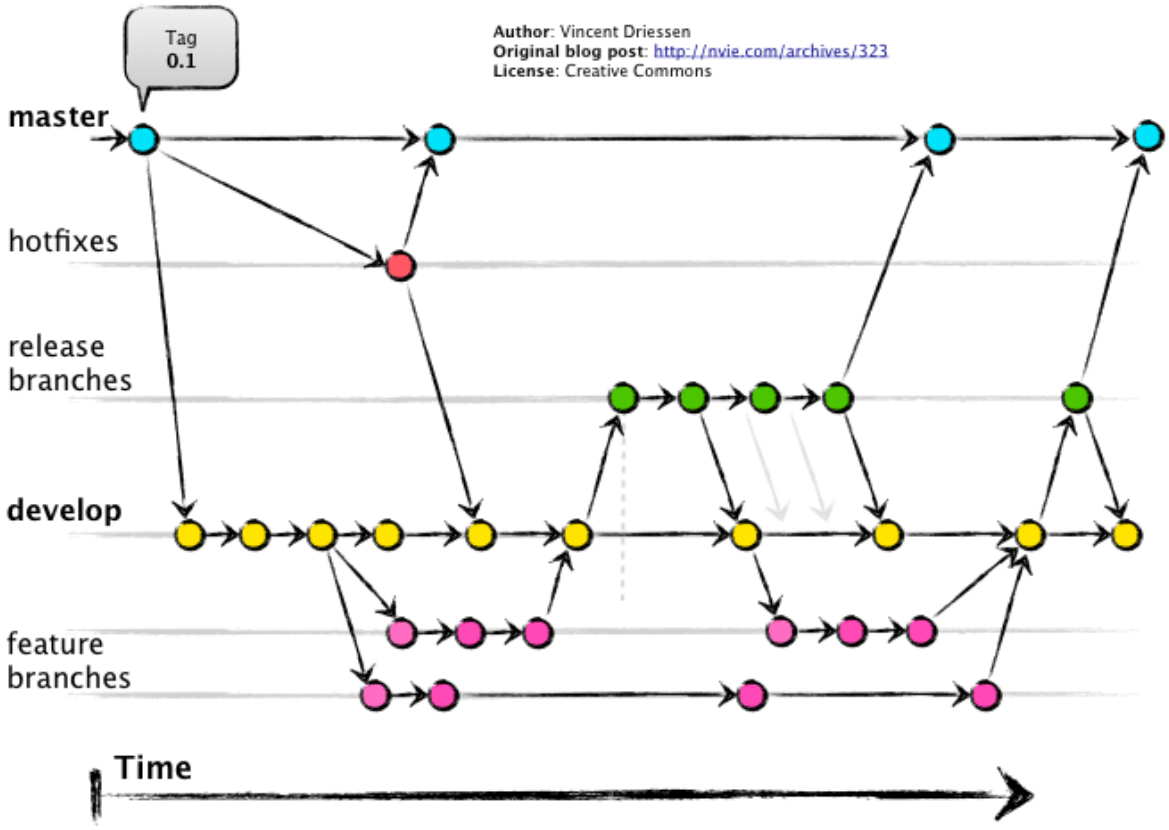
Git Flow:

This is the original branching strategy that was introduced all the way back in 2010 in this blog post.

The idea is to maintain a number of branches in your project, with two the main (or master) and the develop being in central focus. Then there are additional supporting branches to suit your needs such as feature branches, hotfixes and releases.

This approach is very methodic and structured which can both be a good and a bad thing. If your team is extremely disciplined, then this is the approach for you as it will

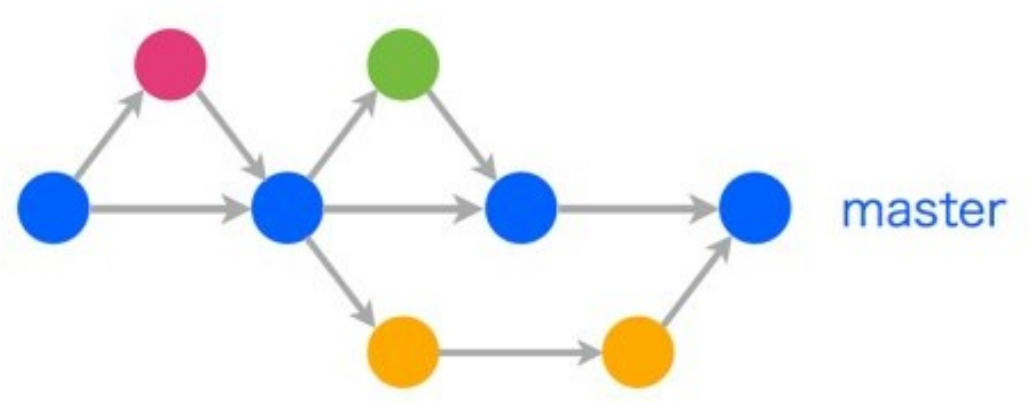
make your development more intuitive and efficient. However, it can also lead to unnecessary complexity which can slow you down and introduce many merge conflicts. The name of the game here, is to keep branches short-lived and merge them often.



GitHub Flow

This model was created by GitHub as an alternative to Git Flow. It offers a simpler and more lightweight branching approach. In this model, there are two types branches the main branch which is always kept production-ready and short-lived feature branches. For more info check the documentation.

Again this has certain benefits and drawbacks. On one hand the workflow is very simple and promotes continues integration as new features are incorporated often. On the other hand, it makes it difficult to support multiple versions at the same time, easier to introduce bugs to the production version and doesn't really help with merge conflicts if your team keeps feature branches open for a long time. This approach is usually preferred by small teams and simpler projects.



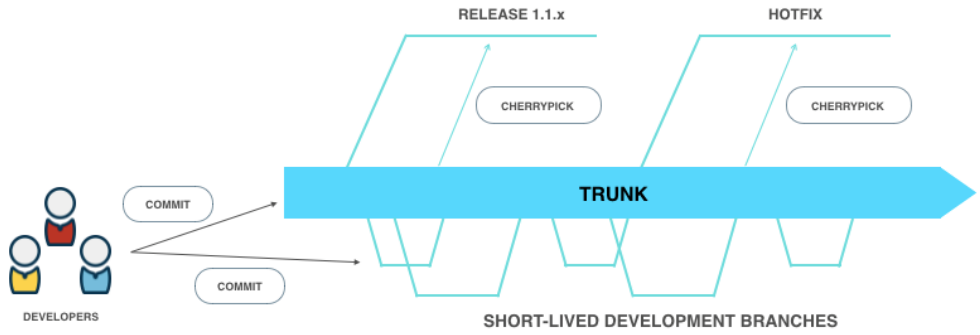
Trunk Based

Finally we will look at an unconventional strategy which is gaining popularity in recent years. This is not a branching strategy as there is only one branch (called trunk) where all developers integrate their changes daily. The trunk should always be production ready and usually developers using this approach integrate their work multiple times per day. The idea is that this way, developers always keep an eye on what each other is doing and adjust to it.

In a sense this approach is very natural, and in a way has been used by people long before Git or other version control systems were introduced. This should increase collaboration and since it basically introduces no overhead should speed up the pace of production. This is also the only true continues integration approach.

However, there are many drawbacks. This can only be practical in very small and very skilled teams. It can doesn't allow code reviews as changes are immediately integrated so you have to trust that mistakes will be few (there is a reason many teams have astrict code review process). Additionally, it forces developers to merge daily or many times per day, which can be especially painful for someone making a big change or changing multiple components. For these reasons, it also can't be used in open source projects.

There are extensions to Trunk based development that make the process easier and safer. For more info check their website.



Bonus Task 2: Workflow Practice

As an example, let's try the GitHub flow.

TO DO: Make sure both workspaces are up to date. Using either A or B, create a new feature branch, add a file called "feature.txt" and commit. Then make a few more changes with both users to this file (you can do them is sequence or in parallel if you want to practice merging again). Finally, merge this branch back to the main branch.