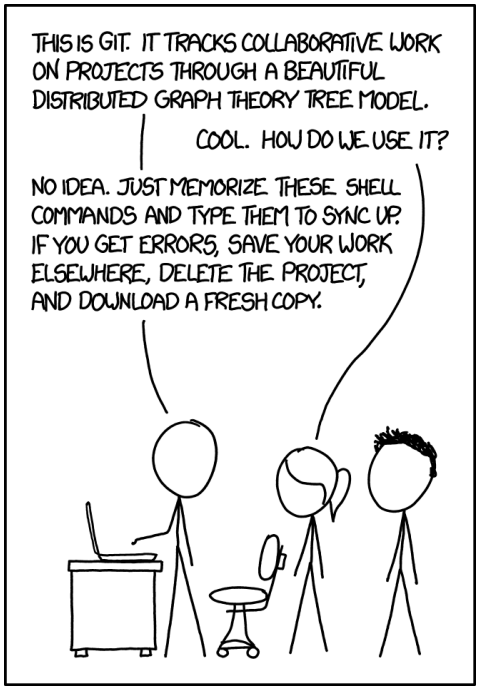
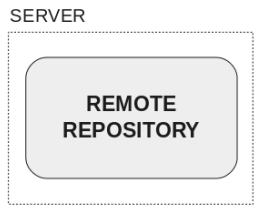
# git

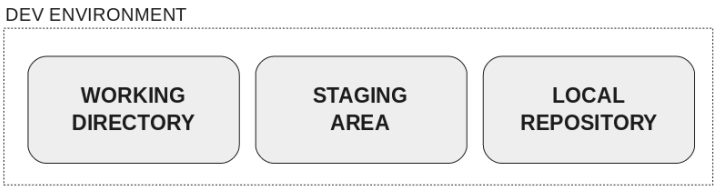
An SSE Talk



# Git what?

- Distributed Version Control Software
- Distributed
  - Version history is stored across many devices
- Version Control
  - Tracks different versions of a project



# Git Config

- Two configs: global and local
- Manage Identity

```
$ git config --global user.name="Git Guru"
$ git config --global
user.email="guru@git-scm.com"
```

- Configure how line endings are handled

```
$ git config --global core.autocrlf true
```

- Set up aliases

```
$ git config --global alias.psuh push
```

- Manage which text editor you want to use

```
$ git config --global core.editor vim
```

# Setting up a repository

Repository:

a central location in which data is stored and managed.

```
$ git init
```

- Creates a .git folder in the current directory
- .git folder contains information Git uses

```
$ git init --bare
```

- Creates a “bare” repository
- This repository can be used as a “remote”
- For most cases, GitHub, BitBucket, GitLab, etc. does this for you

Remote:

A repository that exists outside of your development environment

# Remotes


`$ git remote list`

- Lists all remotes for your repository

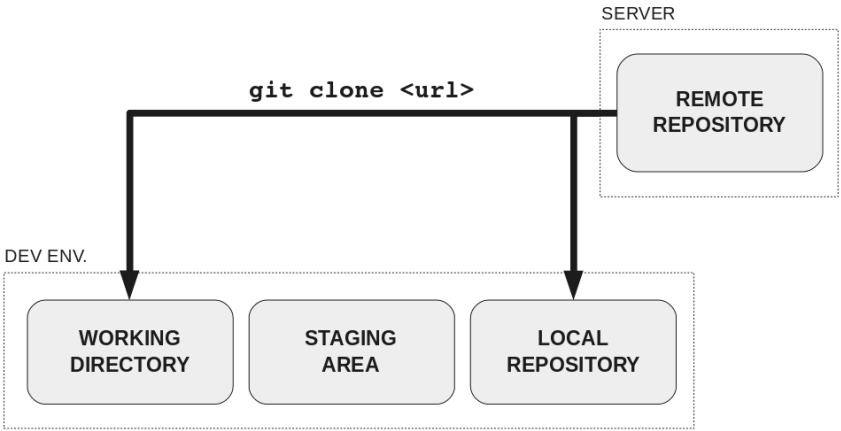
`$ git remote add name url`

- Adds a reference to a new remote

- Multiple remotes is how “Forks” work
  - First clone the repository you want to fork
  - Create a bare repo where you want to host the fork
  - Add the bare repo as a new remote
  - Push to the new remote

Check out git spooning https://bitbucket.org/spooning/

# Git Clone




```
$ git clone https://your.repo.url.com/repo
```

- Creates a local copy of the remote repository
- Stores a `remote` in your local repo that points to that url

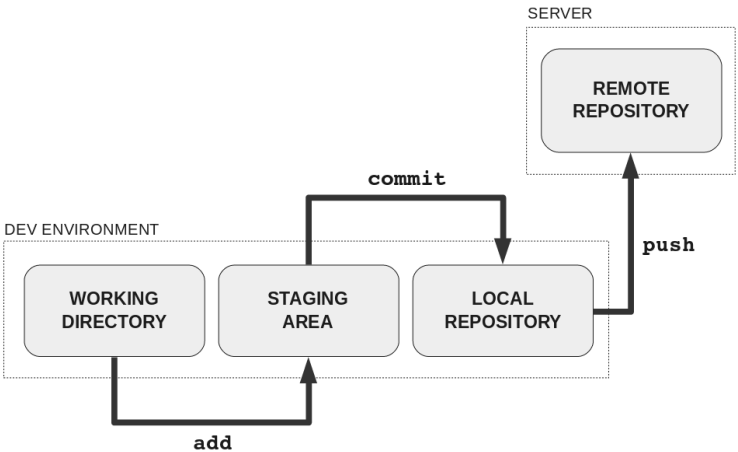
```
$ git clone you@your.repo.url.com:repo.git
```

- Does the same thing, but does so over SSH, configuring Git to use your account

Standard Workflow

# Making and Staging changes

- Changes are not automatically saved
- Process to save changes to the remote repository



## Add

`$ git add filename`

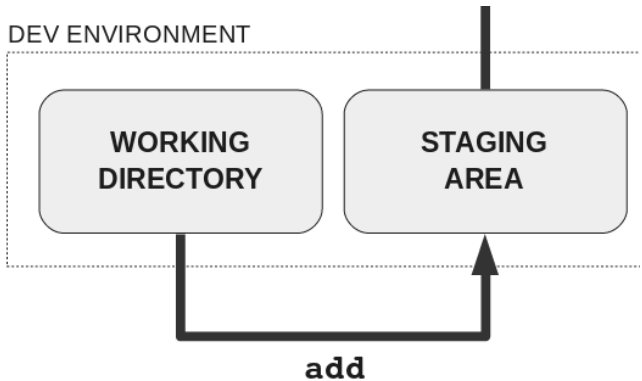
- Supports wildcards
  - `$git add *.java`

`$ git add -A`

- Adds *all* of the tracked files in the current working directory

`$ git status`

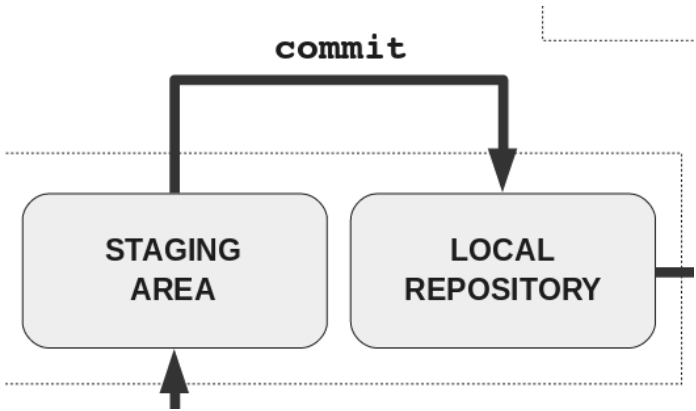
- Shows checked and unchecked files



# Commit

`$git commit -m “Meaningful commit message”`

- Creates a snapshot of the current repository
- Stored as line differences from previous commit
- Takes effort to change, commit wisely



# What is a commit?

- Often Visualized with Directed Graphs
- Contains information about
  - Who made the changes
  - What changes where made
  - When the changes were commit
  - The previous commit(s) in the graph
  - A message about the changes
  - Commit SHA, a unique ID

`$ git log`

- Shows the commit history
- Git keeps track of history on a branch by branch basis
- Your local history may differ from the history on a remote

# More about Commits

- HEAD
  - Often called a “pointer” to a commit
  - Refers to the most recent commit in the current branch
- Refnames
  - You can refer to commits relatively to other commits such as HEAD
  - HEAD~1 means the previous commit
  - HEAD~2 means the commit before the previous commit
  - There are a lot more ways to navigate the structure, learn here: https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection

- Sha1
  - Every commit gets a hash value: `ec392edb07881315ee6d73edb055da49ab671d30`
  - This can be shortened to the first 7 numbers: `ec392ed`

# Push

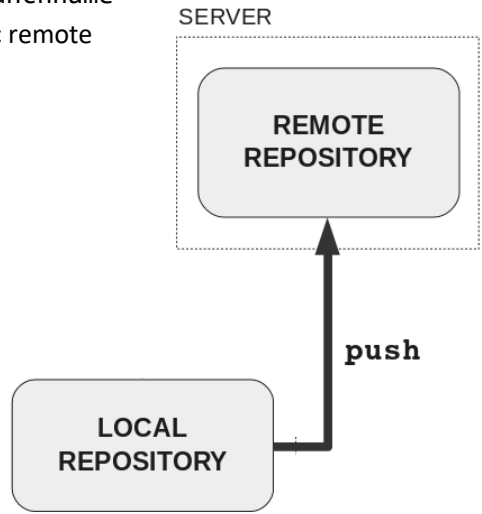
`$ git push origin master`

- Sends the changes from one branch to the corresponding remote branch
- If the remote's history differs from the local history, the push may fail

`$ git push -u remote branchname`

- Tells git to default to a specific remote

`$ git push -f`

# Commitment Issues: How to Undo

`$ git reset --hard 4d4f80b`

- Sets the local repository back to 4d4f80b
- Working directory is discarded

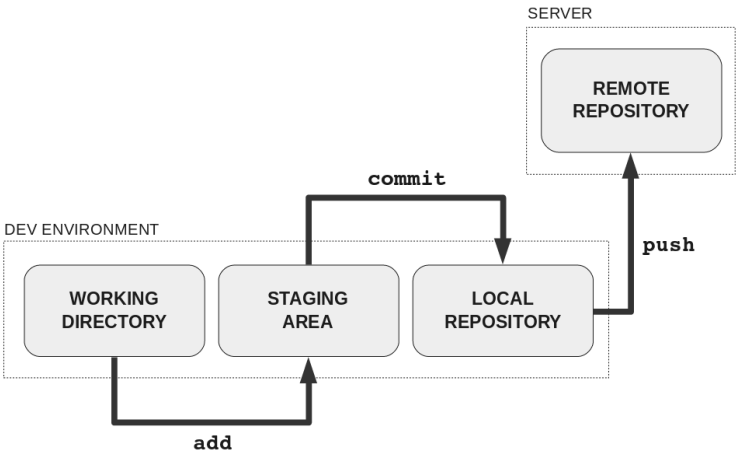
`$ git reset --hard HEAD~2`

- Undoes 2 commits

`$ git revert 4d4f80b`

- Creates a NEW commit
- Changes made after `4d4f80b` are undone

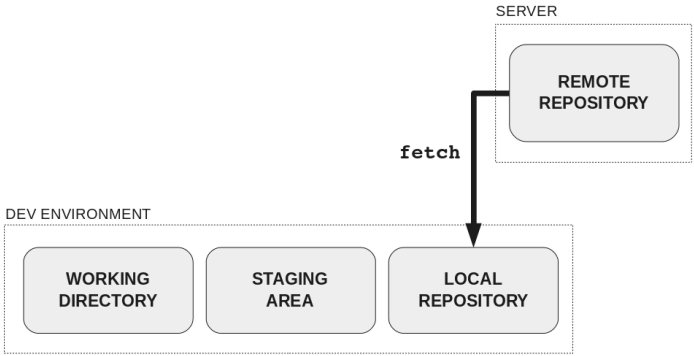
# Standard Workflow Review



# Fetch

`$ git fetch [remote]`

- Retrieves changes made on the remote repository



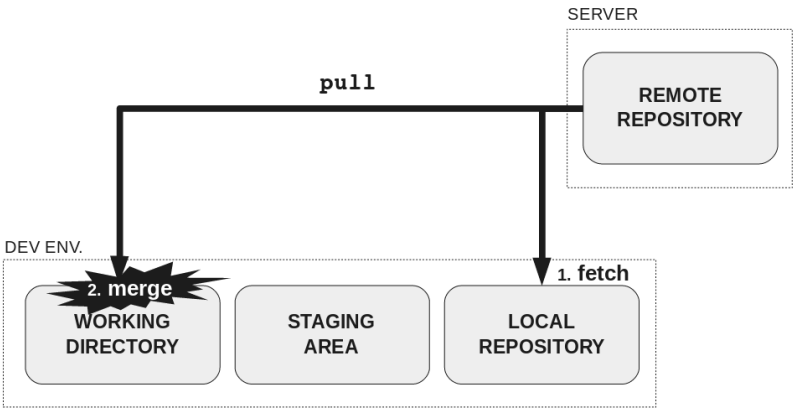

# Pull

`$ git pull [remote]`

- Functionally equivalent to
  - `$ git fetch [remote]`
  - `$ git merge`

`$ git pull –rebase [remote]`

- replace merge with rebase



# Ignoring Files

- Some files you don't want to commit to the remote
- Create a text file titled .gitignore
- .gitignore applies to the directory and its children
- Each line is a pattern of which files to ignore
- Lines that start with # are comments
- Must include the trailing / for directories

```
*.o
[Bb]uild/
```

- Some IDEs will autogenerate a .gitignore appropriate for your project

# Branching

- Some changes could break other features
- Create branches to commit changes without impacting the master branch

```
$ git branch feature
```

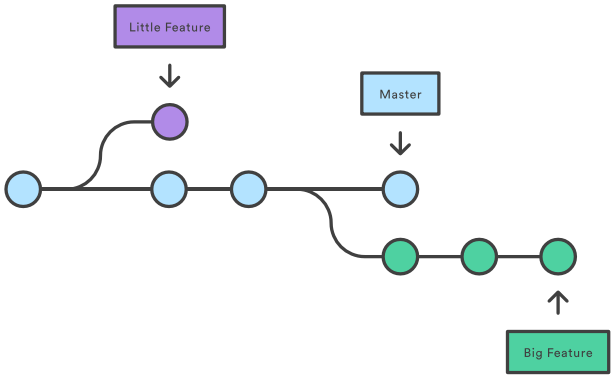
- Creates a local branch feature

```
$ git checkout feature
```

- Changes HEAD to point to feature
- Any new commits end up on this branch
- Pushing from a new local branch creates a new branch on the remote

```
$ git checkout -b feature
```

- Creates a new branch and checks it out



# Checkout

`$ git checkout branchname`

- Used to swap between branches

`$ git checkout ec392ed`

- Moves HEAD to the commit `ec392ed`
  - This creates a `DETACHED HEAD`
  - Any commits you make off of this branch will be garbage collected unless you create a new branch with

    `$ git checkout -b newBranch`

`$ git checkout -`

- Checks out the last branch you were on

`$ git checkout master myFile.java`

- Checks out only myFile.java from the master branch

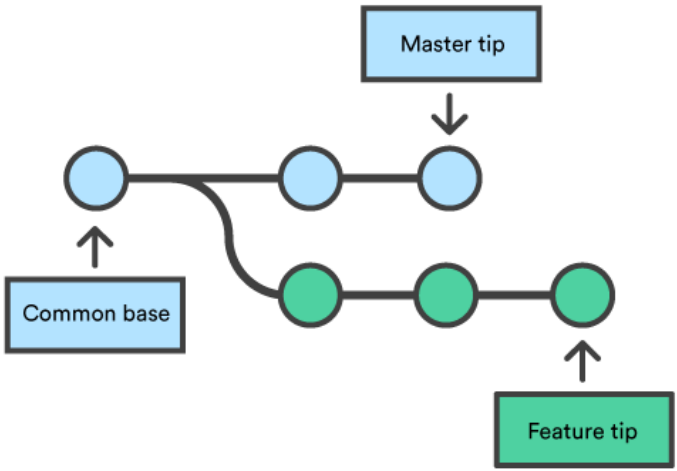
# Merging

- Merge changes from one branch into another

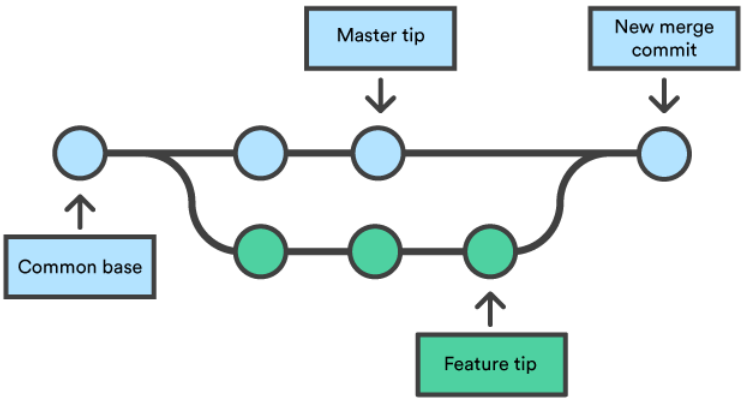
```
$ git checkout master
$ git merge feature
```

- Merges the changes on `feature` into `master` and creates a new commit
- For the most part, merging is pretty painless
- **Merge conflicts can arise**

# Merging Visualized



# Merging Visualized



# Dealing with Merge Conflicts

1. Resolve conflicts in your files
2. `$ git add -A`
3. `$ git merge --continue`

Alternatives:

`$ git merge --strategy=ours`

- Keeps the changes on the checked out branch, ignoring changes on the merging branch

`$ git merge --strategy=theirs`

- Keeps the changes from the merging branch, discarding our changes when necessary

Check out more strategy choices here:

https://git-scm.com/docs/git-merge#_merge_strategies

$ git merge --abort

- Basically giving up on life and trying something else

# Other cool commands

`$ git commit --amend`

- Modifies the last commit
- Any staged changes will be applied

`$ git stash`

- Stores moves any unstaged changes to a hidden stash
- Allows you to checkout another branch without committing or losing work

`$ git stash pop`

- Recovers previously stashed work

`$ git rm filename`

- Stages a file for deletion in the next commit

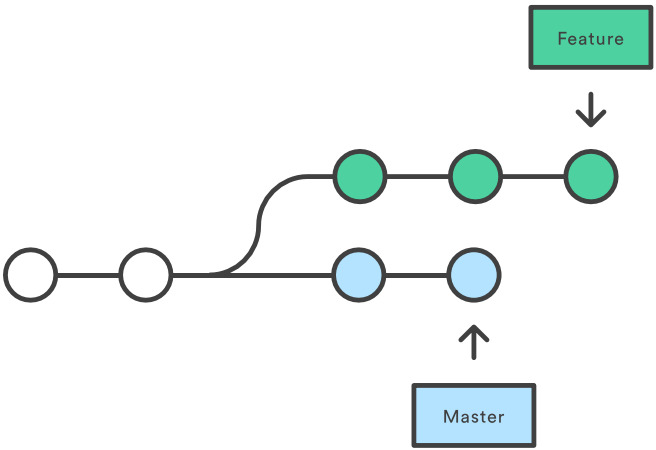
`$ git diff`

- Shows the differences between the working directory and the local repository

# Rebasing

- Literally changing the base commit of a branch to the new tip of a branch
- Useful for handling merge conflicts before they become a bigger issue

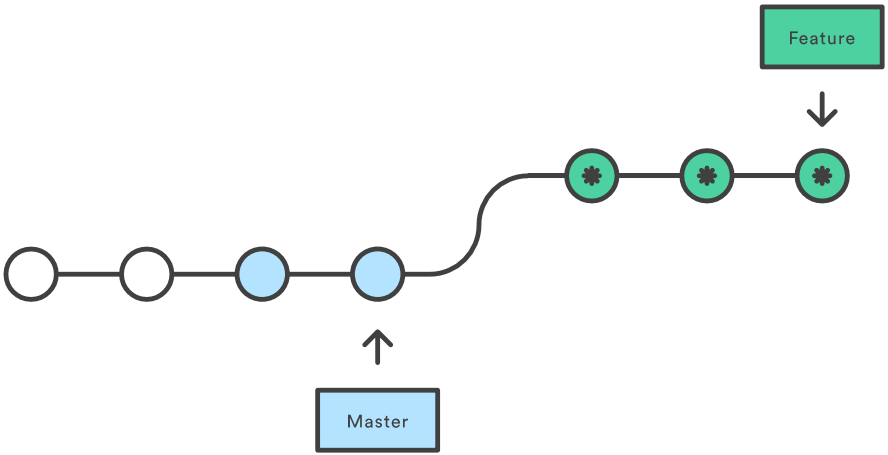
A forked commit history



# Rebasing

- Literally changing the base commit of a branch to the new tip of a branch
- Useful for handling merge conflicts before they become a bigger issue

Rebasing the feature branch onto master



# How to Rebase

- Rebase replays the commits from the checked out branch onto the rebasing branch

```
$ git checkout feature
$ git rebase master
```

- One by one the commits in feature will be reintroduced to the branch
- This will present merge conflicts as they arise and let you logically walk through them

```
$ git rebase master --strategy=ours
```

- This will take the work that is in `master` and discard any of the work in `feature` that conflicts with master. Use theirs to keep the work in feature.

# Rebase Interactive

```
$ git rebase -i a313beb
```

- Rebases in interactive mode
- Opens your text editor and lets you pick what to do for each commit
- Powerful tool, but ripe for abuse
- Literally lets you rewrite history
- Common Use Cases:
  - Removing commits
  - Squashing commits

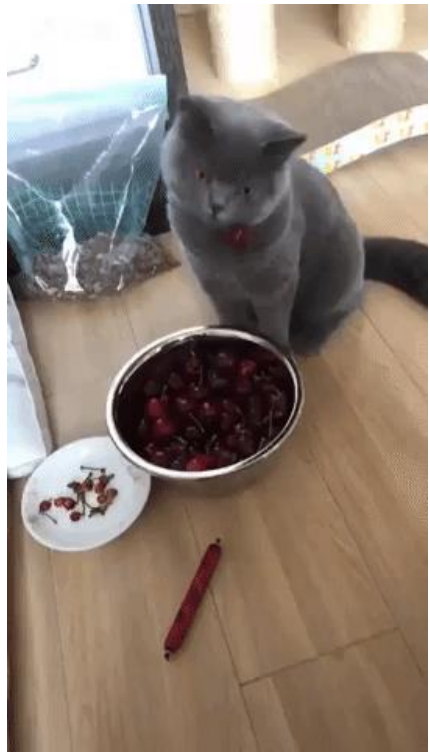
- Might require a force push once complete

# Cherry Pick

```
$ git cherry-pick a313beb
```

- Takes the changes from commit a313beb on another branch and applies those changes to the current branch as a new commit

- Allows you to:
  - incorporate changes from a branch one at a time to prevent merge conflicts
  - recover work from a corrupted branch
  - Test code from a feature branch in a sandbox branch



# Next Steps

- Git Hooks
- Pull Requests/Merge Requests
- Repository Configuration
  - Protected Branches
- Tags
- Dry Runs
  - Try a command to see its affects without risking

# •PRACTICE!

# Useful Links

Git Documentation:
https://git-scm.com/docs

Specific Git Tutorials:
https://www.atlassian.com/git/tutorials

If Command Lines aren't your thing:
https://www.gitkraken.com/

How to Write a Git Commit Message
https://chris.beams.io/posts/git-commit/

Funny Git Commit Message
https://whatthecommit.com

Demo Walkthrough:
www.github.com/smlevorse/GitTechTalk