Git - Versioning and Collaboration

Software Engineering - Tutorial

Dr. Antonio Bucchiarone - bucchiarone@fbk.eu

Academic year 2023/2024

Tutorial Lectures of this course

- Git - Versioning and collaboration with Git and Github
- Database: MongoDB - Mongoose
- Backend: JavaScript- WebAPI - NodeJS - Express - apiary.io (WebAPIs documentation)
- FrontEnd - TypeScript/JavaScript
- Testing - Jest
- CI/CD - Heroku/Vercel

Contents of today class

- Git basics
- Versioning
- Branching
- Collaboration

Tools and references

- CLI (https://git-scm.com/downloads)
- GitHub (https://github.com)
- Visual Studio Code (https://code.visualstudio.com/)
- Tutorials Slides and Material: https://github.com/antbucc/IS-23_24
- IS Organization: https://github.com/IS2324

What is version control?

Version control systems (VCSs) are a category of software tools that help a software team manage changes to source code over time:

- keeps track of every modification to the code
- developers can turn back the clock and compare earlier versions
- each developer may make their changes in several parts of the file tree

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting.

Versioning

If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code.

- How much do you save?
- How do you name these versions?
- How do you know what exactly is different in these versions?

Collaboration

Without a VCS in place, you're probably working together in a shared folder on the same set of files. Shouting through the office that you are currently working on file "xyz" and that, meanwhile, your teammates should keep their fingers off is not an acceptable workflow

Restoring Previous Versions

- If the changes you've made lately prove to be garbage, you can simply undo them in a few clicks
- this should make you a lot more relaxed when working on important bits of a project

Understanding What Happened

- short description of what was changed
- Additionally (if it's a code / text file), you can see what exactly was changed in the file's content

Git Basics

Step 1: Set up your GitHub.com account

For this class, we will use a public account on GitHub.com. We do this for a few reasons:

```
- We don't want you to "practice" in repositories that contain real code
- We are going to break some things, so we can teach you how to fix them (therefore, refer to the bullet above)
- You can set up your free account by following these steps:
```

1. Access https://github.com/ and click Sign up
2. Choose the free account
3. You will receive a verification email at the address provided
4. Click the link to complete the verification process

If you already have an account, verify that you can visit github.com within your organization's network.

Step 2: Install Git

- Git is an **open source** version control application.
- You will need *Git* installed for this class.
- Open Terminal if you are on a Mac, or PowerShell if you are on a Windows machine, and type:

```
$ git version
```

- You should see something like this:

```
$ git version
git version 2.33.1
```

- I recommend you stay within two minor versions of the latest version of Git. The minor version is the second number in the series x.x.x.

Downloading and installing Git

- If you don't already have Git installed, you can download Git at https://www.git-scm.com/

Where is your shell?

Now is a good time to create a shortcut to the command-line application you will want to use with Git:

- If you are working on **Windows**, **Git for Windows** provides a **BASH emulation** used to run Git from the command line.
- If you are working on **macOS** or another **Unix-like system**, you can use the built-in Terminal application.

Step 3: Try cloning with HTTPS

- Open your chosen shell, and type:

```
$ git clone https://github.com/IS2324/Tutorial.git
```

If the clone is successful you'll see:

```
$ git clone https://github.com/IS2324/Tutorial.git
Cloning into 'Tutorial'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

> **NOTE**: many corporate networks restrict SSH traffic, so we highly recommend using HTTPS and verifying the clone works. Also, if you have two-factor authentication enabled and wish to use HTTPS, you will need to set up a personal access token.

set up a personal access token

Proxy configuration - I

- If your organization uses a proxy.
- Open **Git Bash** (on Windows) or **Terminal** (on Mac or *nix) and complete the appropriate steps below:

If your proxy does not require authentication:

```
$ git config --global http.proxy https://YOUR.PROXY.SERVER:8080
```

Replace YOUR.PROXY.SERVER with your proxy's URL.

Proxy configuration - II

If your proxy does require authentication:

```
$ git config --global http.proxy https://YOUR_PROXY_USERNAME:YOUR_PROXY_PASSWORD@YOUR.PROXY.SERVER:8080
```

Replace YOUR_PROXY_USERNAME with the username used to authenticate into your proxy, YOUR_PROXY_PASSWORD with the password used to authenticate into your proxy, and YOUR.PROXY.SERVER with your proxy's URL.

Step 4: Set up your text editor

- You can use almost any text editor, but we have the best success with the following:

Visual Studio Code

Notepad

Vi or Vim

Sublime

Notepad++

GitPad

If you do not already have a text editor installed, go ahead and download and install one of the above editors now!

Getting started with collaboration

What is GitHub?

- GitHub is a **collaboration platform** built on top of a distributed version control system called **Git**.
- GitHub is focused on **developers**, the people who code and create software.

GitHub concentrates on three things:

- Building a technology platform that is like no other, on which developers can **create**, **share** and **grow** the best code possible.
- Nurturing a **community for developers**; a safe and collaborative place that facilitates sharing, amplifies creativity, and supports the principles of open source.
- Opening up a **community of opportunity**, where new developers can be born and where experienced developers can hone their skills and expand their knowledge.

What is GitHub?

In addition to being a place to host and share your Git projects, GitHub provides a number of features to help you and your team collaborate more effectively. These features include:

- Issues
- Pull Requests
- Projects
- Organizations and Teams

Exploring a GitHub repository

- A **repository** is the most basic element of GitHub. It is easiest to imagine as a **project's folder**. However, unlike an ordinary folder on your laptop, a GitHub repository offers simple yet powerful tools for collaborating with others.
- A repository contains all the project files (including documentation), and stores each file's **revision history**. Whether you are just curious or you are a major contributor, knowing your way around a repository is essential!

User accounts vs. organization accounts

- There are two account types in GitHub, **user accounts** and **organization accounts**.

User accounts

- When you signed up for GitHub, you were automatically given a user account.
- Permissions for a user account are simple, you add people as collaborators to specific repositories to give them full read-write access to the project.

Organization accounts

- Organization accounts provide more granular control over repository permissions. - In an organization account you create **teams of people** and then give those teams access to **specific repositories**.
- Permissions can be assigned at the team level (e.g, read, write, or admin).

Repository navigation

Code

The Code tab is where you will find the files included in the repository. These files may contain the project code, documentation, and other important files.

Issues

Issues are used to track **bugs** and **feature** requests. Issues can be assigned to specific **team members** and are designed to encourage discussion and collaboration.

Pull requests

A Pull Request represents a **change** (**adding**, **modifying**, or **deleting**) files. Pull Requests help you write better software by facilitating code review.

Projects

Projects allow you to visualize your work with Kanban style boards. Projects can be created at the **repository** or **organization** level.

Wiki

Wikis in GitHub can be used to communicate project details, display user documentation, or almost anything your heart desires. And of course, GitHub helps you keep track of the edits to your Wiki!

README.md

It is a special file that we recommend all repositories contain. GitHub looks for this file and helpfully displays it below the repository. The README should explain the project and point readers to helpful information within the project.

CONTRIBUTING.md

It is another special file that is used to describe the process for collaborating on the repository. The link to the CONTRIBUTING.md file is shown when a user attempts to create a new issue or pull request.

ISSUE_TEMPLATE.md

It is used to generate templated starter text for your project issues. Any time someone opens an issue, the content in the template will be pre-populated in the issue body.

Using GitHub Issues

In GitHub, you will use issues to record and discuss ideas, enhancements, tasks, and bugs. Issues make collaboration easier by:

- Replacing email for project discussions, ensuring everyone on the team has the complete story, both now and in the future.
- Allowing you to cross-link to related issues and pull requests.
- Creating a single, comprehensive record of how and why you made certain decisions.
- Allowing you to easily pull the right people into a conversation with @ mentions and team mentions.

Activity: Creating a GitHub issue

- Follow these steps to create an issue in the class repository:

1. Click the Issues tab.
2. Click New Issue.

Activity: Creating a GitHub issue

3. Type the following in the Subject line: YOUR-USERNAME Workflow

4. In the body of the issue, include the text below:

```
- [ ] Create a branch
- [ ] Edit the file
- [ ] Commit the changes
- [ ] Create a Pull Request
- [ ] Request a Review
- [ ] Make more changes
- [ ] Get an approval
- [ ] Merge the Pull Request
```

5. Click the Submit new issue button to open your issue.

Using Markdown

GitHub uses a syntax called **Markdown** to help you add basic text formatting to Issues, Pull Requests, and files with the .md extension.

Commonly used Markdown syntax

Header

The # indicates a Header. # = Header 1, ## = Header 2, etc.

* List item
A single * or - followed by a space will create a bulleted list.

Bold item
Two asterisks ** on either side of a string will make that text bold.

- [ ] Checklist

A - followed by a space and [ ] will create a handy checklist in your issue or pull request.

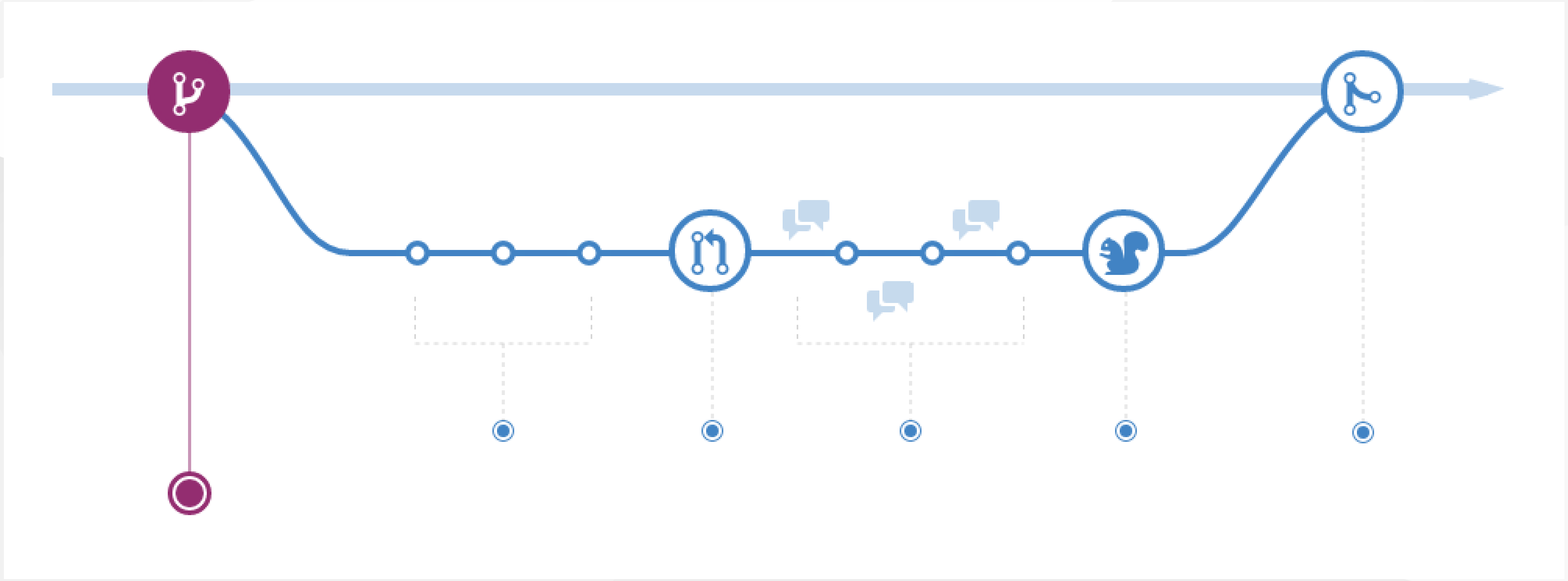
@mention

When you @mention someone in an issue, they will receive a notification - even if they are not currently subscribed to the issue or watching the repository.

#975

A # followed by the number of an issue or pull request (without a space) in the same repository will create a cross-link.

Understanding the GitHub flow



Understanding the GitHub flow

- ***Branching*** is a key concept you will need to understand.
- Everything in GitHub lives on a branch.
- By default, the "blessed" or "canonical" version of your project lives on a branch called **main**.
- When you are ready to experiment with a **new feature** or **fix an issue**, you create a **new branch** of the project.
- The branch will look exactly like main at first, but any changes you make will only be reflected in your branch.
- Such a new branch is often called a **"feature"** branch.

Understanding the GitHub flow

- As you make changes to the files within the project, you will **commit** your changes to the **feature** branch.
- When you are ready to start a discussion about your changes, you will open a **pull request**.
- A **pull request** doesn't need to be a perfect work of art - it is meant to be a starting point that will be further refined and polished through the efforts of the project team.
- When the changes contained in the pull request are approved, the feature branch is **merged** onto the main branch.

Branching with Git

The first step in the **GitHub Workflow** is to ***create a branch***. This will allow us to experiment with new features without accidentally introducing untested changes on our production branch.

- When you create a branch, you are essentially creating an **identical copy of the project at that point in time**. This isn't the same as creating a physical copy on disk. In the background, a branch is just a pointer.

Activity: Creating a branch with GitHub

Earlier you created an **issue** about the file you would like to edit. Let's **create the branch** you will use to edit your file.

- Follow these steps to create a new branch in the class repository:
- You will need to have collaborator access on the class repository before you can create a branch on GitHub.
    - Navigate to Code tab of the class repository.
    - Click the branch dropdown.
    - Enter the branch name github-username-caption.
    - Click on your branch name to create the branch.

Local Git configuration

Now we prepare your local environment to work with Git.

```
$ git version
```

- Git allows you to set configuration options at three different levels.

--system

- These are system-wide configurations. They apply to all users on this computer.

--global
These are the user level configurations. They only apply to your user account.

--local
These are the repository level configurations. They only apply to the specific repository where they are set.

Viewing your configurations

If you would like to see which config settings have been added automatically, you can type *git config --list*. This will automatically read from each of the three config files and list the setting they contain.

- You can also narrow the list to a specific configuration level by including it before the list option.

```
$ git config --global --list
```

Configuring your name and email address

Git uses the config settings for your **user name** and **email address** to generate a unique fingerprint for each of the commits you create. You can't create commits without these settings:

```
$ git config --global user.name "First-name Surname"
$ git config --global user.email "you@email.com"
```

```
$ git config --global user.name "Antonio Bucchiarone"
$ git config --global user.email "bucchiarone@fbk.eu"
```

Working locally with Git

- Creating a local copy of the repository

1. Navigate to the Code tab of the class repository on GitHub.
2. Click the green Code button.
3. Copy the clone URL to your clipboard.
4. Open your command-line application.
5. Retrieve a full copy of the repository from GitHub: ***git clone <CLONE-URL>***
6. Once the clone is complete, navigate to the new directory created by the clone operation: ***cd <REPOSITORY-NAME>***

Our favorite Git command: *git status*

```
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
nothing to commit, working tree clean
```

- ***git status*** is a command to verify the *current state of your repository* and the files it contains. Right now, we can see that we are on branch main, everything is up-to-date with origin/main and our *working tree* is clean.

Using branches locally

```
$ git branch
```

```
$ git branch --all
$ git branch -a
```

- If you want to see all the branches, including the read-only copies of your remote branches, you can add the --all option or just -a.

Switching branches

```
$ git switch <BRANCH-NAME>
```

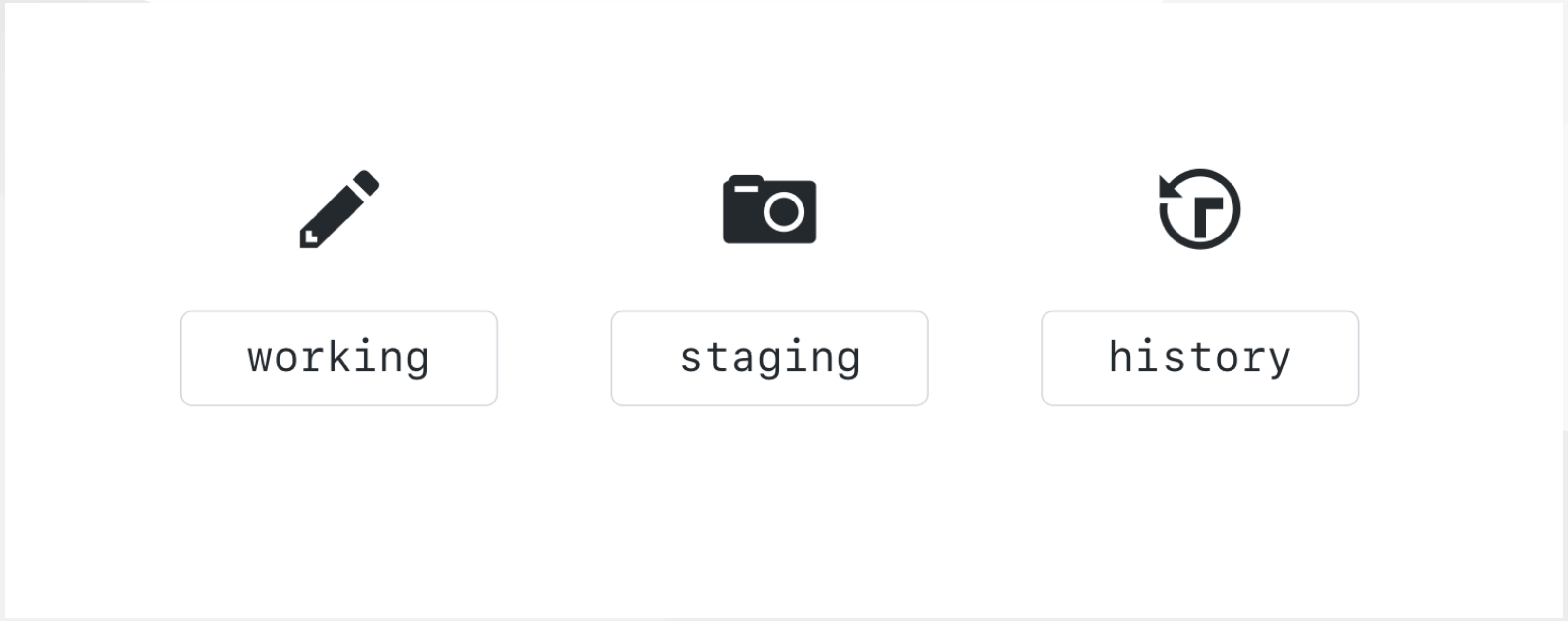
- Git will provide a message that says you have been switched to the branch, and it has been set up to track the same remote branch from origin.

Activity: Edit Your File

- Now it is time to create and modify your file:
    - Create a file in your local folder
    - Open your file in your favorite editor (VSCode).
    - Add a text line
    - Save your file.

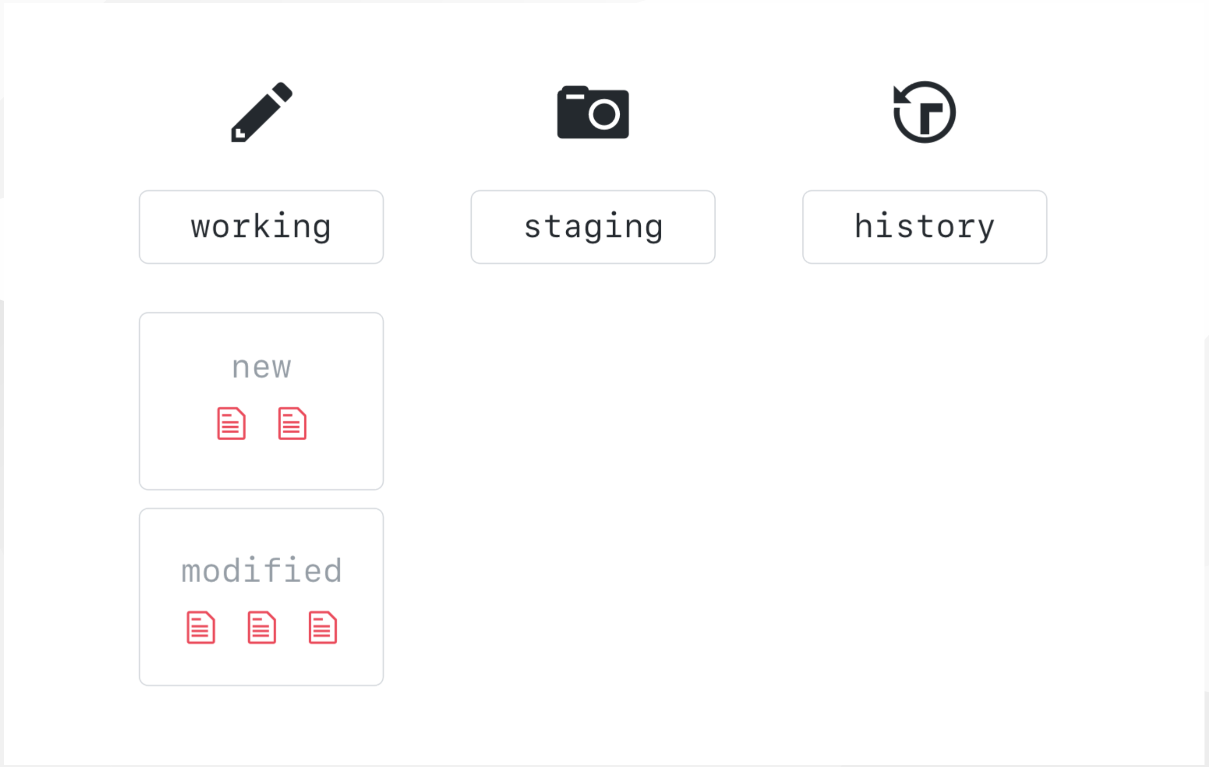
The two stage commit

- After you have created your file, it is time to create your first snapshot of the repository.
- When working from the command line, you will need to be familiar with the idea of the two stage commit.

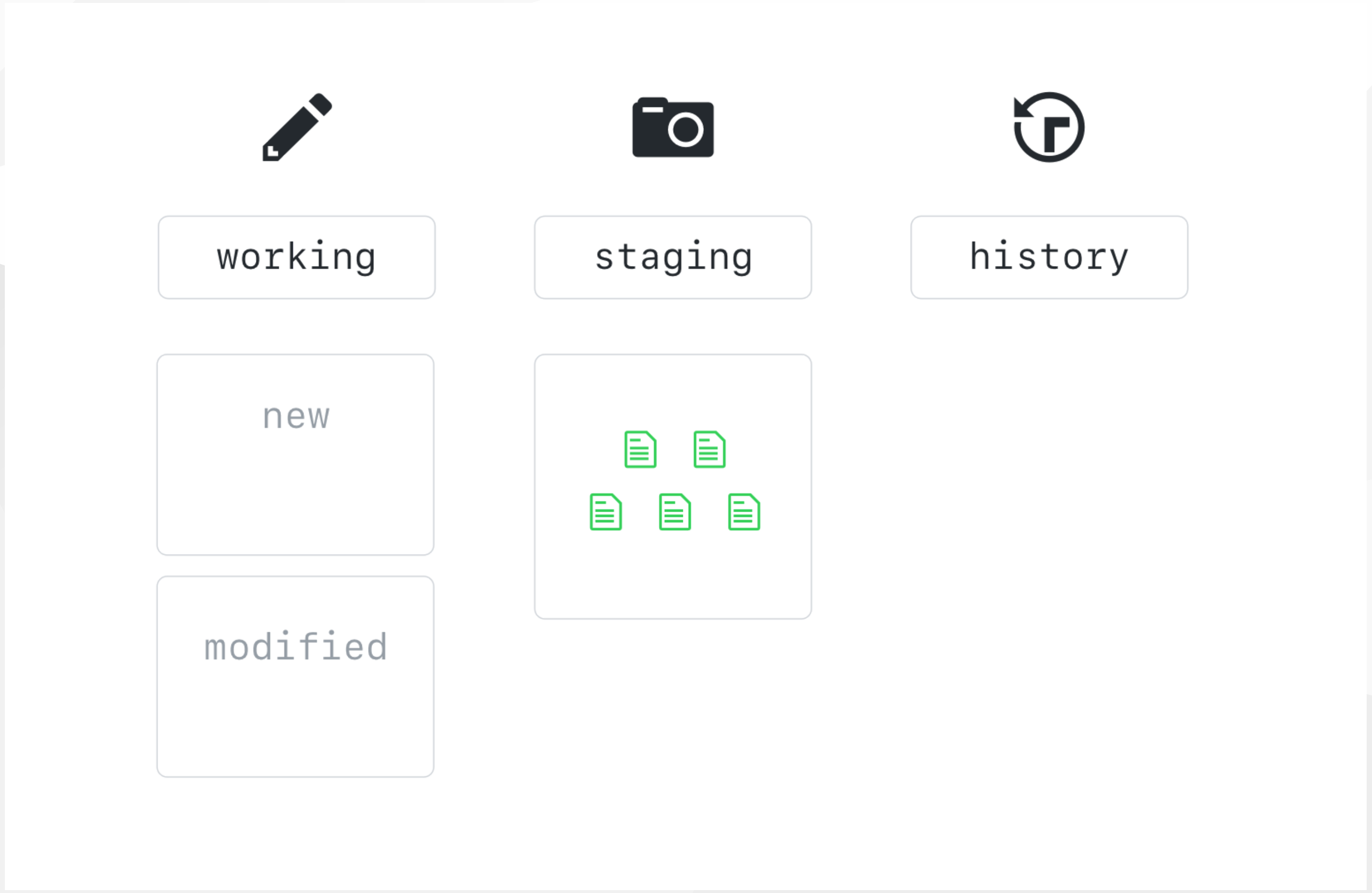


Working Trees

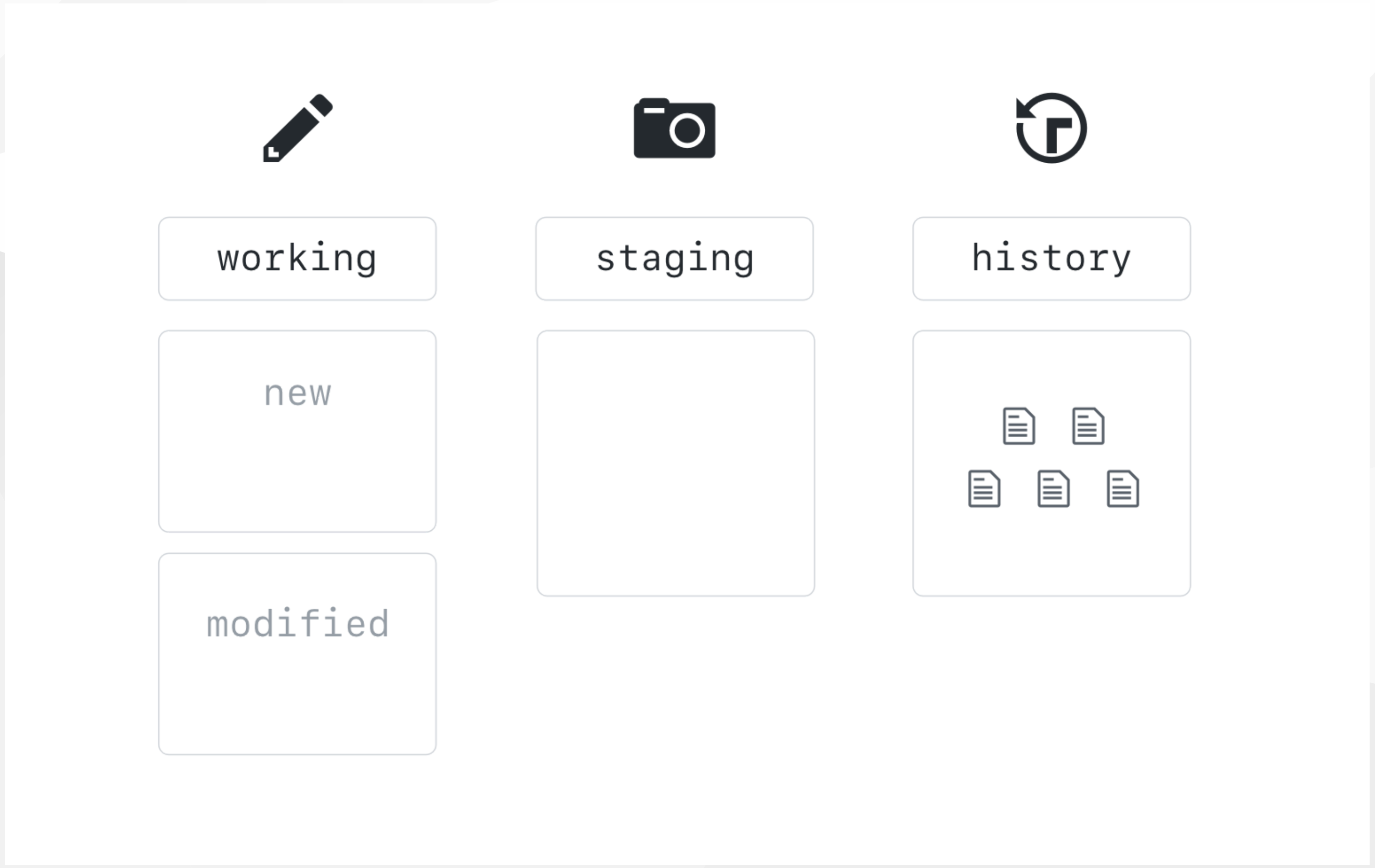
- When you work locally, your files exist in one of four states. They are either **untracked**, **modified**, **staged**, or **committed**.
    - An **untracked** file is a new file that has never been committed.
- Git tracks these files, and keeps track of your history by organizing your files and changes in three working trees.
- **Working**, **Staging** (also called Index), and **History**.
- When we are actively making changes to files, this is happening in the working tree.


working
staging
history
new
modified

- To add these files to version control, you will create a collection of files that represent a discrete unit of work. We build this unit in the staging area.



- When we are satisfied with the unit of work we have assembled, we will take a snapshot of everything in the staging area. This is called a commit.



Activity : Commit

1. let's check the status of our working tree:

```
$ git status
```

2. Move the file from the working tree to the staging area:

```
$ git add my-file.md
```

3. Let's see what happened:

```
$ git status
```

4. Now let's take our first snapshot:

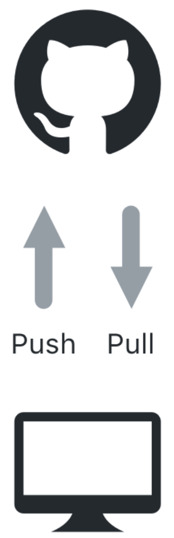
```
$ git commit - m "<commit message>"
```

Commit Message

Good commit messages should:

- Be short. ~50 characters is ideal.
- Describe the change introduced by the commit.
- Tell the story of how your project has evolved.

Collaborating on your code



- In this case, our remote is GitHub.com, but this could also be your company's internal instance of GitHub Enterprise.

Push

- To push your changes to GitHub, you will use the command:

```
$ git push
```

Activity: Creating a pull request

- Pull Requests are used to propose changes to the project files.
- A pull request introduces an action that addresses an Issue.
- A Pull Request is considered a "work in progress" until it is merged into the project.
- Now that you have started to change your file, you will open a pull request to discuss the file with your teammates.

Activity: Creating a pull request

- Follow these steps to create a Pull Request in the class repository:

1. Click the **Pull Request** tab.
2. Click **New Pull Request**.
3. In the **base** dropdown, choose **main**
4. In the **compare** dropdown, choose **your branch**.
5. Type a subject line and enter a comment.
6. Use Markdown formatting to add a header and a checklist to your Pull Request.

Activity: Creating a pull request

7. Include one of the keywords: **closes**, **fixes**, or **resolves** followed by the **issue number** you created earlier to note which Issue the Pull Request should close.

Example: **This resolves #3**

8. Click **Preview** to see how your Pull Request will look.
9. Assign the Pull Request to yourself.
10. Select your partner as a Reviewer for the Pull Request.
11. Click **Create pull request**.

Exploring a pull request

- Conversation tab
  - A Pull Request contains a discussion about the changes being made to the repository.
  - This discussion is found in the Conversation tab.
- Commits tab
  - The Commits tab contains more detailed information about who has made changes to the files.
- Files changed tab
  - The Files changed tab allows you to see cumulative effect of all the changes made on the branch.

Activity: Code review

- One of the best ways to ensure code quality is to make peer reviews a part of every Pull Request.
- Let's review your partner's code now:

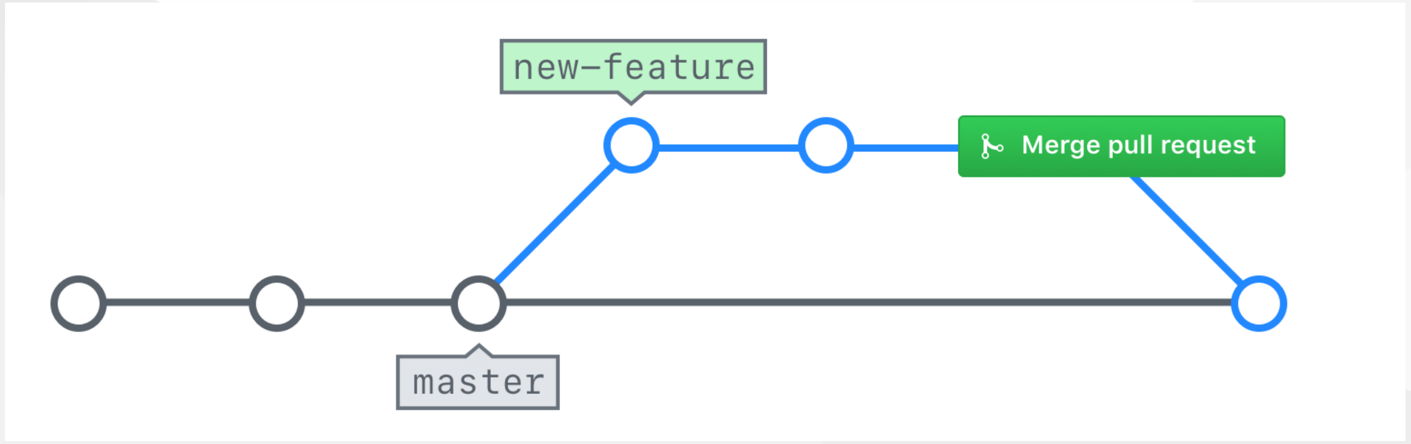
1. Click the **Pull Request** tab.
2. Use the **Author** drop down to locate your partner's pull request.
3. Click the **Files Changed** tab.
4. Hover over a single line in the file to see the blue +. Click the + to add a line comment.
5. Comment on the line and click **Start a review**.

Activity: Code review

6. Add additional line comments to the pull request.
7. Click **Review changes** in the top right corner.
8. Choose whether to **Approve** or **Request changes**.
9. Enter a general comment for the review.
10. Click **Submit review**.
11. Click the **Conversation** tab to check out your completed review.

Merging pull requests

- When you merge your branch, you are taking the content and history from your feature branch and adding it to the content and history of the main branch.



Merging pull requests

- Many project teams have established rules about who should merge a pull request.
  - Some say it should be the person who created the pull request since they will be the ones to deal with any issues resulting from the merge.
  - Others say it should be a single person within the project team to ensure consistency.
  - Still others say it can be anyone other than the person who created the pull request to ensure at least one review has taken place.

Merging your pull request

1. Navigate to your Pull Request
2. Click **Conversation**
3. Scroll to the bottom of the Pull Request and click the **Merge pull request** button
4. Click **Confirm merge**
5. Click **Delete branch**
6. Click **Issues** and confirm your original issue has been closed

Updating your local repository

- When you merged your Pull Request, you deleted the branch on GitHub, but this will not automatically update your local copy of the repository. Let's go back to our command line application and get everything in sync.

First, we need to get the changes we made on GitHub into our local copy of the repository:

1. Start by switching back to your default branch:

```
$  git switch main.
```

2. Retrieve all the changes from GitHub:

```
$  git pull.
```

Updating your local repository

- **git pull** is a combination command that retrieves all the changes from GitHub and then updates the branch you are currently on to include the changes from the remote.
- The two separate commands being run are **git fetch** and **git merge**

Cleaning up the unneeded branches

- If you type **git branch --all** you will probably see that, even though you deleted your branch on the remote, it is still listed in your local copy of the repository.

1. Take a look at your local branches: **git branch --all**
2. Let's see which branches are safe to delete: **git branch --merged**
3. Delete the local branch: **git branch -d <branch-name>**
4. Take another look at the list: **git branch --all**
5. Your local branch is gone, but the remote tracking branch is still there. Delete the remote tracking branch: **git pull --prune**

NOTE: If you would like pruning of the remote tracking branches to be set as your default behavior when you pull, you can use the following configuration option:

git config --global fetch.prune true.

Viewing local project history

- Using Git log
  - When you clone a repository, you receive the **history** of all the commits made in that repository. The **log command** allows us to view that history on our local machine.

```
$ git log
$ git log --oneline
$ git log --oneline --graph
$ git log --oneline --graph --decorate
$ git log --oneline --graph --decorate --all
$ git log --stat
$ git log --patch
```

- Use the **up** and **down** arrows or press enter to view additional log entries.
- Type **q** to quit viewing the log and return to the command prompt

Activity: Do yourself

1. Create a new branch, and switch to that branch.
2. Create a new file in the **deliverables** folder
3. Edit your file with this content

```
# Slide 1

![w:700](IMAGE URL)
```

4. Save and commit your changes on your new branch.

5. If working locally, push your changes up to the remote:

```
$ git push -u origin NEWBRANCHNAME
```

6. Open a *pull request*. In the body of the pull request, @ mention anyone you'd like to review your changes.
7. Once the tests pass on your pull request, merge the pull request.

Add a caption to an existing slide

1. Find a pull request that you'd like to add a caption to.
2. See what image they have chosen by clicking *'files changed'*, and then *'view'*.
3. Edit the file locally, to add a line with your text. (Please do not erase work that others have added to this file.)
   - If you are working locally, you will need to check out to the branch.
   - Make sure you have all the remote changes updated in your local repository: **git pull**.
   - Look for the name of the branch in the pull request, and check out to that branch locally: **git switch BRANCHNAME**.

4. Save the changes, and commit the file.
    - If you are working locally, push the changes up to the remote: **git push**.
5. Do not merge the pull request, simply @ mention the user who opened the pull request to let them know about your changes.

Improve the README.md

- Improve our *README.md* by adding some of your favorite resources.

1. Create branch, and switch to that branch: *git switch -c NEWBRANCHNAME*.
2. Edit the *README.md* to be better in some way. This could mean adding a new resource, or making the existing descriptions more clear. Save and commit your changes.
3. If working locally, push your changes up to the remote: *git push -u origin NEWBRANCHNAME*.

4. Open a pull request In the body of the pull request, with base: **main** and **compare: NEWBRANCHNAME**.
5. @ mention anyone you'd like to review your changes.
6. Once the tests pass on your pull request, merge the pull request.

Restyle slides

If you'd like a more advanced challenge, and you have an eye for style, change the colors, font, and other aspects of the class slide deck. ***Note: If multiple participants attempt this, there may be merge conflicts*.**

1. Create a new branch and check out to it: git switch -c NEWBRANCHNAME.
2. Make changes in the file (Add one more slide).
3. Save and commit your changes on your branch.
4. Push your branch to the remote: **git push -u origin NEWBRANCHNAME**.
5. Open a pull request with **base: main** and **compare: NEWBRANCHNAME**.
6. @ mention anyone you'd like to review your changes.
7. Once the tests pass on your pull request, merge the pull request.

Exercise 1

Exercise Title: Collaborative Git Workflow

Exercise Description:

In this exercise, you'll simulate a collaborative Git workflow where two team members make separate commits to a shared Git repository. This exercise is designed to help you understand how multiple individuals can work together using Git.

Setup:

1. Create a Git repository on GitHub.
2. Clone the repository to your local machine using `git clone`.

Team Member A's Tasks:

1. Team Member A should create a new branch using `git checkout -b feature-A` and switch to it.
2. Create a new text file named `featureA.txt` in the repository directory.
3. Add some content to `featureA.txt`.
4. Commit the changes with a meaningful commit message using `git commit`.
5. Push the branch to the remote repository using `git push origin feature-A`.

Team Member B's Tasks:

1. Team Member B should create a new branch using `git checkout -b feature-B` and switch to it.
2. Create a new text file named `featureB.txt` in the repository directory.
3. Add some content to `featureB.txt`.
4. Commit the changes with a meaningful commit message using `git commit`.
5. Push the branch to the remote repository using `git push origin feature-B`.

Team Member A's Tasks (Continued):

1. After Team Member B has pushed their changes, Team Member A should switch back to the `main` branch using `git checkout main`.
2. Pull the latest changes from the remote repository using `git pull origin main`.
3. Merge the changes from the `feature-B` branch into the `main` branch using `git merge feature-B`.
4. Resolve any conflicts if they occur during the merge.
5. Commit the merge changes with a meaningful commit message.
6. Push the `main` branch to the remote repository using `git push origin main`.

Team Member B's Tasks (Continued):

1. After Team Member A has merged their changes, Team Member B should switch back to the `main` branch using `git checkout main`.
2. Pull the latest changes from the remote repository using `git pull origin main`.
3. Merge the changes from the `feature-A` branch into the `main` branch using `git merge feature-A`.
4. Resolve any conflicts if they occur during the merge.
5. Commit the merge changes with a meaningful commit message.
6. Push the `main` branch to the remote repository using `git push origin main`.

Exercise 2

In this exercise, you will practice a collaborative Git workflow involving two team members who will use Git issues, create pull requests, and perform merges. This exercise will help you understand how these Git features facilitate teamwork and code collaboration.

Setup:

1. Create a Git repository on a platform like GitHub.
2. Clone the repository to your local machines using `git clone`.

Team Member A's Tasks:

1. **Issue Creation:**
    - Create a new issue on the repository's issue tracker. Name it "Feature A Implementation" and describe it briefly.
2. **Branch Creation:**
    - Create a new branch named `feature-A` and switch to it.

Team Member A's Tasks (Continued):

3. **Work on Feature A:**
    - Create a new text file named `featureA.txt` and Add some content to `featureA.txt`.
    - Commit the changes with a meaningful commit message.
4. **Push the Branch:**
    - Push the `feature-A` branch to the remote repository using `git push origin feature-A`.

Team Member B's Tasks:

1. **Issue Creation:**
   - Team Member B should create a new issue on the repository's issue tracker. Name it "Feature B Implementation" and describe it briefly.
2. **Branch Creation:**
   - Create a new branch named `feature-B` and switch to it.

Team Member B's Tasks: (Continued):

3. **Work on Feature B:**
    - Create a new text file named `featureB.txt` in the repository directory.
    - Add some content to `featureB.txt`.
    - Commit the changes with a meaningful commit message.

4. **Push the Branch:**
    - Push the `feature-B` branch to the remote repository using `git push origin feature-B`.

Team Member A's Tasks (Continued):

1. **Create a Pull Request (PR):**
   - Team Member A should create a pull request (PR) from the `feature-A` branch to the `main` branch. In the PR description, reference the "Feature A Implementation" issue.
2. **Review and Merge:**
   - Team Member B should review the PR, add comments if necessary, and approve it if satisfied.
   - Team Member A can merge the PR into the `main` branch once it's approved.

Team Member B's Tasks (Continued):

1. **Create a Pull Request (PR):**
   - Team Member B should create a pull request (PR) from the `feature-B` branch to the `main` branch. In the PR description, reference the "Feature B Implementation" issue.
2. **Review and Merge:**
   - Team Member A should review the PR, add comments if necessary, and approve it if satisfied.
   - Team Member B can merge the PR into the `main` branch once it's approved.

Quiz Time :)

Questions?

bucchiarone@fbk.eu