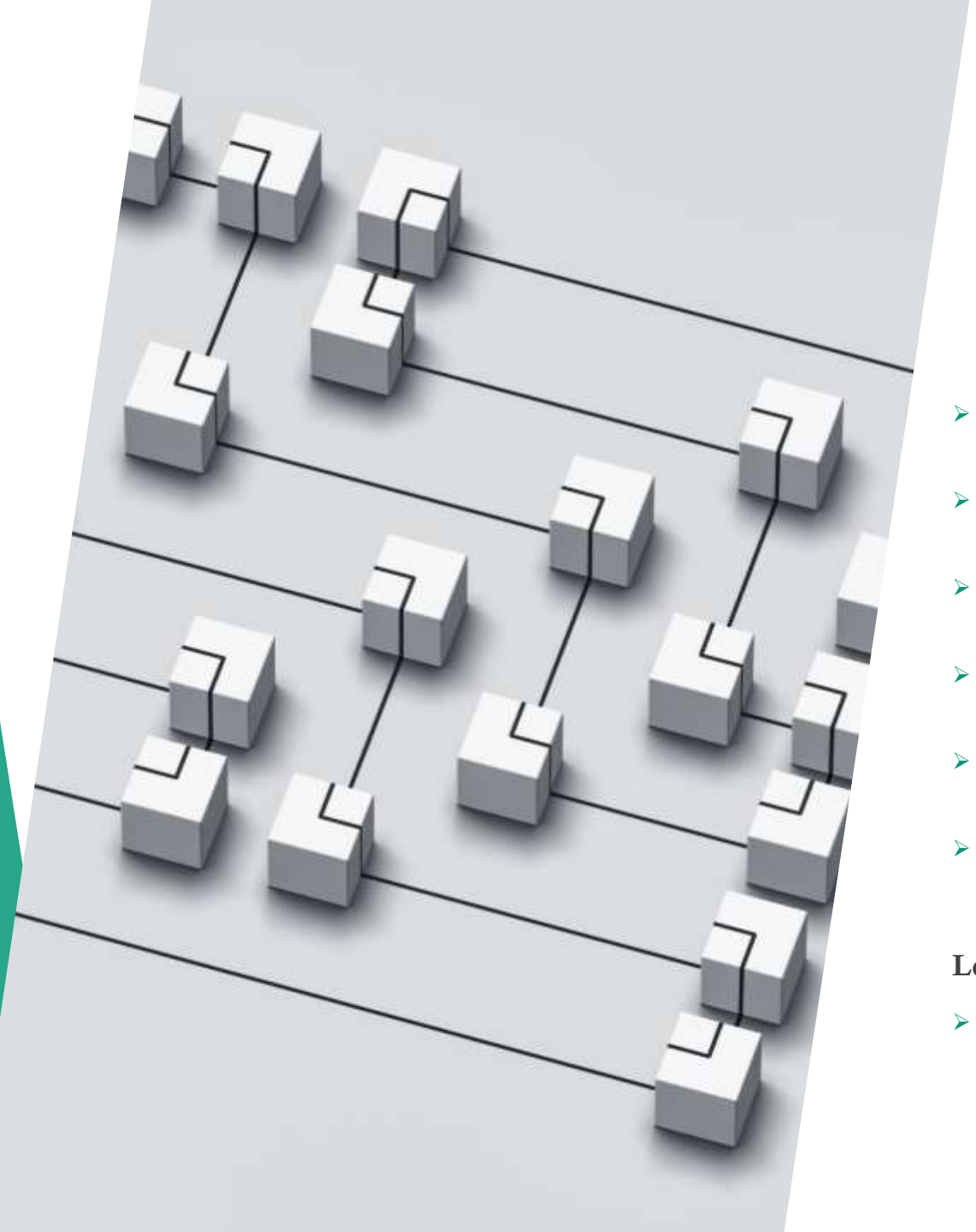
Project Management with Git

TCF LABS

Course Objectives:

- To become familiar with basic Git commands.
- To create and manage branches.
- To understand how to collaborate and work with remote repositories.
- To become familiar with version controlling commands.



Course Overview:

- **Basic Git Commands:** Learn essential commands to get started with Git.
- **Branching and Merging:** Understand how to create and manage branches, and merge them.
- **Stashing:** Temporarily store changes without committing them.
- **Remote Repositories:** Work with Git repositories hosted on platforms like GitHub.
- **Collaboration Workflows:** Best practices for collaborating with others using Git.
- **Advanced Git Operations:** Explore more complex operations like rebasing, cherry-picking, and resolving merge conflicts.

Learning Outcomes:

- By the end of this course, you will be proficient in using Git for version control and collaboration.

Experiment 1

Setting Up and Basic Commands

Introduction

What is Git?

- Git is a distributed version control system used to track changes in source code during software development.
- Allows multiple developers to collaborate on a project without overwriting each other's work.

Why Use Git in Software Development?

- Enables easy tracking of changes and history.
- Facilitates collaboration and code sharing among team members.
- Helps in managing and merging different versions of code efficiently.
- Protects your project from loss or corruption.

Git Installation Check and Configuration

Download and Install Git :

- Download and install Git from git-scm.com.
- During installation, choose Git Bash as the default terminal.

Configure User Details Globally:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Verify Configuration:

```
git config –list
```

Exercise: Initialize a New Git Repository

- Initialize a Git Repository

```
git init
```

- Create a README File

```
echo "# My Project" > README.md
```

- Stage the File

```
git add README.md
```

- Commit the File

```
git commit -m "Initial commit"
```

Basic Git Commands

Git Workflow:

The three key areas in Git are: Working Directory, Staging Area, and Repository.

Working Directory:

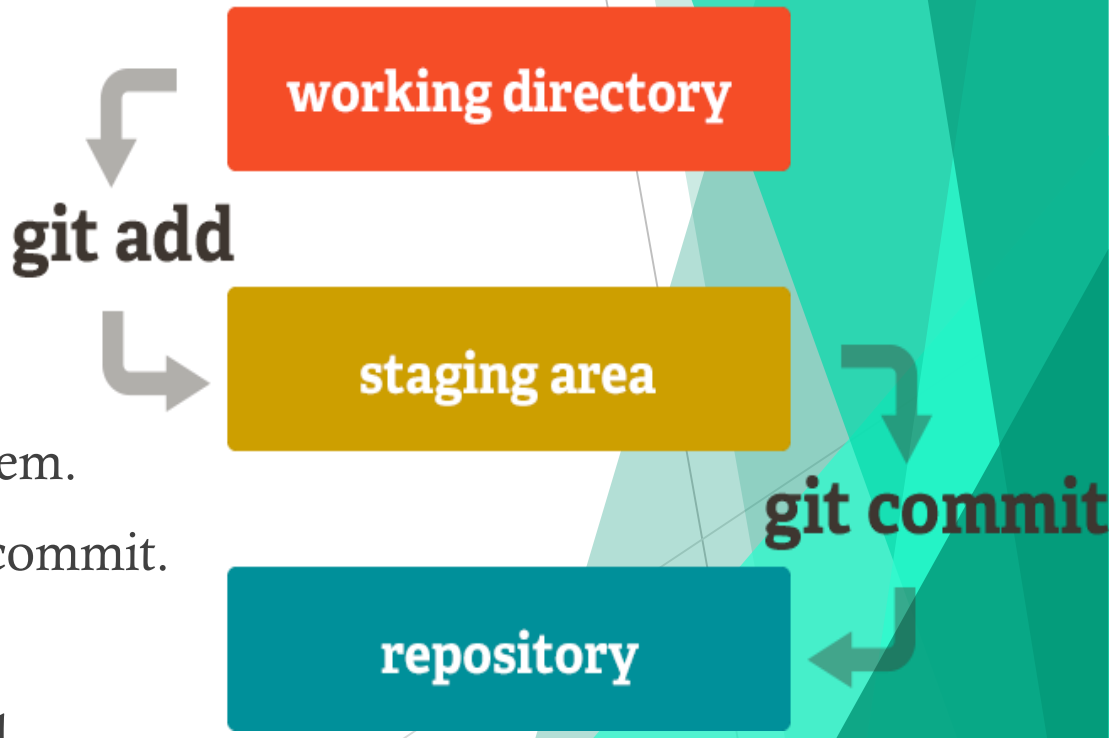
- This is where you make changes to your files.
- This is the current state of your project.

Staging Area:

- A place to prepare your changes before committing them.
- Files added here are marked for inclusion in the next commit.

Repository:

- The actual database where all your commits are stored.
- A history of your project.



Commands Demonstration

Commands:

- git status: Check the status of files.
- git add: Add files to the staging area.
- git commit: Commit changes to the repository.
- git log: View the commit history.

Exercise - Multiple File Operations

- Create Three files:

```
echo "Hello, World!" > file1.txt
echo "Another file" > file2.txt
echo "Yet another file" > file3.txt
```

- Stage and commit two files together:

```
git add file1.txt file2.txt
git commit -m "Add two new files"
```

- Stage and Commit the Third File Separately:

```
git add file3.txt
git commit -m "Add third file"
```

Experiment 2

Creating and Managing Branches

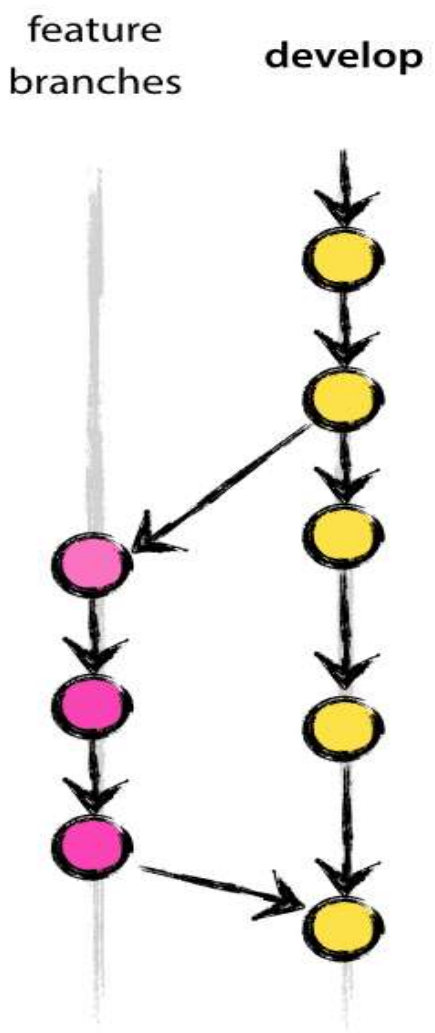
Concept of Branches

Branching:

- Allows independent work on features without affecting the main codebase.
- The main branch (e.g., master, develop) remains stable while feature branches are developed.
- The main branch and multiple feature branches are shown in the figure.

Why use branches?

- Feature Development: Work on new features without disrupting the main project.
- Bug Fixes: Quickly address bugs on a separate branch.
- Experimentation: Test new ideas safely.



Commands Demonstration

Commands:

- git branch: List and create branches.
- git checkout: Switch between branches.
- git merge: Merge branches together.

Example Workflow - Branch Management

```
git branch  # List branches
git branch new-branch  # Create a new branch
git checkout new-branch  # Switch to the new branch
git merge new-branch  # Merge new-branch into the current branch
```

Exercise - Feature Branch Workflow

- Create a New Branch:

```
git branch feature-branch
```

- Switch to the New Branch:

```
git checkout feature-branch
```

- Make Changes and Commit:

```
echo "Feature work" > feature.txt
git add feature.txt
git commit -m "Add feature file"
```

- Switch Back to Master:

```
git checkout master
```

- Merge the Feature Branch:

```
git merge feature-branch
```

Experiment 3

Creating and Managing Branches

Concept of Stashing

What is Stashing?

- Temporarily saves changes in your working directory.
- Allows you to switch contexts or work on something else without committing.

When to Use:

- When you need to switch branches or work on a different task without losing current progress."

Commands Demonstration

Commands:

- git stash: Save uncommitted changes.
- git stash pop: Apply the stashed changes and remove them from the stash.

Example Workflow - Stashing

```
echo "Temporary work" > temp.txt
git stash  # Save changes to the stash
git stash pop  # Apply the stashed changes
```

Exercise - Stash Workflow

- Make Changes in Your Working Directory:

```
echo "Work in progress" > temp.txt
```

- Stash the Changes:

```
git stash
```

- Switch Branches and Apply the Stashed Changes:

```
git checkout master
git stash pop
```

Experiment 4

Collaboration and Remote Repositories

Introduction to Remote Repositories

Remote repositories: are centralized locations that allow multiple developers to collaborate on the same project.

- Examples: Popular platforms include GitHub, GitLab, and Bitbucket.
- Significance: Enable team collaboration and version control across different locations.

Why Use Remote Repositories?

- Centralized version control for distributed teams.
- Facilitates easy sharing and collaboration on code.
- Ensures that everyone is working on the most up-to-date version of the project.

Commands Demonstration

Commands:

- git clone: Clone a repository from a remote.
- git remote: Manage remotes (list, add, remove).
- git fetch: Fetch changes from the remote without merging.
- git pull: Fetch and merge changes from the remote.

Example Workflow

```
git clone [URL]                # Clone the remote repository
cd [repository-folder]         # Change to the repository directory
git remote -v                  # List all remotes
git fetch origin               # Fetch updates from the 'origin' remote
git pull origin master         # Fetch and merge updates from 'origin' into 'master'
```

Exercise - Working with Remotes

- Clone a Remote Repository:

```
git clone [URL]  # Replace [URL] with the repository's URL
```

- Fetch the Latest Changes:

```
git fetch origin  # Fetch updates from 'origin'
```

- Rebase Your Local Branch:

```
git pull --rebase origin master  # Fetch and rebase your local branch
```

Experiment 5

Collaboration and Remote Repositories

Pull Requests and Code Review

Pull Requests: A method to submit changes to a project, allowing team members to review code before merging.

Code Review: The process of reviewing code changes for quality and correctness.

Platforms: Commonly used in GitHub, GitLab, and Bitbucket.

Commands Demonstration

Commands:

- git push: Push commits to a remote repository.

Example Workflow

```
git push origin master  # Push commits to the master branch
```

Exercise: Merging with Custom Message

- Create a Feature Branch:

```
git branch feature-branch
git checkout feature-branch
```

- Make Changes, Commit, and Push:

```
echo "Feature changes" > feature.txt
git add feature.txt
git commit -m "Add feature changes"
git push -u origin feature-branch
```

- Merge with Custom Message:

```
git checkout master
git merge feature-branch -m "Custom merge message"
```

Experiment 6

Collaboration and Remote Repositories

Merging Branches with Custom Commit Message

Merging Branches:

- Merging combines changes from one branch into another.
- Custom commit messages can help provide context or explanation for the merge.
- This command should be used to provide a clear documentation of the reasons for making the merge.

Commands Demonstration

Commands:

- git merge feature-branch -m "Your custom merge message here“: Merges feature-branch into the current branch (master) with a specified commit message.

Example Workflow

```
# 1. Switch to the master branch
git checkout master

# 2. Merge feature-branch into master with a custom commit message
git merge feature-branch -m "Merge feature-branch into master: Added new feature X"

# 3. Verify the merge with a custom commit message
git log --oneline
```

Exercise - Merging Branches with Custom Message

- Switch to master Branch:

```
git checkout master
```

- Merge feature-branch with Custom Message:

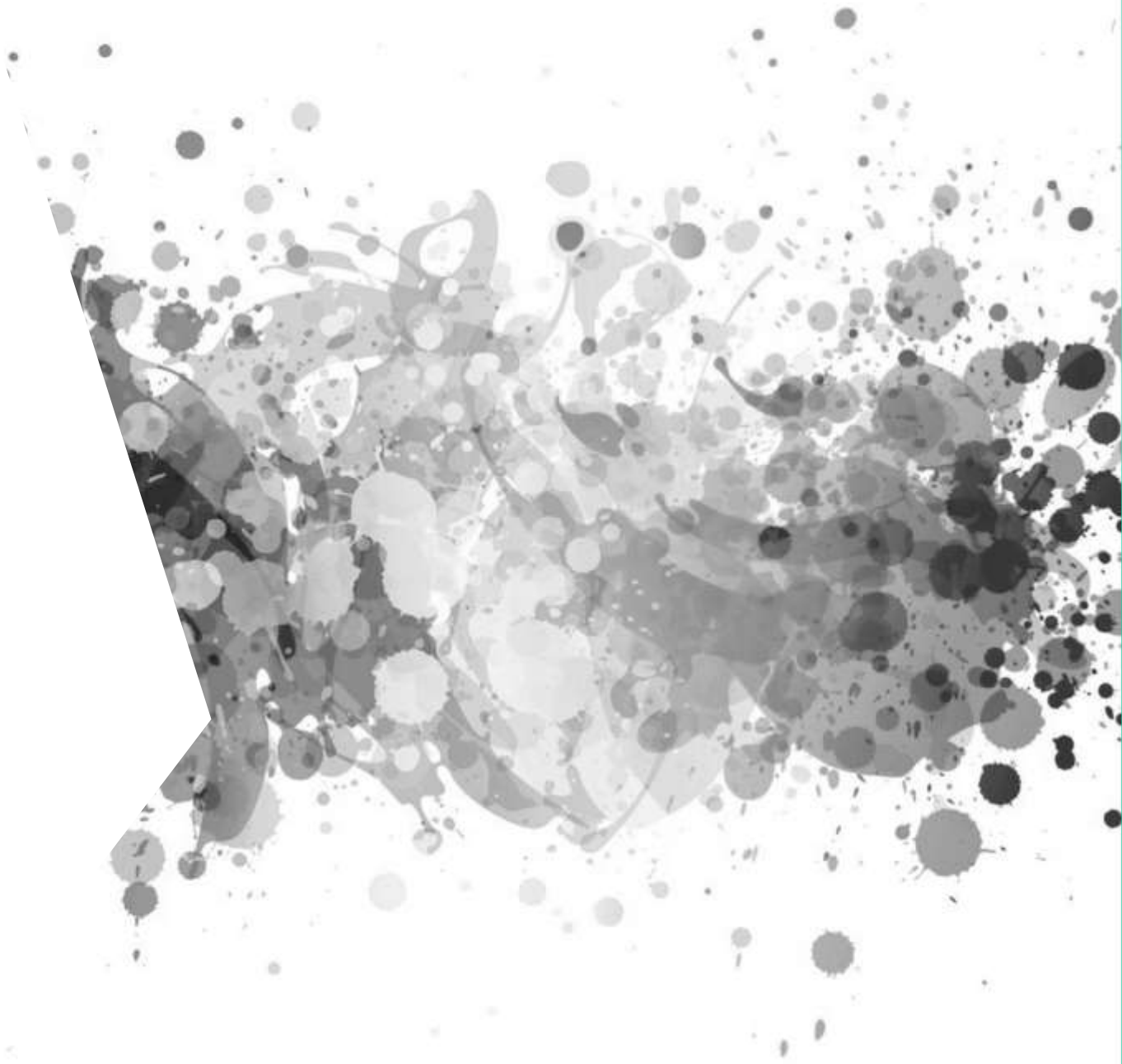
```
git merge feature-branch -m "Add feature branch changes"
```

- Verify the Merge:

```
git log
```

Experiment 7

Git Tags and Releases



Git Tags

- **Tags:** Mark specific points in the Git history, often used for releases.
- **Lightweight tags:** are simple pointers to commits
- **Annotated tags:** Include metadata like the tagger's name, date, and message.

When to Use Tags:

- When releasing a version or marking significant milestones.

Commands Demonstration

Commands:

- git tag: Create and manage tags.
- git push origin [tag]: Push a tag to a remote repository.

Example Workflow

```
git tag v1.0  # Create a lightweight tag
git push origin v1.0  # Push the tag to the remote
```

Exercise: Create a Tag

- Create a Lightweight Tag:

```
git tag v1.0
```

- Push the Tag to the Remote Repository:

```
git push origin v1.0
```

Experiment 8

Advanced Git Operations

Cherry-picking in Git

Cherry-picking:

A Git operation that allows you to apply specific commits from one branch to another.

When and Why to Use:

Use cherry-picking when you need a particular feature or bug fix from another branch without merging the entire branch.

Commands Demonstration

Commands:

- git cherry-pick: Apply specific commits to the current branch.

Example Workflow

```
git cherry-pick <commit-hash>  # Apply a specific commit
```

Exercise - Cherry-pick Commits

- Identify a Commit to Cherry-pick:

```
git log  # Find the commit hash
```

- Cherry-pick the Commit:

```
git cherry-pick <commit-hash>
```

Experiment 9

Analysing and Changing Git History

Viewing Commit Details with Git Show

Commit Details:

- Each commit in Git stores information like the author, date, commit message, and changes made.
- Reviewing these details helps understand the project's history and tracking changes.
- Helpful for code reviews or when understanding the impact of a particular change..

Commands Demonstration

Commands:

- git show: Display detailed information about a specific commit.

Example Workflow

```
git show <commit-hash>  # View commit details
```

Exercise - Viewing Commit Details

- Identify a Commit to Inspect:

```
git log  # Display the commit history
```

- View the Commit Details:

```
git show <commit-hash>  # Display detailed commit information
```

- Analyze the Output: Review the commit message, author information, and the changes made in the commit

Experiment 10

Analysing and
Changing Git History

Listing Commits by Author and Date

Filter by Author and Date:

- Use these filters to narrow down commit history based on specific criteria.
- This is useful for finding changes made by a particular developer or during a certain time period.
- This allows for reviewing contributions by a specific team member or analyzing activity within a particular time frame.

Commands Demonstration

Commands:

- git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31": List commits by a specific author within a date range.

Example Workflow

```
git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"
```

- --author filter: Specifies the author of the commits.
- --since and --until filters: Define the date range for the commits.

Exercise - Listing Commits by Author and Date

- Filter by Author:

```
git log --author="JohnDoe"
```

- Filter by Date Range:

```
git log --since="2023-01-01" --until="2023-12-31"
```

- Combine Filters:

```
git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"
```

Experiment 11

Analysing and Changing Git History

Viewing Recent Commits

Recent Commits:

- Viewing recent commits helps you quickly check the latest changes made to the repository.
- Useful for reviewing recent work and ensuring that recent commits are as expected.
- Quickly review the latest changes made to the repository.

Commands Demonstration

Commands:

- git log -n 5: Show the last 5 commits.

Example Workflow

```
git log -n 5
```

- -n 5: Limits the output to the most recent 5 commits.

Exercise - Viewing Recent Commits

- Display Recent Commits:

```
git log -n 5
```

- Analyze the Output: Review the commit messages, authors, and dates to understand recent changes.

Experiment 12

Analysing and Changing Git History

Reverting Changes

Reverting Commits:

- Reverting a commit creates a new commit that undoes the changes made in a specified commit.
- This is useful when you want to undo changes without altering the commit history.
- Use this command to undo changes from a past commit while preserving the history.

Commands Demonstration

Commands:

- git revert <commit-hash>: Revert a specific commit.

Example Workflow

```
git revert <commit-hash>
```

- <commit-hash>: The hash of the commit you want to revert.

Exercise - Reverting Changes

- Identify a Commit to Revert:

```
git log
```

- Revert the Commit:

```
git revert <commit-hash>
```

- Review the Changes: Check the repository to ensure that the changes have been successfully reverted.

References

- https://git-scm.com/doc
- https://www.atlassian.com/git
- https://www.w3schools.com/git/
- https://docs.github.com/en