Git Gud

Git, Project Management and You

Before we begin...

- This mini-presentation will have a lot of information about a lot of things
- *Tons* of documentation links will be provided at the end
- Don't worry if you're lost: the slides are pretty complete and will be uploaded soon-ish
- Feel free to ask questions on #prog, a fellow student (or a bored ACDC) may jump in to help you! Just don't ping us 😉

The Problem

I want to...

- have a full history of my project
- share my projects with others
- experiment with my code safely

A Solution

Copy/pasting into multiple folders? That's

- ❌ Wasteful
- ❌ Requires a lot of manual actions
- ❌ Accidents are a click away...
- ❌ How do I even share my project?
- ❌ We're developers, we're lazy!

Another Solution

Using a cloud service like Mega or Google Drive? That's

✅ A bit more efficient
❌ Still requires actions (and more copy pasting)
❌ *Still* error-prone (Delete button go brrr)
✅ Sharing is possible

Still not fantastic...

A Better Solution

A proper *versioning system*, like Git!

✅ Efficient
✅ Does a lot in a few commands
✅ Hard to mess things up (unless you *really* try)
✅ Easily share your projects

What is Git?

> Git (/gɪt/) is a **distributed version-control system** for tracking changes in any set of files, originally designed for coordinating work among programmers cooperating on source code during software development. -- *Wikipedia*

- Distributed: We'll see that later
- Version-control system: Allows us to *version* our code

Versioning

We want to...

- Store a full *history timeline of our project*
- Tag parts of the timeline (like "versions")
- Even have alternate timelines!
- Let's use Git for all of these!

Git repository

A repository is "a folder where Git tracks stuff". Git...

- ... tracks all changes in that repository
- ... keeps a full history of what happened
- ... is able to "push" to and "pull" from other repositories (even remote ones!)

A simple example

You already know a lot about Git...

```
# Create a Git repository
$ mkdir hello
$ cd hello
$ git init
# Write stuff in a file
$ echo Hi! > file.txt
# Tell Git to "track" this file
$ git add file.txt
# Create a commit
$ git commit -m "Added my file"
```

So, what happened?

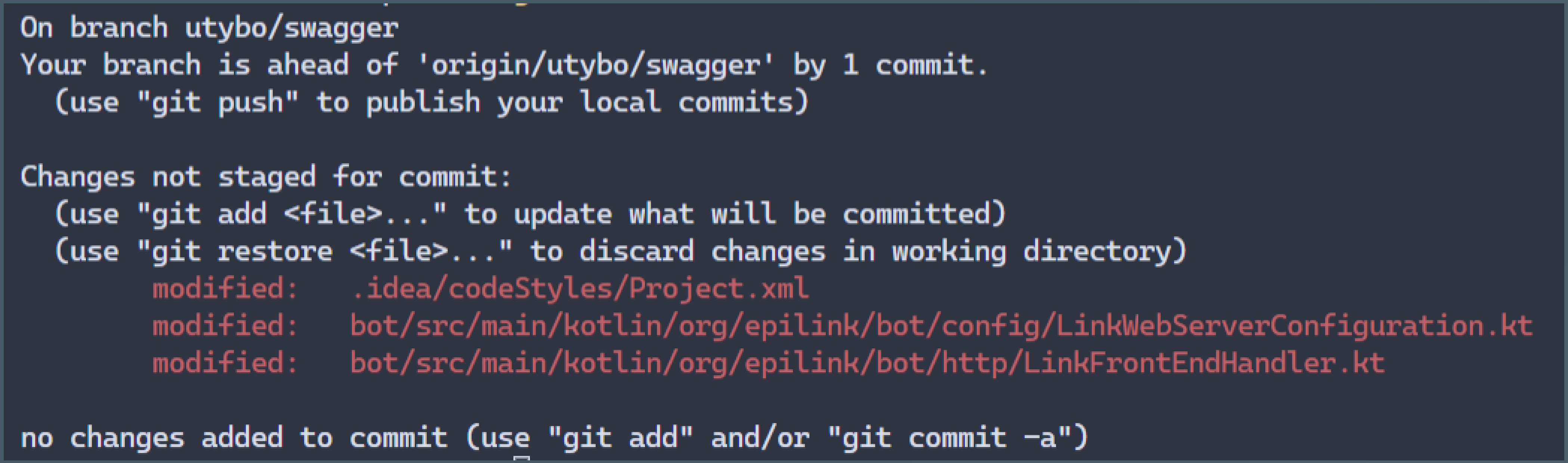
- We created an empty repository with `git init`
  - `git clone` copies a repository from somewhere else
- We told Git: "hey, I want you to care about this change"
- We created a commit, a "checkpoint" on our timeline
  - This checkpoint stores a lot of information, such as the author, dates, etc.
  - Checkpoints only contains the actual changes. This is what makes Git efficient: store changes instead of entire file copies.

Understanding what's going on

From your point of view, Git may look like a "black box". Let's make it clearer using some built-in commands!

git status

`git status` gives you an overview of what's going on in your repository



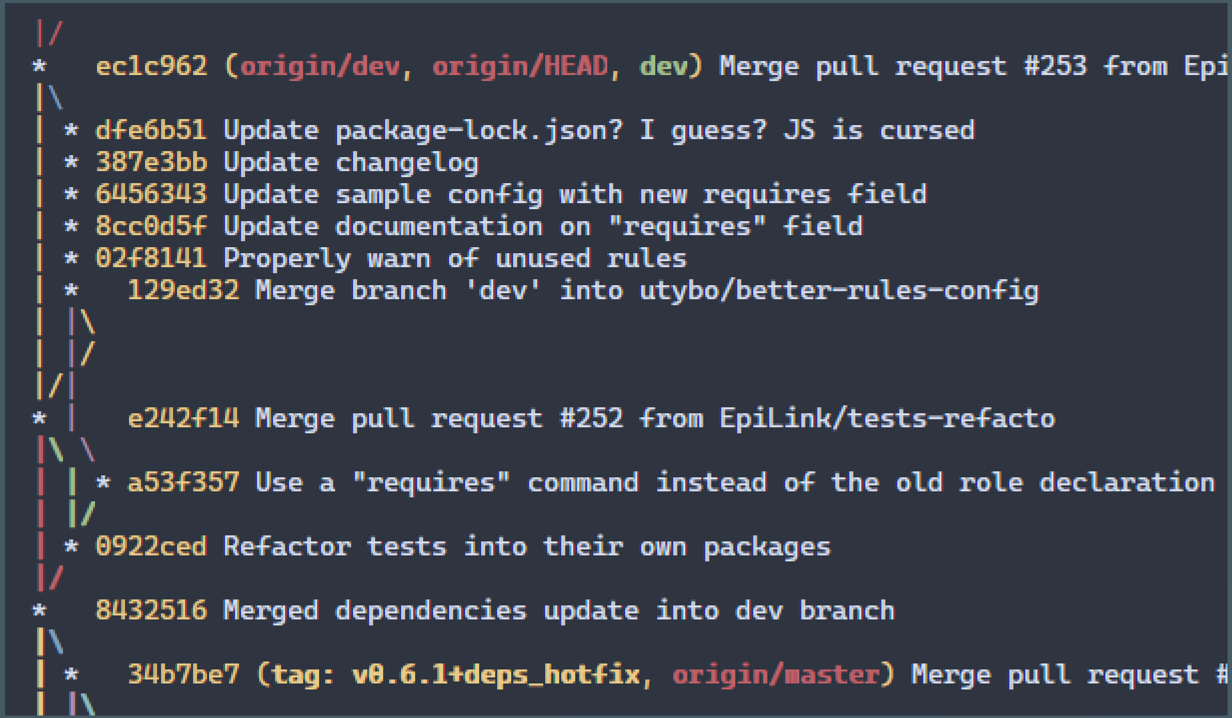
```
On branch utybo/swagger
Your branch is ahead of 'origin/utybo/swagger' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .idea/codeStyles/Project.xml
        modified:   bot/src/main/kotlin/org/epilink/bot/config/LinkWebServerConfiguration.kt
        modified:   bot/src/main/kotlin/org/epilink/bot/http/LinkFrontEndHandler.kt

no changes added to commit (use "git add" and/or "git commit -a")
```

git log

- `git log` to see the timeline of what's going on
  - `git log --oneline --graph --all` to get a nice graph view



```
|/
*   ec1c962 (origin/dev, origin/HEAD, dev) Merge pull request #253 from Epi
|\
| * dfe6b51 Update package-lock.json? I guess? JS is cursed
| * 387e3bb Update changelog
| * 6456343 Update sample config with new requires field
| * 8cc0d5f Update documentation on "requires" field
| * 02f8141 Properly warn of unused rules
| *   129ed32 Merge branch 'dev' into utybo/better-rules-config
| |\
| |/
|/|
* |   e242f14 Merge pull request #252 from EpiLink/tests-refacto
|\ \
| | * a53f357 Use a "requires" command instead of the old role declaration
| |/
| * 0922ced Refactor tests into their own packages
|/
*   8432516 Merged dependencies update into dev branch
|\
| *   34b7be7 (tag: v0.6.1+deps_hotfix, origin/master) Merge pull request #
| |\
```

So far...

- `git init`: create a Git repository in the current directory
- `git clone`: get a Git repository from somewhere else and copy it locally.
- `git add`: Tell Git "I want these changes in my next commit"
- `git commit`: Create a commit
- `git log` and `git status`
- Also, remember to use `.gitignore` files! List one pattern per line: Git will act as if these files/folders do not exist.

Alternate timelines

Alright, cool, we have our timeline, but I want to go further.

- I'd like to be able to work on "my own timeline", without impacting the "main timeline".
- I'd also like to "reconcile", "merge" the main timeline and my timeline when I'm done
- Hey, let's take it further: the main timeline is our production line, all experiments are done in other timelines and merged into the main one when ready.

Branches

We can have timelines *in parallel*. Breaking the space-time continuum, hooray!

- You can create a branch from any point in your timeline(s) (1)
- You can merge two diverging branches (2). A "merge commit" (3) is created on the "receiving" branch. The merged branch can still be used after the merge (it does not "terminate" the branch). (4)

```
                                          (3)
(1) ----o-x-----o---->          (2) -o-----o----o--->       o   = commit
           \                              /                --- = branch
            ---o---o->              --o--x---o----o->
```

Example: main/dev workflow

```
         Merge                Merge   Merge
main ------o--------------------o-------o----> main
          /                    /       /                   o  = commit
dev  -o--x-----o-------o------x---o---x------>  dev       --- = branch
               |   Fix stuff     Meh
           Add thing
```

`main` (or `master`) is sacrosanct. `dev` is where active work happens.

Example: main/dev/feature workflow

```
           M                             M
main ------o-----------------------------o----> main     o  = commit
          /   M              M  M       /               --- = branch
dev  -x--x----o-x----x-------o--o--x---x------>  dev    f_  = feature branch
       \     /   \    \     /  /    \                    M  = merge commit
     f1 -o--x     \ f3 --o-x  /   f4 ---o--o-->
                f2 -o---o--o-x
```

- Clean `main` (or `master`) branch, the latest version
- Clean `dev` branch, the current WIP version
- Feature branches (e.g. `add-this`, `zoroark/fix-bug`, ...)

Using branches with Git

Your repository is always somewhere at one of the timelines. You can change which timeline you are on using various commands.

- `git branch NAME`: Create a branch named NAME from where I am
- `git switch NAME`: Switch to the branch named NAME
- `git merge ONE --into TWO`: Merge branch ONE into branch TWO
    - e.g. `git merge zoroark/fix-bug --into dev`

Sharing your repository

- Your repository can live on many other computers or servers ("distributed", remember?).
- This is done using "remotes". A remote is just a version of the repository that lives somewhere else. This will generally be on a server somewhere (like the one you use for your TPs).
- You `git clone` from a remote. Git automatically adds the URL you cloned from as a remote (generally named `origin`).
- You can have multiple remotes.

Remote operations

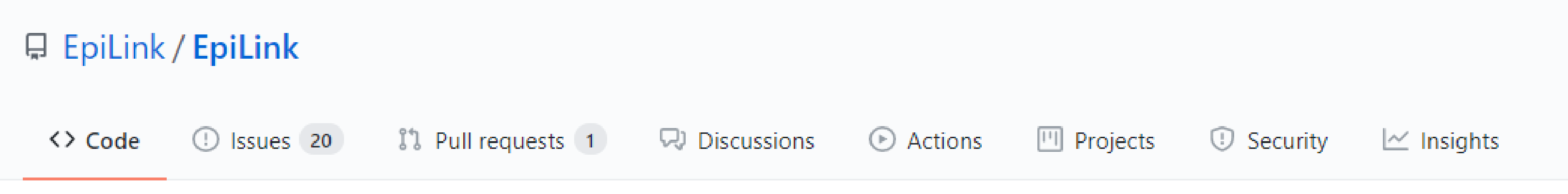
There are 3 main operations: pushing, pulling and fetching.

- Pushing (`git push`): sends your changes on your local branch to the remote's version of the branch.
- Pulling (`git pull`): opposite of pushing, retrieves changes on the remote and applies them to your local version.
- Fetching (`git fetch`): retrieves the changes from the remote but does not apply them on your branches. This is useful because the remote's branches are actually stored as *separate branches*; pulling just merges them automatically for you.

Forge

> [...] A forge is a web-based collaborative software platform for both developing and sharing computer applications. [...] For software developers it is a place to host, among others, source code (often version-controlled), bug database and documentation for their projects. *-- Wikipedia*

While not mandatory, they are an essential tool for all of your projects, even personal ones.



Forges provide a wide array of features, such as:

- Code hosting (Git server/remote)
- Bug and task tracking (Issues, projects, issue tags, kanbans, etc.)
- Release management (Releases, milestones)
- Forums (Discussions)
- CI/CD (GitLab CI, GitHub Actions)
- Security alerts and vulnerability disclosure
- Code statistics

Popular forges

- GitHub (owned by Microsoft) https://github.com
- GitLab (independent) https://gitlab.com

Both provide a similar array of features for S2 projects. Note that GitLab is more flexible for free *private* repositories.

Forges support private (only available to you and people you select) and public (everyone can access it) repositories.

Issues

An issue is a discussion thread about a bug, feature request, question or, more generally, a "task". Issues are very versatile and useful for planning your work.

- You can use tags, such as "bug", "high priority", or "area: graphics"
- You can use milestones to group tags into versions, i.e. saying X tasks should be done for Y version.
- Issues can be opened (meaning they are active) or closed (meaning they are resolved).

Example: https://github.com/EpiLink/EpiLink/issues/243

Branches on forges

- In order to avoid tons of conflicts, you really should use branches when using forges.
- Merging on collaborative projects is a bit different.
- Pull Requests (or Merge Requests on GitLab) are like civilized `git merge` commands.
    - They offer comments, tags, review tools, etc.
- Once all checks are green, GitHub or GitLab will do the merge for you after you click the big ol' *Merge* button.

Example: https://github.com/EpiLink/EpiLink/pull/198

References and documentation

- **Git Book:** https://git-scm.com/book/en/v2
  - Official Git book, has a ton of in-depth information
  - Links: Commits, Remotes, Branches
- **Git Workflows:** Ways to organize your Git repository
  - From the Git Book itself
  - Master + Topic branches = GitHub Flow
  - GitFlow: original blog post, re-explanation from BitBucket. A *very* in-depth workflow. Quite overkill for 90% of uses.

References and documentation (cont.)

- **GitHub:** Official GitHub documentation, Quickstart, Intro/Ad video
- **GitLab:** Getting started
- **My own tutorial:** *Far* from complete, but covers the basics. Link
- **Website:**
  - Want a website for the project? Check out Hugo and Jekyll
  - GitHub Pages and GitLab Pages allow you to host your website directly from your repository

That's all!