BASIC WORKFLOW

BASIC WORKFLOW

1. Open terminal and create directory on your machine
2. Go into directory in terminal
3. Initialize repository in this directory
    a. git init
    b. This will create .git hidden folder in your directory which will make your current folder, a git repository

BASIC WORKFLOW

4. Create two files
5. Check status, it will show you two untracked files
   a. git status
6. Add these files to staging area
   a. Three ways to add files in staging
      i. git add filename-1 filename-2 OR
      ii. git add .
      iii. git add *.html

BASIC COMMANDS

- git status
- git commit –m “commit message”
- git log

UNSTAGE FILES OR REMOVE CHANGE

- git reset command will remove file from staging area.
- git reset can remove changes in files if the are not committed by using git reset –hard command.

IGNORING FILES

- There are a couple of files in a project that you don‘t want to be version controlled so.
  - Create an empty file in your favorite editor and save it as ”.gitignore” in your project‘s root folder.
  - You can define rules in “.gitignore” file to ignore files.
  - Add file or directory path or extension or name.

IGNORING FILES

Ignore one specific file

- path/to/file.ext

Ignore all files with a certain name (anywhere in the project)

- filename.ext

Ignore all files of a certain type (anywhere in the project)

- *.ext

Ignore all files in a certain folder:

- path/to/folder/*

BASIC WORKFLOW DEMO WITH TERMINAL

BASIC WORKFLOW DEMO WITH SMART GIT

BRANCHES AND MERGING

WHY BRANCHES ARE IMPORTANT?

- In every project, there are always multiple different contexts where work happens.
- Each feature, bugfix, experiment, or alternative of your product is actually a context of its own.
- There can me unlimited amount of different contexts.
- Most likely, you‘ll have at least one context for your “main” or “production” state, and another context for each feature, bugfix, experiment, etc.

WHY BRANCHES ARE IMPORTANT?

In real-world projects, work always happens in multiple of these contexts in parallel:

- While you‘re preparing two new variations of your website‘s design (context 1 & 2)
- you‘re also trying to fix an annoying bug (context 3).
- On the side, you also update some content on your FAQ pages (context 4)
- one of your teammates is working on a new feature for your shopping cart (context 5)
- and another colleague is experimenting with a whole new login functionality (context 6).

WHY BRANCHES ARE IMPORTANT?

Another example:

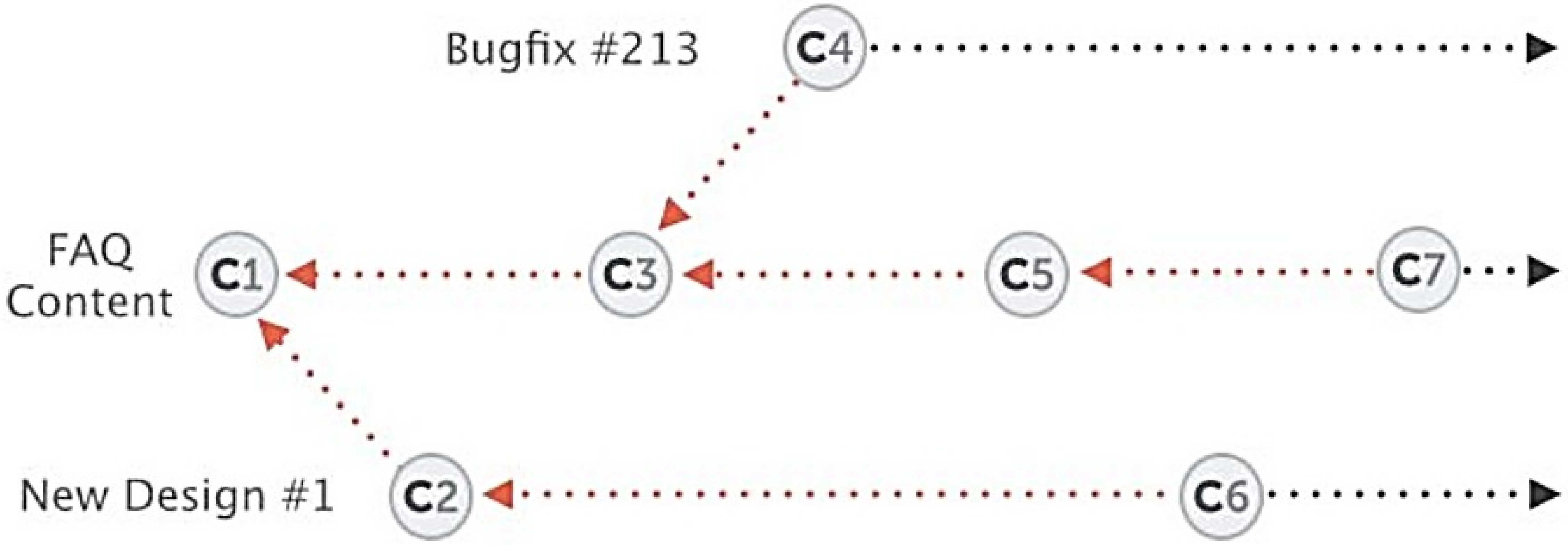
- Production, development and feature context.
- Production code is tested and deployed and we don’t want any problem in that.
- Development team is working on new feature and we don’t want that feature to effect production code until it is properly tested.
- All feature combines in development to have final testing before deployment.

BRANCHES

Branches are what we need to solve context problems.

Because a branch represents exactly such a context in a project and helps you keep it separate from all other contexts.

BRANCHES



BRANCHES

All the changes you make at any time will only apply to the currently active branch; all other branches are left untouched.

This gives you the freedom to both work on different things in parallel and, above all, to experiment - because you can‘t mess up!

In case things go wrong you can always go back / undo / start fresh / switch contexts.

WORKING WITH BRANCHES

Without knowing, we were already working on a branch.

This is because branches aren‘t optional in Git: you are always working on a certain branch (the currently active, or “checked out”, or “HEAD” branch).

Check ‘git status’ and it will show you current branch.

The “master” branch was created by Git automatically for us when we started the project.

‘master’ branch does not mean anything special ❖ It is initially created when we start project.

‘HEAD’ always represent current branch.

BRANCH COMMANDS

git branch

- Show list of branches

git branch –v

- Show list of branches with some details

git branch new-dev

- Creates new branch with name ‘new-dev’

git checkout new-dev

- Switch to ‘new-dev’ branch

git merge new-dev

- Merge ‘new-dev’ branch into current active branch

git log new-dev..master

- Show commit difference in two branches

DEMO WITH TERMINAL

DEMO WITH SMART GIT

MERGE CONFLICT

DEMO WITH SMART GIT

STASH

STASH

- Commit wraps up changes and saves them permanently in the repository
- Stash save changes temporarily

STASH

- If you want to switch to another branch without committing changes in current branch.
- You will need to save your changes somewhere so that you have clean working directory.
- Stash will work in this case.

STASH

- Later, at any time, you can restore the changes from stash in your working copy - and continue working where you left off.
- You can create as many Stashes as you want.

STASH COMMANDS

- git stash

Save local changes in stash clipboard

- git stash save <name>

Save local changes in stash clipbaord with the name provided in command

- git stash list

Show list of stashes

- git stash pop

Apply latest stash and remove it from clipboard

- git stash apply stashname

Apply specific stash and that stash will remain saved in clipboard

REMOTE REPOSITORIES

REMOTE REPOSITORIES

Most of the version control related work happens in the local repository: staging, committing, viewing the status or the log/history, etc.

If you're the only person working on your project, chances are you'll never need to set up a remote repository.

Only when it comes to sharing data with your teammates, a remote repo comes into play.

REMOTE REPOSITORIES

Think of it like a "file server" that you use to exchange data with your colleagues.

Local repositories reside on the computers of team members.

In contrast, remote repositories are hosted on a server that is accessible for all team members.

REMOTE REPOSITORIES

For remote repository you have to first select the online service that offer git and allow you to create remote repository.

Following are few online services:

- GitHub (https://github.com)
- BitBucket (https://bitbucket.org)
- GitLab (https://gitlab.com)
- Many others.

GIT HUB

Git hub is online service for git to create remote repositories on github’s server and collaborate with teammates/colleague.

CLONING A REPOSITORY

Go to the local root folder in terminal,

Type git clone https://github.com/usama/MyProject.git

WORK ON REMOTE REPOSITORY

After clone, make changes to your project on your local machine

- Add file or change a file.
- Add file to staging area by git add .
- Commit file git commit -m “updated feature”.
- Run command “git push” to push your change to remote repository on server.

COMMANDS TO INTERACT REMOTE REPOSITORY

git push

- Push changes to remote repository.

git fetch

- Fetch changes from remote repository.

git merge

- Merge changes that was fetch by ‘git fetch’ command.

git pull

- Fetch and merge changes from remote repository

git remote -v

- Show remote urls

git remote show origin

- Show details of origin

COMMANDS TO INTERACT REMOTE REPOSITORY

“git remote add myremote https://github.com/usamamusharaf/MyProject.git”

PUBLISH A LOCAL REPOSITORY TO GIT HUB

If you already have local repository on your machine and want to connect it with remote repository then:

Create repository on github

Open terminal in your local repository

Run following command

- git remote add origin https://github.com/usamamusharaf/MyProject.git”
- git push -u origin master

DEMO WITH TERMINAL

DEMO WITH SMART GIT

PUBLISH A LOCAL BRANCH TO A REMOTE REPOSITORY

git branch work

This will create new branch ‘work’ on local repo.

git checkout work

Switch to ‘work’ branch

Make changes

git push -u origin work

HAVE A GOOD DAY!