Yrkes
Akademin
Vi hjälper dig att lyckas!

Version Control Systems and Strategies

Version Control Systems and Strategies

- Your Daily Tasks
  - **Create** files, **Edit** and **save** the them **again and again**
    - Version control systems help with
      - History tracking of the content changes
      - Collaboration and collaborative history tracking
      - Comparing and reverting of changes over time
      - Recovering of screwed or lost files
      - And more

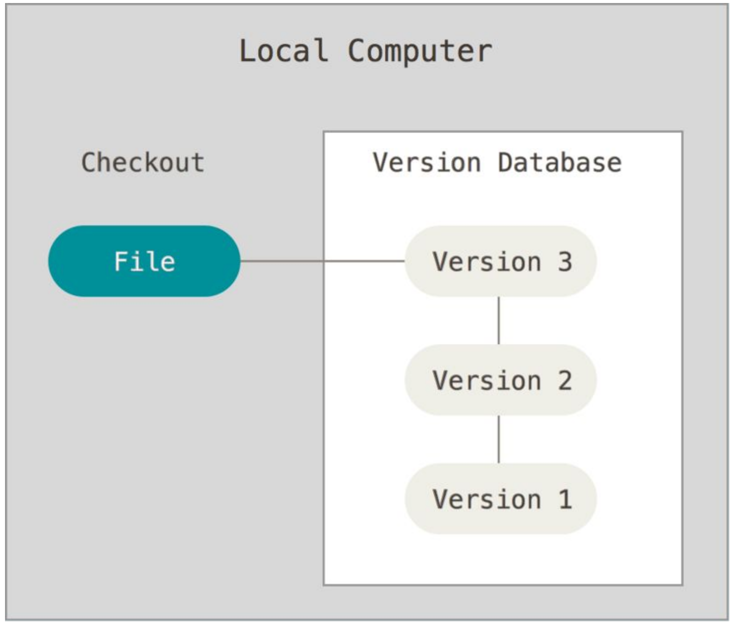
- Version Control System (VCS)
  - Is a system that records changes to a file or

    set of files over time so that you can recall specific versions later

Version Control Systems and Strategies (VCS)

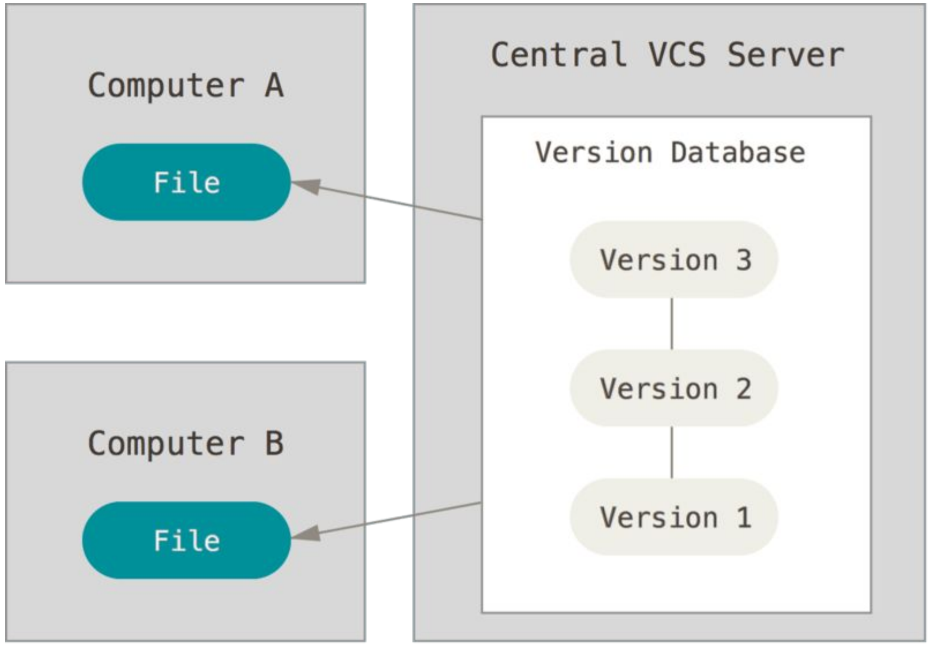
- Version Control System types
  - Local
  - Centralized
  - Distributed

- **Local** Version Control System
  - A simple database that keeps all the changes to files as patch sets (the differences between files) on the disk
  - The disk failure means you risk losing everything
  - It does not support collaboration
  - Example: RCS



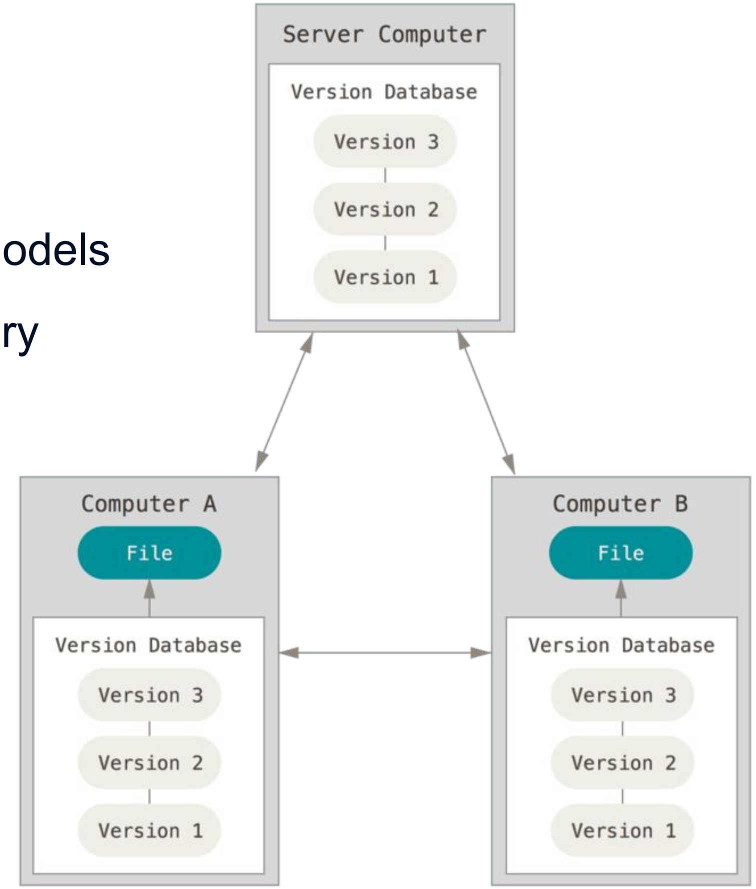
Version Control Systems and Strategies (VCS)

- **Centralized** Version Control Systems
  - Have a single server that contains the entire history of the project
  - Support collaboration and a number of clients can checkout the files
  - Administrators have control over the system
  - In the case of server failure
    - No one can collaborate or save changes
  - If the central database gets corrupted
    - Risk of losing everything exists
  - Examples
    - [SVN]
    - [CVS]



Version Control Systems and Strategies (VCS)

- **Distributed** Version Control Systems
  - Support several remote servers and repositories
  - Support collaboration with different groups of people
  - Support several types of workflows such as hierarchical models
  - Clients check fully out the repository, including its full history
  - Every clone is a full backup of all the data
  - If any server dies, the repositories can be restored
    - By copying of a client repositories to the server
  - And many more...
  - Examples
    - Git (the most popular one)
    - Mercurial

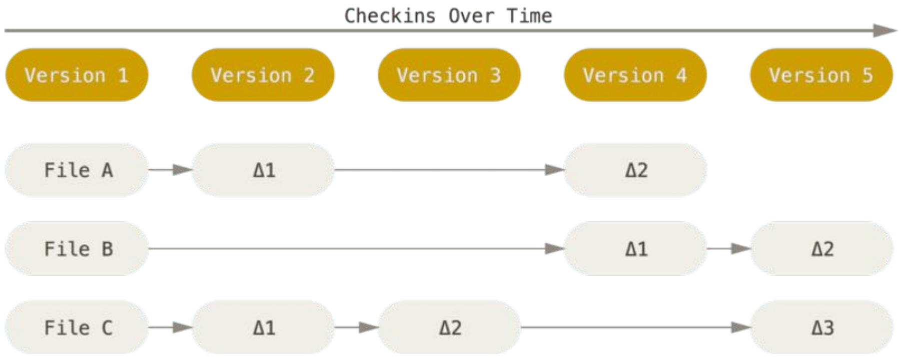


Version Control Systems and Strategies (Git)

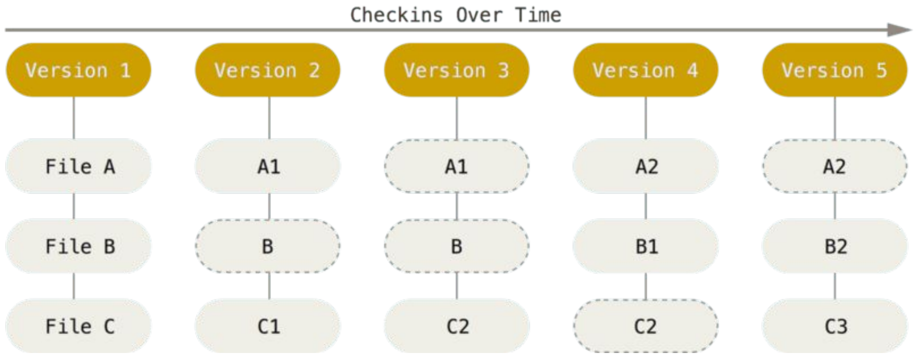
- Created in 2005 for development and maintenance of the Linux kernel by
  - Linux development community
    - In particular Linus Torvalds, the creator of Linux
  - The Linux kernel is an open source software project of fairly large scope
- Some features of Git
  - Speed
  - Simple design
  - Strong support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like the Linux kernel efficiently (speed and data size)
  - Ability to be integrated with automation tools

Version Control Systems and Strategies (Delta-based vs. snapshots)

- Delta-based VCS
  - The information are stored as a set of files and the changes made to each file over time
    - Like SVN, Perforce and etc.
- Stream of snapshots
  - Data is stored as a series of snapshots of a miniature filesystem
  - A picture of what all the files look like at that moment and stores a reference to that snapshot
  - Unchanged file isn’t stored again, just a link to the previous identical file that has already been stored is included in the snapshot



Delta-based version control like SVN, Perforce and etc.



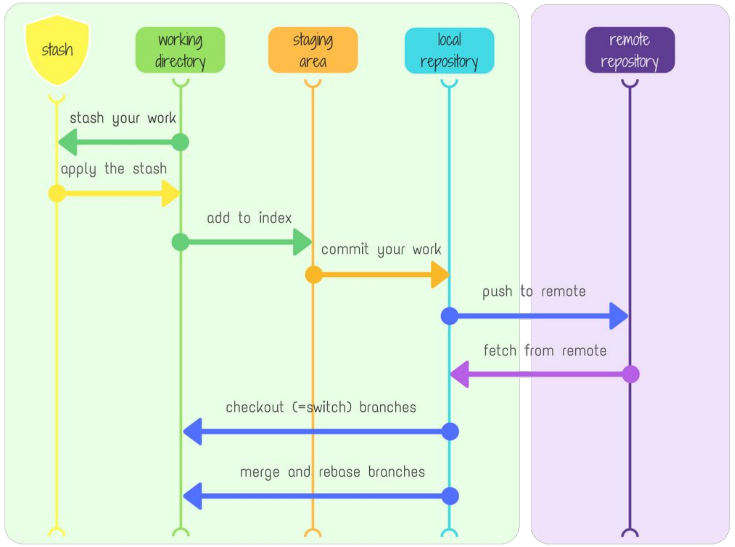
Git: **Stream of snapshots**

Version Control Systems and Strategies (Git)

- Git thinks about its data more like a stream of snapshots
  - It makes Git as a fast DVCS
- Nearly Every Operation Is Local
  - Most operations in Git need only local files and resources to operate
    - Because we have the entire history of the project on your local disk
    - In the case of no connection you can still commit and etc.
    - E.g. unlike Perforce; in Perforce you can’t do much when you aren’t connected to the server
- Git Has Integrity
  - Everything in Git is checksummed before it is stored
  - SHA-1 (40-character string composed of hex characters) is used as the checksum
  - Impossible to change the contents of any file or directory without Git knowing about it

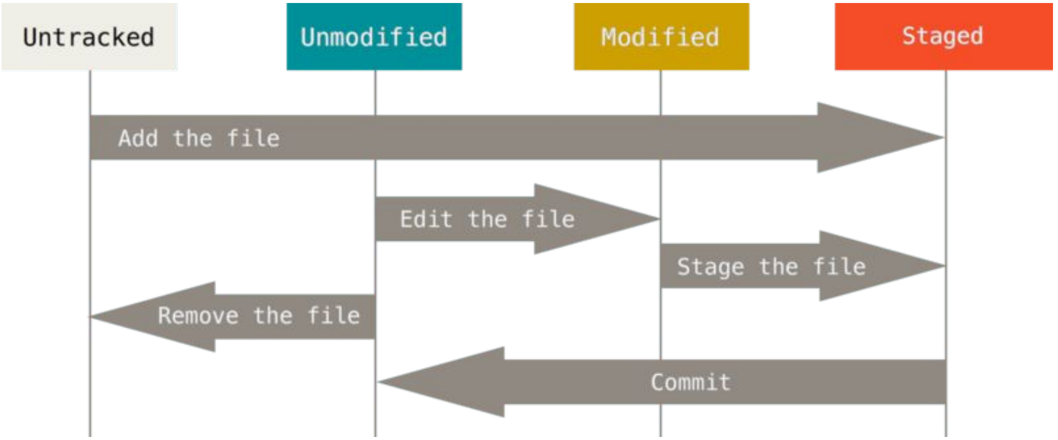
Version Control Systems and Strategies (Git)

- Git Generally Only Adds Data
  - It is hard to make it doing anything that is not undoable
  - It is hard to make it erase data in any way
  - it is very difficult to lose, especially if you regularly push your repository to a sevre
- The three states of the data are **modified**, **staged** and **committed**
- The basic Git workflow
  - Working tree
    - You modify files
  - Staging area
    - You selectively stage the changed files
  - Git directory
    - You make a new commit



Version Control Systems and Strategies (Git)

- Git help: **git --help** or **git <verb> --help** or **git help <verb>** or **git <verb> -h**
- Git configuration: **git config**
  - For examples: **git config --global user.name** and **git config --global --list**
- Create a repository
  - **git init** (in the project directory)
- Checking the Status of Your Repository
  - **git status**
- Tracking New Files
  - **git add <pathspec>**



- Ignoring Files: Create **.gitignore** in the root of the repo and list the files or the patterns in it

Version Control Systems and Strategies (Git)

- Add modified files to the staging area: **git add <file>...**
- Untrack files: **git rm --cached <file>...**
- Delete files from git directory: **git rm <--force> <file>**
- Discard changes in the working directory
    - **git checkout -- <file>...** or **git checkout HEAD <file>...** or in the newer version **git restore <file> ...**
- Unstage the staged files
    - **git reset <HEAD> <file>...** or in the newer version **git restore --staged <file> ...**
- Discard changes in the staging area
    - **git reset --hard <HEAD> <file>...** or **git checkout <HEAD> <file>...**
- Commit changes: **git commit -m “<message>”**
- Redo a commit: **git commit --amend -m “<message>”**
- Checkout a specific commit: **git checkout commit_id**

Version Control Systems and Strategies (Git)

- Show the commit logs: **git log** (for example; *git log --decorate --oneline --graph*)
- Show the reference logs: **git reflog**
- Notes to a commit: **git notes (***for example; git notes add -m “note text” commit_Id***)**
- Show the changes of files: **git diff <pathspec>**
- Delete untracked files: **git clean** (*for example; git clean -xf removes all the ignored files*)
- Undoing changes
  - **git reset commit_id**
    - Unlike git checkout, it moves both the HEAD and branch refs to a specific commit
  - **git revert commit_id**
    - It reverts a specific commit and creates a new commit
    - In case of conflicts:
      - Run **git revert --abort** to abort or **solve** the conflicts and run **git revert --continue**

Version Control Systems and Strategies (Git)

- Rename or moving a file, a directory, or a symlink
  - **git mv oldname newname**
- List, create, rename or delete branches
  - **git branch <--all>**
  - **git branch new_branch base_branch** (e.g. *git branch new_branch*; branch off the current branch)
  - **git branch -d branch_name** (delete a branch)
- Switch to a specific branch
  - **git checkout branch_name** or **git switch branch_name**
- Merge branches
  - **git merge branch_name -m “message”**
  - In the case of conflicts: run **git merge --abort** to abort or solve the conflicts and run ***git merge --continue***
  - For example: **git merge --squash develop -m “develop was merged with master”**

Version Control Systems and Strategies (Git)

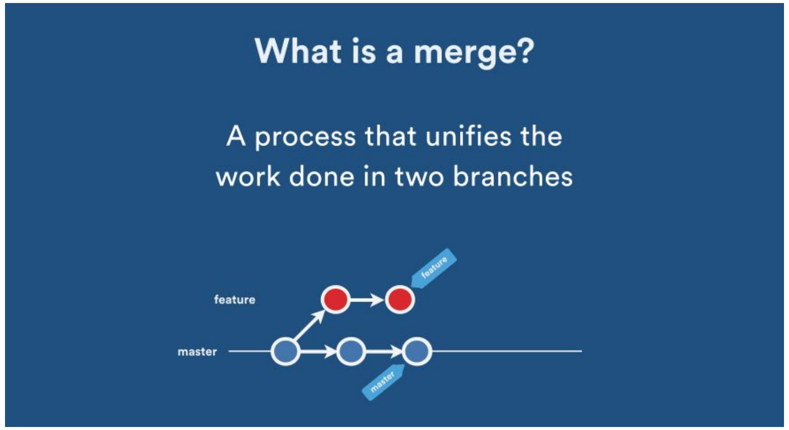
- Git Reset
  - It is a complex tool for undoing changes
  - It moves the **HEAD** and **current branch** ref pointers
  - it is used in three modes (***--soft***, ***--mixed*** *and* ***--hard***)
  - **git reset <mode> <commit_id>** ... (*by default mode is --mixed and commit_id is HEAD*)
- git reset --soft [commit_id]
  - The ref pointers are updated and the reset stops there
  - The staging area and the working directory are left untouched
- git reset --mixed [commit_id]
  - The ref pointers are updated and the staging area is reset to the state of the given commit
  - Any undone changes from the staging area are moved to the working wirectory

Version Control Systems and Strategies (Git)

- git reset --hard [commit_id]
  - The references are updated to the given commit
  - The staging area and the working directory are reset to the given commit
  - Any pending work in the staging area and the working directory will be lost
    - Pending changes in the staging area gets reset to match the state of the given commit
    - Pending changes in the working Directory gets reset to match the state of the given commit
- Git Cherry Pick
  - Is used to pick a commit from a branch and applying it to another
  - Is useful for undoing changes, restoring lost commits, team collaboration and bug hotfixes
  - **git cherry-pick commit_id** <--no-commit | ...>. Conflict? solv it, then **git cherry-pick --continue**
  - Example: **git checkout dev**; **git cherry-pick 129f261**; **git checkout master**; **git cherry-pick 129f261**

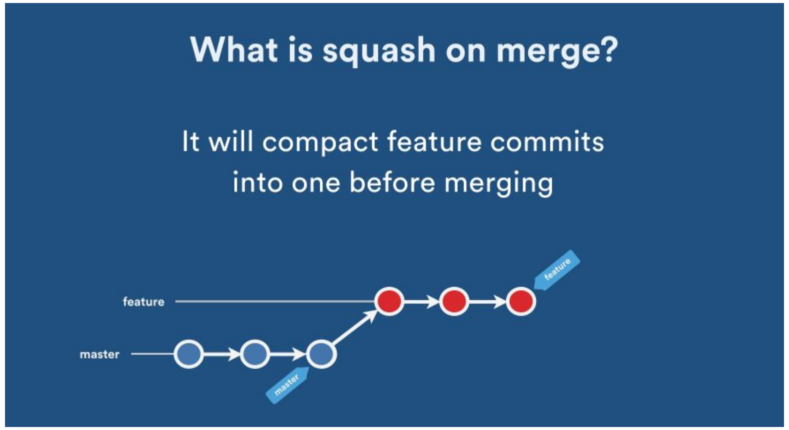
Version Control Systems and Strategies (Git)

- ❖ Merge, Fast Forward Merge, Squash Merge and Rebase



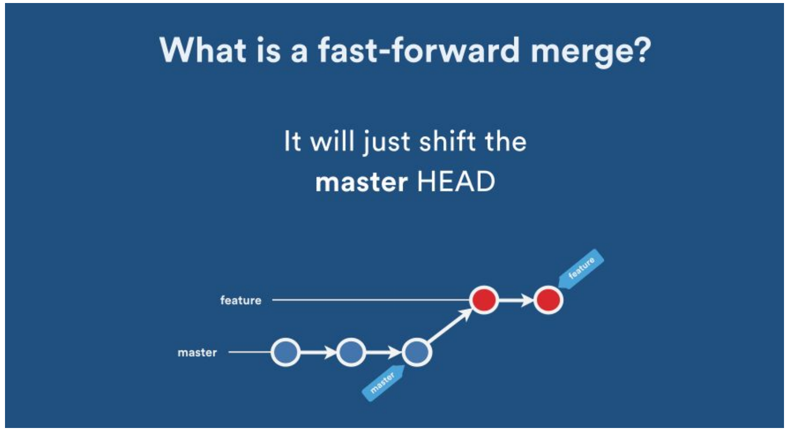
What is a merge?

A process that unifies the work done in two branches



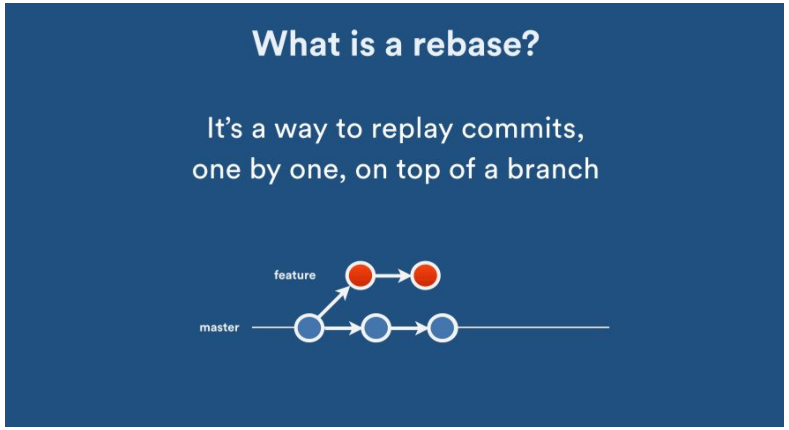
What is squash on merge?

It will compact feature commits into one before merging



What is a fast-forward merge?

It will just shift the **master** HEAD

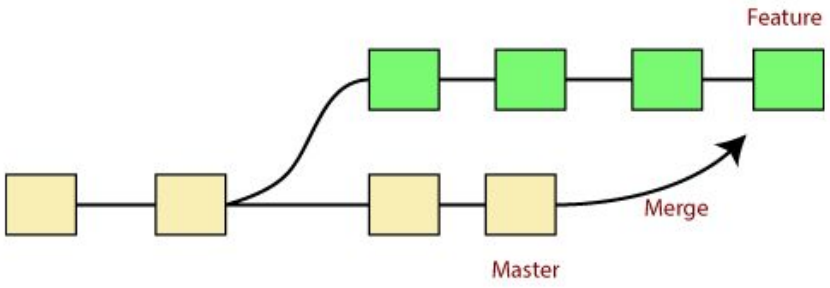


What is a rebase?

It's a way to replay commits, one by one, on top of a branch

Version Control Systems and Strategies (Git)

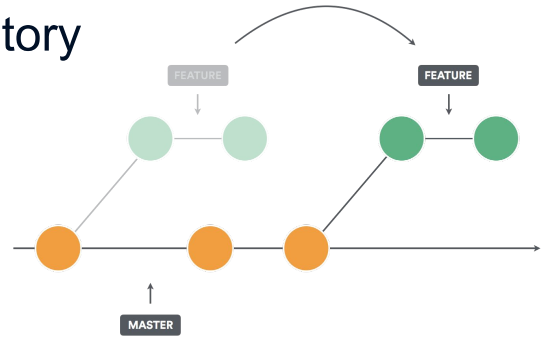
- Git Merge
  - It is used for combining two or more branches
  - **git merge <-s strategy> branch <or branches>** (e.g. git merge -s recursive branch1 branch2)
  - Git will select the most appropriate merge strategy based on the provided branches
  - Supports different merge strategies
    - **Recursive:** is the he default strategy for two heads
    - **Octopus**: is the default merge strategy for more than two heads
    - And etc.



[Git Merge Strategy Options and Examples]

Version Control Systems and Strategies (Git)

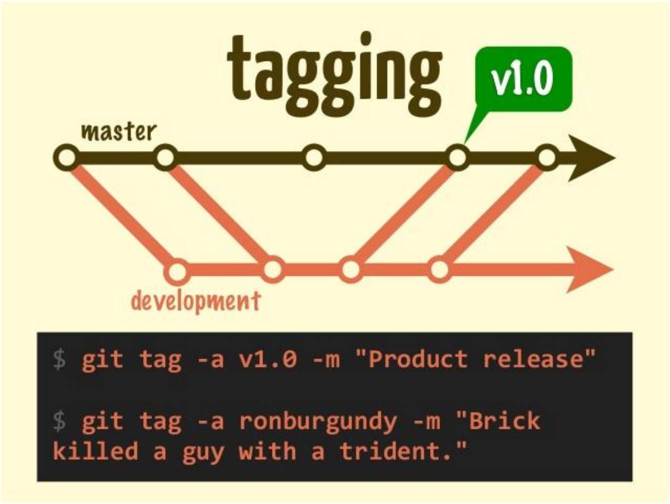
- Rebasing
  - Is the process of moving or combining a sequence of commits to a new base commit
  - The primary reason for rebasing is to maintain a linear project history
  - **git rebase new_base** (for example; *git rebase master*)



- Stashing
  - It takes the uncommitted changes, saves them away for later use
  - **git stash [save ...]**
  - **git stash list / show [<stash>]**
  - **git stash drop/clear**
  - **git stash pop/apply stash@{stash_index}**
  - **git stash branch <name> stash@{stash_index}**

Version Control Systems and Strategies (Git)

- Tagging: tags are ref's that point to specific points in the git history
- Create a tag: **git tag <tagname> [commit_id]**
- Annotated tags (tags with extra metadata): Example: ***git tag -a v1.4 -m "my version 1.4"***
- List tages: **git tag**
- Checking Out Tags: **git checkout tage_name**
- Deleting Tags: **git tag -d tage_name**
- Clone a repository into a directory: **git clone <url> [directory]**
  - E.g. **git clone https://github.com/tryalab/example.git**
- Cache the credential
  - **git config --global credential.helper 'cache --timeout=300'**



```
$ git tag -a v1.0 -m "Product release"

$ git tag -a ronburgundy -m "Brick
killed a guy with a trident."
```

Version Control Systems and Strategies (Git)

- Git Remote
  - It lets you create, view, and delete links to other repositories
  - View the remotes list: **git remote [-v]**
  - Add a remote: **git remote add <name> <url>**
  - Remove a remote: **git remote remove <name>**
  - Rename a remote: **git remote rename <old-name> <new-name>**
- Git Fetch
  - It downloads commits, files, and refs from a remote repository into the local repo
  - **git fetch**, **git fetch remote_name**, **git fetch remote_name branch_name**
  - **git fetch --all** (fetching all branches of all the remotes)
  - To prune the local branches we can: **git fetch --prune**

Version Control Systems and Strategies (Git)

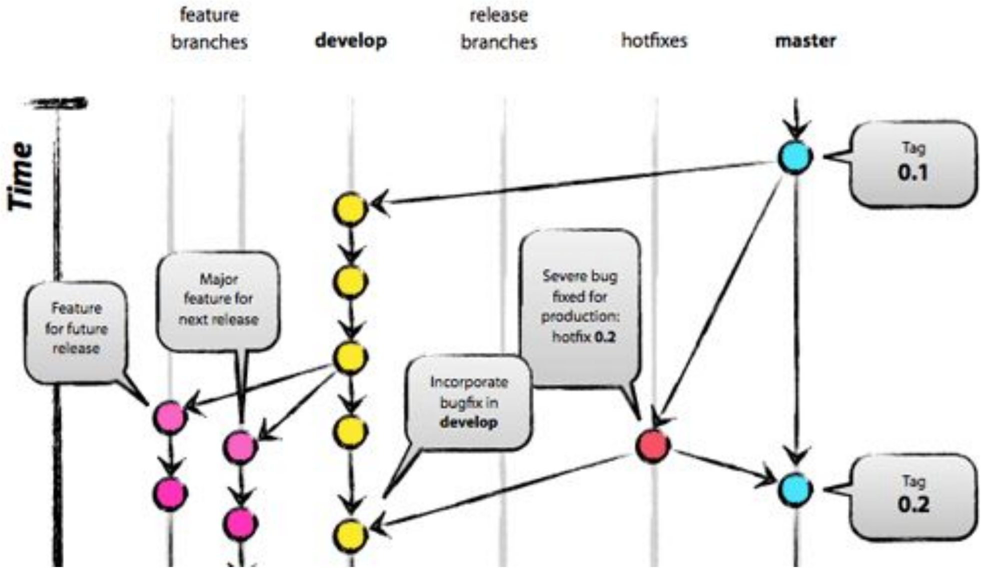
- Git Push (git push)
  - It is used to upload local repository content to a remote repository
  - **git push**
  - **git push --tags**
  - **git push remote_name branch_name**
  - **git push --all** (push all the branches)
  - When we rewrite the history we need to use: **git push --force …**
    - E.g. When we amend a commit; *git commit --amend ...*
  - Delete a remote tag: **git push origin :tag_name** or **git push origin --delete tag_name**
  - Delete a remote branch: **git push origin :branch_name** or **git push origin --delete branch_name**
  - To push a new branch we need to set the upstream: **git push -u origin branch_name**

Version Control Systems and Strategies (Git)

- Git Pull (git pull <remote_name> <branch_name> or <--all>)
  - First it runs **git fetch** which downloads content from the specified remote repository
  - Then a **git merge/rebase** is run to integrate the remote into a new local merge commit
  - **git pull --no-commit** (it does not create a new merge commit)
  - **git pull --rebase** (instead of git merge, git rebase is used)
  - **git pull --verbose** (displays the content being downloaded and the merge details)
- Git submodule
  - Git submodules allow you to have a repository as a subdirectory of another git repository
  - Add git submodule: **git submodule add <url> <directory>**
  - Cloning submodule: **git submodule init** and then **git submodule update**
  - **OR** clone the repo recursively: **git clone --recursive <repo-url>**

Version Control Systems and Strategies (Strategies)

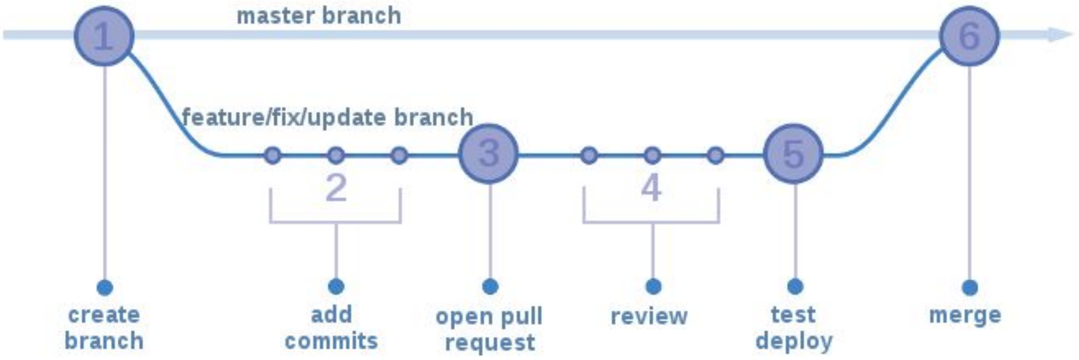
- A strategy is a set of rules and conventions of the workflow which determines:
  - How many types of branch we should have
  - When a developer should branch
  - From which branch it should be branched off
  - Which merge strategy should be used
  - When a branch should be merged back
  - And to which branch it should be merged back


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2

- Popular strategies
  - GitHub Flow (feature branch)
  - GitFlow
  - Forking Workflow
  - Centralized Workflow (All the changes are committed to the **master** branch)

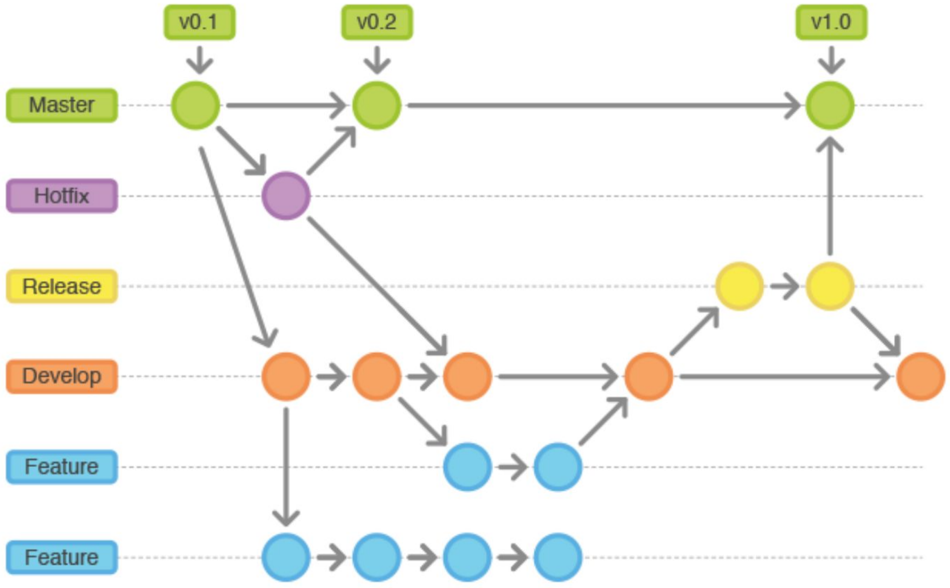
Version Control Systems and Strategies (Strategies)

- GitHub Flow
  - Anything in the master branch is deployable
  - Create descriptive feature branch off of **master** (e.g. user-content-cache-key)
  - Commit to the feature branch locally and regularly push your work to the server
  - Open a pull request if you need feedback/help, or the branch is ready for merging
  - Merge only after pull request review
  - Once it is merged and pushed to **master**, you can and should deploy immediately



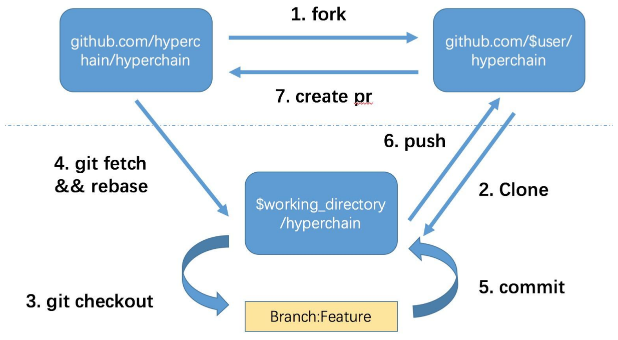
Version Control Systems and Strategies (Strategies)

- GitFlow: It relies on
  - Two long-lived branches
    - The permanent one which is **master**
      - It is the stable version of the product
    - A potentially unstable branch
      - Where **development** happens in
      - Which is a branch off **master**
  - And some short-lived ones
    - Feature branches (branch off **development**)
    - Release branches (branch off **development,** merge to **master,** **tag** the commit on master)
    - Hotfix branches (branch off **master**, merge to **master** and **development**)



Version Control Systems and Strategies (Strategies)

- ❖ Forking Workflow
  - ➢ A developer forks an official server-side repository. This creates her/his own server-side copy.
  - ➢ The new server-side copy is cloned to the developer’s local system
  - ➢ A Git remote path for the official repository is added to the local clone
  - ➢ A new local feature branch is created
  - ➢ The developer makes changes to the new branch
  - ➢ New commits are created for the changes
  - ➢ The branch gets pushed to the developer's own server-side copy.
  - ➢ The developer opens a pull request from the new branch to the official repository.
  - ➢ The pull request gets approved for merge and is merged into the original server-side repository



- ➢ *It is widely used in developing of open-source software projects*

Version Control Systems and Strategies

- Some useful links

  - Git - Documentation
  - Git Handbook
  - Git Cheat Sheet
  - What is Version Control?
  - What Is a Branching Strategy?
  - Define a Branching Strategy
  - Git Branching and Merging Strategies
  - Comparing Workflows
  - Git patterns and anti-patterns
  - Git Tutorial – Commands And Operations In Git
  - Setting up Git with Visual Studio Code
  - Introduction to Git - Branching and Merging
  - Git: Command-Line Fundamentals
  - Git & GitHub Crash Course For Beginners
  - Git and GitHub for Beginners - Crash Course
  - Learn Git in 20 Minutes
  - Top 20 Git Commands With Examples
  - Learn Git in 1 Hour
  - Git merge vs rebase