Project Version Control with Git

During the life of a project you will implement new features and continuously make changes to the working directory by adding, deleting, renaming and editing source code and other files.

Implementing a feature request

A feature request is a requirement to modify or add functionality to an application or website.

Open pilots.html to add "Delta"

Open cities.html to add "Detroit"

Open names.html to add "David"

Updating the local repository

The reason why Git uses two phases - staging and commiting - to update repository is that by staging first you can ***group all related changes into a single commit*** and give it a meaningful comment.

View from Visual Studio Code

A graphical vision of Git commit

Committing letter D

Viewing Project History

What is a Branch?

A branch represents an independent line of development in a project with its own separate history of commits.

You can list the branches in the local repository by running the git branch command. The --remote switch shows the local copy of the branches in the remote repository on GitHub.

Branch

Right now our repository has only a single branch called **main** both locally and remotely.

When you first setup a repository Git creates the **main** branch automatically for you.

All Git projects have a main branch by default.

The image shows a 'main' trunck representing the main branch.

Each commit is a circle. The arrow symbolizes the HEAD pointer.

The branch pointed to by HEAD is the current or checked out branch.

HEAD -> main It means that HEAD is currently pointing to the main branch.The contents of the working directory reflect the last commit on the checked out branch plus any changes you have made.

Comparing Versions

It is important at this point to note that there are ***several versions*** in Git project.

1-The **working directory** version (the one you use for editing)

2-The **staged** version(after you run **git add** to add the file to the index for the next commit)

3-The **commited** versions (one version for each commit)

The **git diff** command shows changes between the working directory, index and commit versions.

Exercise

Feature request: ***add the letter E*** to the phonetic website.

1 pilots.html

- ***Echo***

2 cities.html

- ***Eldorado***

3 names.html

- ***Eva***

Check status and add files to index

Exercise

Feature request: ***add the letter F*** to the phonetic website.

1 pilots.html

- ***Foxtrot***

2 cities.html

- ***Fillmore***

3 names.html

- ***Fred***

Let´s check the status after letter 'F' added

After checking the status we have now two different versions of the HTML files.

One version contains the staged changes (letter D). The other one contains the unstaged changes we did adding letter 'F'.

And of course we also have the commited versions as show by **git log** in the next slide.

Let´s check git log and viewing unstaged changes by git diff

Interpreting the output of the `git diff pilots.html` command line by line.

1. `diff --git a/pilots.html b/pilots.html`: This line indicates that the `git diff` command is comparing two versions of the file `pilots.html`. The `a/pilots.html` represents the original file, while `b/pilots.html` represents the modified file.
2. `index fb4318f..32706f8 100644`: This line provides information about the file index. It shows the commit hashes (`fb4318f` and `32706f8`) of the original and modified versions of `pilots.html`. The `100644` represents the file mode, which is typically associated with regular files.

3. `--- a/pilots.html`: This line indicates the start of the section related to the original version of the file (`a/pilots.html`).
4. `+++ b/pilots.html`: This line indicates the start of the section related to the modified version of the file (`b/pilots.html`).
5. `@@ -22,6 +22,7 @@`: This line is the unified diff header. It provides information about the line numbers and context for the changes that follow:
    - `-22,6`: This means that the change begins at line 22 in the original file and affects 6 lines.
    - `+22,7`: This means that the change begins at line 22 in the modified file and affects 7 lines. The `+` indicates that a line has been added.

Viewing changes between the index and the last commit

For this we have to use the switch **--cached** .

Viewing changes since the last commit

Viewing changes between any two commits

We can use the short hash obtained from git log to view the difference between any two commits

Let´s commit letter 'E', git add for letter 'F' and commit.

What if you want to undo changes

One of the reasons to track a project´s history is to have the ability to **undo** changes.

It is a common situation in software development: you change somenthing, you test the change and it does not quite work the way you expected.

You then decide to revert the code back to the previous version.

With Git you can easily accomplish that.

There are three possible scenarios in Git

1 - Undoing changes in the working directory before staging

2 - Undoing changes after staging and before committing

3 - Undoing committed changes

Exercise : Unwanted Change

Let´s demonstrate the three scenarios using the Phonetic Website.

Open **pilots.html** and delete all words except "Alpha".

Now we will learn how to recover the lost words using **Git** undo features.

The ideia is that you have to change dozens of lines of code in various parts of the source file.

Undoing Changes Before Staging

To recover the lost words in **pilots.html** all you have to do is to revert the file to its last '**modified**' version using the **git checkout** command:

Browse the website to verify that the words have been recovered.

Undoing Changes After Staging

Open **pilots.html** and delete all words except "Alpha".

After running git status it shows that pilots.html is ready to be committed.

It also suggests to run **git reset HEAD (file)** to un-stage the change.

Git restore x Git reset

Both `git restore` and `git reset` are used to undo changes in your repository. However, they work differently.

`git restore` is used for restoring files in the working tree from the index or another commit. This doesn't update the branch. On the other hand, `git reset` is used to update your branch.

So, if you want to undo changes that you have made to a file in your working directory but haven't staged yet, you can use `git restore`. If you want to undo changes that you have made to a file and have already staged it, you can use `git reset`.

The **git reset** command has removed the file from the index (staging area) however the unwanted change is still in the working directory.

To recover the lost words, you still need to repeat the process for undoing un-staged changes and run **git checkout** to restore the last **'modified'** version.

Browse the website to verify that the words have been recoverd.

Undoing Committed Changes

Open **pilots.html** and delete all words except "Alpha".

Now we-re going to commit the 'unwanted change'.

Committing "unwanted changes"

Undoing the committed change running git revert using the short hash of the unwanted commit.

Explaining the `git revert 032aca5 --no-edit` command:

- `git revert`: This is a Git command used to create a new commit that undoes the changes introduced by a previous commit. It's a way to reverse changes without removing commit history.
- `032aca5`: This is the commit hash or identifier of the commit you want to revert. In this case, you are specifying a commit with the hash `032aca5` as the target.

Continue

- `--no-edit`: This is an option used with `git revert`. When you use `--no-edit`, it tells Git to create the revert commit without opening the default text editor for you to edit the commit message. Instead, Git will automatically generate a commit message for the revert based on the commit you're reverting.

So, when you run `git revert 032aca5 --no-edit`, Git will create a new commit that undoes the changes introduced by the commit with the hash `032aca5`, and it will do so without asking you to edit the commit message. Git will generate a default commit message indicating that this new commit is a revert of the specified commit. This is useful when you want to quickly create a revert commit without modifying the commit message.

Checking the project history

Tagging Versions

You can use Git to attach a tag to easily identify a stable version of a project with **git tag** command.

1- **Create a Tag:**

To create a tag, you can use the git tag command followed by a tag name. For example, if you want to tag a version 1.0 of your Python project, you can do:

```
git tag 1.0
```

2- **List Tags:**

To see a list of tags in your repository, you can use:

```
git tag
```

3- **Tagging Commits:**

You can also tag specific commits by specifying the commit's SHA-1 hash:

```
git tag -a v1.1 <commit-SHA-1>
```

This creates an annotated tag with a message. Annotated tags are recommended for versioning.

Tag - Continue

4 - **Push Tags:**

Tags are not automatically pushed to remote repositories when you push changes. To push tags, you can use:

```
git push origin --tags
```

This sends your tags to the remote repository.

5 - **Checkout Tags:**

You can check out a specific tag to view the code at that version:

```
git checkout v1.0
```

6 - **Delete Tags:**

To delete a tag, use the -d option:

```
git tag -d 1.0
```

7 - **To delete a tag on the remote repository:**

```
git push origin --delete 1.0
```

Tagging sample

Checking status and update GitHub

Commit Messages for Effective Version Control

Special script to add to '.bashrc' file

```
# Your existing configuration settings

# Start SSH agent if not already running
```bash
if [ -z "$SSH_AUTH_SOCK" ]; then
    eval $(ssh-agent -s)
    ssh-add
fi
```

Other configurations and aliases if applicable
```