Git Fundamentals

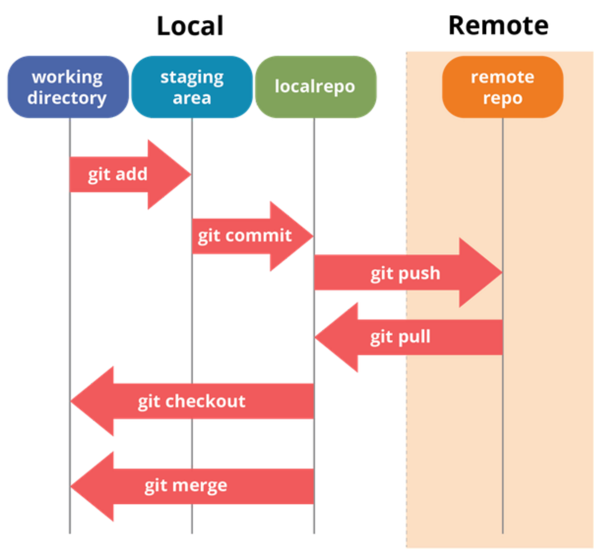
Version Control

- Git (Version control System)
- Used by anyone who creates content
    - Versions update
    - History of changes / commits
    - Revert to earlier version
- Offsite copy of data
    - Easy synchronisation with local copy
- Supports collaboration
    - People can update simultaneously
    - Overlapping changes
    - Branches - (dev) feature or experimental branches and merging to main
    - Merging and dealing with conflicts

Git

- CLI called **Git bash** to run commands
- Git server used to copy data - GitHub GitLab BitBucket
- Graphical UI - GitHub Desktop and IDE integration
- Directory or folder controlled by git is a repository or repo
- Single branch or multi branch
- Scope levels: System, Global, then Local
    - Change default branch name
    - **Git config --global init.default branch main**
- Commands (*pwd, mkdir GitExample, cd, ls or ls -al*)
    - Git init (run in folder to be controlled by git)
        - Creates a sub folder called .git, stores config information for the repo, remembers all history and versions of the files, all branches, keeps track of currently working HEAD
- **.gitignore t**emplates to exclude files from my repo such as .env or .properties
- HEAD - reference to the currently checked-out branch latest commit

Staging and committing files



Commands

- **Git add**
- **Git status** - shows untracked files
- **Git commit** -m “Initial commit”
- **Git diff** - difference between working files and commits
- **Git log**

Github

- Copy a local repo to Github
    - Copy the HTTPS URL
    - Set the connection to GitHub and push
    - **Git remote add origin <url>**
    - **Git push origin main**
    - Create personal access token
    - All tracked files will be copied to GitHub
- Clone a repository
    - **Git clone <url>**
- Pull a new version
    - Checks differences between local and remote repos and downloads changes and updates local files
    - **Git pull**
- Changing from one branch to another
    - **Git checkout**
- Merging contents of working directory with a version from the local repository
    - **Git merge**

Collaborative git

Git authority is responsible for creating and configuring the initial repository Team members are added as collaborators or members to a repository. These individuals have specific permissions based on their role, such as read-only access, write access, or administrative access.

- Create repository
- Public, add readme, and add gitignore
- Add a collaborator using github id or email
- Team members can accept invitation to repository and clone the repo
- Git status command is used to display the current state of the working directory and staging area in relation to the Git repository. It provides information about which files have been modified, which are staged for the next commit, and which are untracked. Displays branch and whether you are up to date and if the working tree is clean

Adding basic content

- Git add . is used to stage changes for the next commit
- Git status
- Git commit -m is used to save the changes you've staged in your local repository
- Git push origin main (destination and branch)
- Git push and enter personal access token to push local changes to remote repository on Github

Branches

Git branches are independent lines of development in a Git repository, allowing multiple tasks or features to be worked on simultaneously while keeping changes isolated from the main codebase until they are merged.

- Single branch projects work for solo developers as work is done serial and limits the usefulness of the team
- Multiple branch projects, create a dev branch for development work for the sprint for example, merge dev branch to main branch to produce the next working version of software. Main branches contains tested and working code
    - Small groups of developers can create a feature branch which represents work on one element of the code and is merged into the dev branch. Feature branches allow parallel working in teams as we can have multiple feature branches which can be assigned to multiple teams.

- The "main" branch is the default and typically the most stable branch in a Git repository.It represents the tested and production-ready version of your project. Code on the "main" branch is expected to be in a deployable state.
- The "dev" branch, short for development, serves as an intermediary between feature branches and the main branch. It's where ongoing development work occurs. Developers often merge their feature branches into the "dev" branch to test and integrate new features collectively. The "dev" branch is not considered as stable as the "main" branch, but it should still be in a usable state for testing purposes. Once all planned features for a release have been integrated and tested on the "dev" branch, it can be merged into the "main" branch to prepare for a new release or increment in scrum.
- Feature branches are created for specific features, bug fixes, or tasks. Each feature branch focuses on a single piece of work. Feature branches can run in parallel and When a feature is complete and tested, the changes in the feature branch are merged back into the "dev" branch for further integration testing. Feature branches are typically short-lived and may be deleted after their changes have been merged.

Creating main and dev branches

On github repo page

- Settings and branch protection rule
- Add a branch protection rule
- Branch name pattern is main so rule applies to main only
- Select protecting requirements for main
    - Require a pull request before merging
    - Require approvals from a specified number of collaborators - preferably 2 collaborators must approve merges from other branches into main (verify all tests have been successful)

Branch Protection Rule: Branch protection rules are security measures in Git hosting platforms that enforce restrictions on critical branches, like "main" or "master," to prevent unauthorised or accidental changes and ensure code quality and stability through requirements like code reviews and limitations on force pushes.

Branch Name Pattern: Branch name patterns are conventions or rules used to name branches consistently within a Git repository, helping teams organise and manage their work by providing a clear and predictable structure for branch names, such as prefixing feature branches with "feature/" or bug fix branches with "bugfix/."

Create and configure dev branch

- Git branch Lists all branches in the repository.
- Git checkout -b dev Creates and switches to a new branch with the given name.
- Git add, commit and git push origin dev is used to push the local "dev" branch to a remote Git repository named "origin." This command updates the remote "dev" branch with the changes from your local "dev" branch, effectively synchronising the branches.
- Set up branch protection rules for dev branch
    - Require a pull request before merging
    - Request approvals from n number of reviewers
- Go to General
    - Specify default branch to dev

<u>Using feature branches</u>

- Collaborators can create their own feature branches
- Collaborates can clone the repository

- Create a new repository
- And clone using clone URL to clone the repository locally
- Git branch shows on dev branch as its the default branch
- To create a new feature branch for each team member
    - Git checkout -b manny-user-stories
    - Git checkout -b <new-branch> always copies the content form the current branch which is dev in this case
    - Git checkout -b jess-improve-ui
    - Update github with branches created by each team member
        - Git commit and Git push origin manny-user-stories
        - Git commit and Git push origin jess-improve-ui
        - Each member uses personal access token which is specific to github account

Team members can create their own branches
- First clone dev
- Make a new branch using git checkout

Use meaningful branch names
- Based on the feature
- Based on the team members name

When finished or a milestone reached, push

<u>Pull requests</u>

A pull request is a request made by a developer to merge changes from their branch (the feature branch) into another branch (the "target" branch), typically the main or development branch. It serves as a mechanism for code review, discussion, and collaboration among team members, ensuring that code changes meet quality and coding standards before being merged into the target branch.

Collaborator can merge their feature branch with dev
- The branch is 1 commit ahead
- Pull request page allows comparing of changes
- Able to merge shows branches can be automatically merged
- Create a pull request - Creating a pull request means submitting a formal request to merge your code changes from a source branch into a target branch, usually for code review and inclusion in the main project.
- Assign reviewers which requests reviews from other collaborators
- Collaborates can comment, approve, or request changes
- Once approved, any team member can merge pull request
- If branch is no longer required, delete the branch
- Merging to branches with no branch protection rules can be merged freely

Pull requests

- When you're ready to merge to dev
    - It depends on the circumstances
- If the branch is only commits behind dev
    - Just create a pull request
    - Assign reviewers
    - Once all approved, merge the changes (often merges automatically)
- If the branch is commits behind and ahead
    - Make a pull request to merge latest dev into your branch (get the latest dev by pull request from dev to our feature branch which has no protection rules so can be freely merged
    - Then make another pull request to merge the other way from our feature branch to dev
    - Follow the approval process as before
    - Merge the changes

Resolving merge conflicts

Merge conflicts happen when Git cannot automatically combine changes from different branches due to conflicting edits to the same part of a file, requiring developer intervention to decide which changes to keep.

- When 2 developers edit the same lines of code in a file, git cannot automatically merge
- Resolve conflicts locally in the development environment, ensure test still pass before resolving conflicts and resubmit changes
- Switch to the dev branch
    - Git pull to get latest version
    - View the file to see file changes
    - Switch to own feature branch and merge locally into dev branch (conflict message shows automatic merge has failed
    - In editor Head shows where conflict exists
    - Edit the file to remove the conflict
    - Git add and commit and push to own feature branch (use fix conflict commit message)
    - Now branch is commits ahead of dev allowing us to merge as github has identified the merge conflict
    - Once collaborators have approved the pull requests, merge branch and delete feature branch

<u>Merge conflicts</u>

- Hard to avoid merge conflicts entirely
    - The GitHub branch protection system helps
- Easiest to resolve locally
    - Use the code editors and communicate with team mates to discuss and resolve conflicts
    - Repeat the pull request process once resolved
- How to reduce merge conflicts
    - Good communication
    - Let everyone in the team know which files you will work on
    - Pull down latest version of branches at regular intervals
    - Have a standard process which everyone follows

<u>Workflow</u>

One team member is the Git Auth
They are responsible for:

- Creating the repo on their own GitHub account
- Create a new dev branch
- Adding branch protection rules to the main and dev branches
- Requiring reviewers for merging to main or dev
- Creating the initial folder structure for the project
- Setting the dev branch as the default

Team members should

- Clone project locally
- Create their own feature or developer branches with suitable names
- Develop all code on these branches
- Never code directly on main or dev
- Commit locally at frequent intervals
- Push updates to github

Ready to merge

- Update your-feature-branch
    - Git branch dev
    - Git pull
- Get latest changes from the dev branch
    - Locally switch to dev branch
    - Sync with the remote dev branch on the github using git pull
    - Resolve merge conflicts and commit locally

Merge feature branch to dev

- Merge the latest changes in dev with your-feature-branch
    - Switch to your-feature-branch and merge from dev
    - Resolve merge conflicts and commit locally
    - Check everything still builds and tests pass
    - Commit any changes
    - Push your-feature-branch to github

Pull request

- Open a pull request on github
    - Choose the reviewers
    - Reviewers inspect your updates
        - May add comments
        - May approve the request
- If everything is approved which most likely will be automatically merged
    - Complete the pull request
- If changes are required
    - Make them locally on your-feature-branch
    - Check that test pass, commit, and push to remote
    - Complete the pull request once approved

Finish uo

- Update your local dev branch
- Locally switch to dev and sync with github (git pull again)
- Either
    - Switch back to your-feature-branch and merge from local dev, or
    - Create a new branch for your next pierce of work based on local dev

Git Workflow for Collaborative Development

Creating a new project

One team member (the Git Auth) makes a GitHub repository:

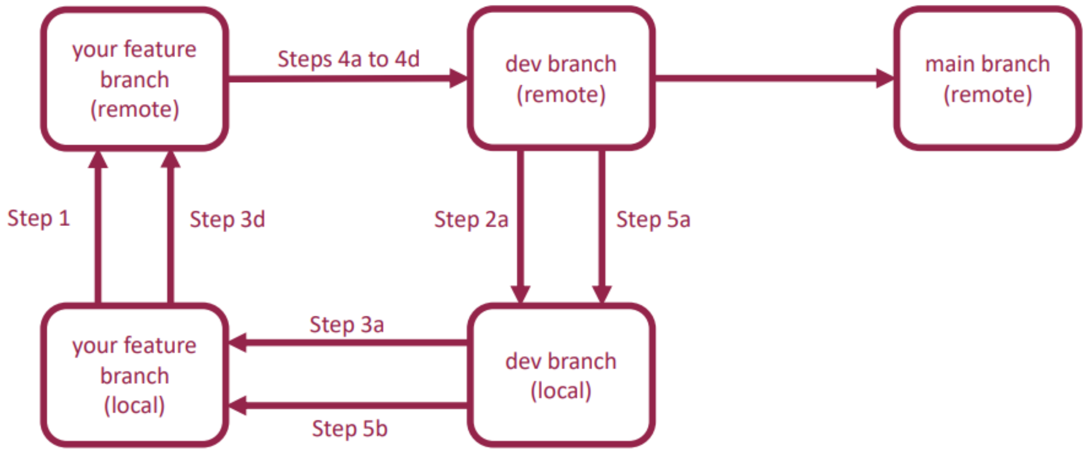
- Go to your GitHub account and select the Repositories page
- Click the New button to make a new repository
- Add a README and a .gitignore file with the appropriate template
- Go to the repository on your GitHub account and select the Settings tab
- Manage access – add your collaborators to the project
- Branches:
    - You should be on the main branch
    - Add a branch protection rule with the same name (main)
    - Select "Require a pull request before merging", "Require approvals" and 2 approvals before merging and save changes
    - Create a dev branch based on the main branch
        - You will normally merge your code changes into the dev branch and push this code to main at intervals when you have a stable build (eg after each sprint)
    - Update the default branch to dev
    - Add a branch protection rule for the dev branch, normally like the one for main

Working on the project

- One pair of developers can set up the initial code for the project and folder structure and push it to dev and main before adding the protection rules
- Team members can now clone the project locally
- Do NOT edit the code in the main branch or dev branch, instead:
    - Pull the dev branch to your local machine
    - Create a feature branch locally; give it your own name, and/or the name of the feature you are working on (eg woody_roundup)
    - git checkout –b woody_roundup
    - Develop your code on this branch

Seeing remote branches

- If you can't see someone else's remote branch:
    - Open a command window in the top directory of your solution
    - git status – should show you which branch you are on and if there is anything to commit
    - git fetch origin – fetches references to any new remote branches
    - You still need to pull down a local copy of a branch if you want to use it.



See next page for step descriptions

Step 1

Commit the changes in your local branch regularly, and push them to remote

Step 2

When you are ready to merge to dev, first update your feature branch to include the latest changes from dev:
a) Locally, switch to dev branch and sync with the server (git pull)
b) If there are any merge conflicts, resolve them and commit locally

Step 3

Merge the latest changes with your feature branch:
a) Locally switch to your feature branch and merge from dev
b) If there are any merge conflicts, resolve them and commit locally
c) Check everything still builds and tests pass; commit any changes
d) Push your branch to remote

Step 4

Pull request:
a) Create a pull request
b) Fill in the code reviewer(s)
c) The reviewer may add comments and may approve the request
d) If approved and no changes, complete the pull request
e) If changes required, make them locally, make sure the code still compiles and runs, commit and push to remote; complete pull request once approved
f) If you are finished with this feature branch, delete it

Step 5

Update your local dev branch:
a) Switch to dev and sync with the server (git pull)
b) Switch back to your feature branch and merge from local dev, or create a new branch for your next piece of work, based on local dev