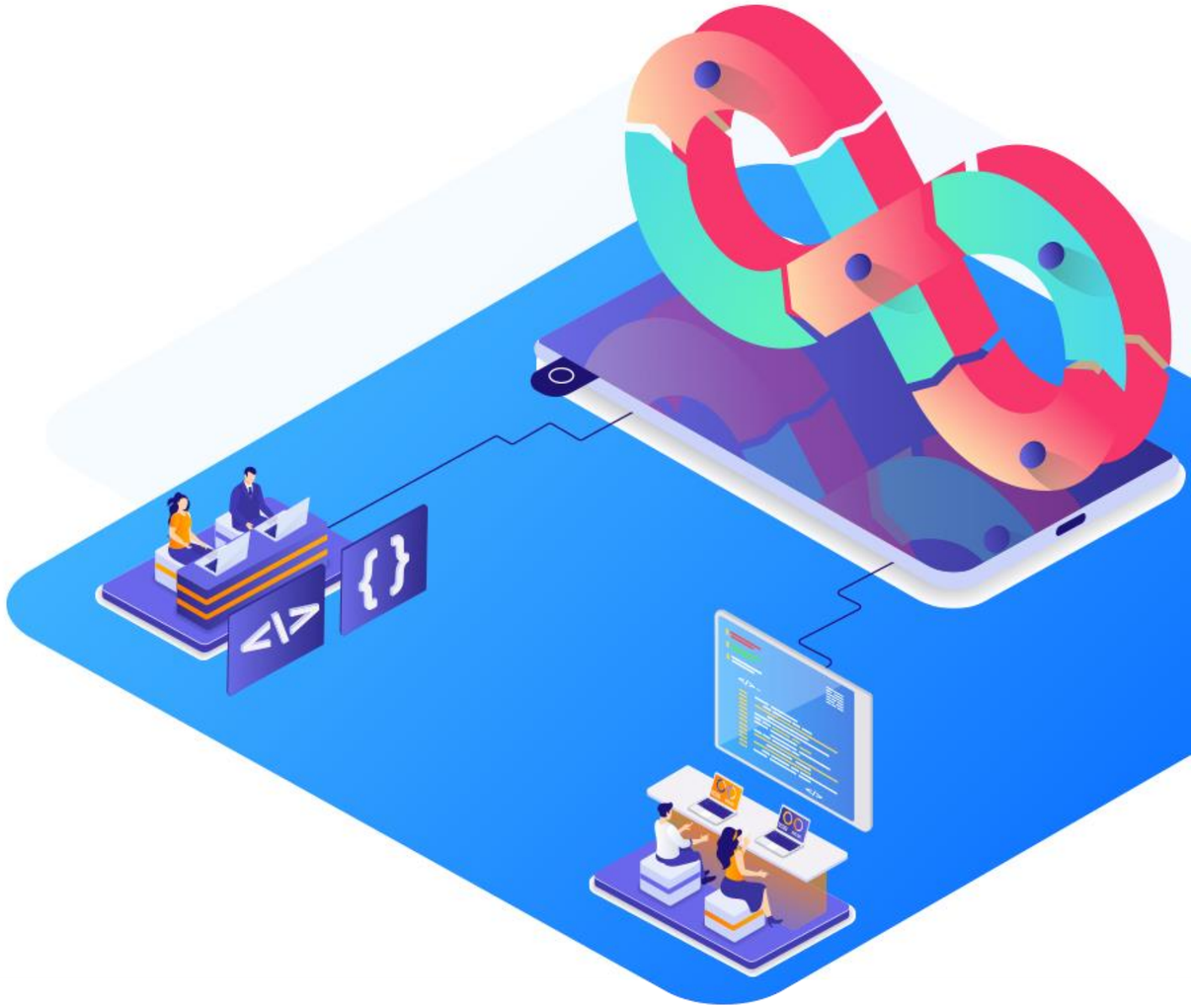
DevOps Foundations: Version Control and CI/CD with Jenkins



Exploring Advanced Concepts in Git

Learning Objectives

By the end of this lesson, you will be able to:

- Create and track issues in Git to prioritize the issues and address them efficiently
- Set up an upstream connection in Git to track changes made to the remote repository, enabling synchronization between local and remote branches
- Create and delete tags in Git to facilitate deployment and rollback processes
- Perform stashing and rebasing in Git to enable efficient collaboration and debugging processes

Advanced Operations in Git

Issue Tracking in GitHub

Issue tracking in GitHub is a core feature designed to plan, discuss, and track the progress of tasks and bugs related to a project.

GitHub Issues can be utilized to manage ideas, enhancements, tasks, or bugs for work on GitHub.

Creating Issues

Users create issues for bug reports, feature suggestions, or task discussions. There are multiple ways to create issues in GitHub, which include:

1. Creating an issue from repository
2. Creating an issue with GitHub CLI
3. Creating an issue from a comment
4. Creating an issue from code

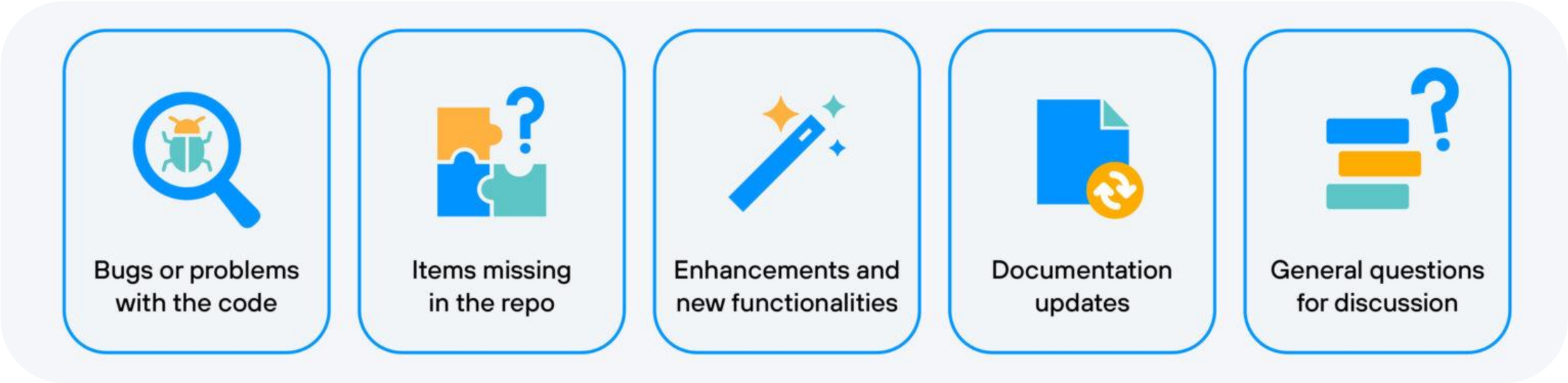
Creating Issues

There are multiple ways to create issues in GitHub, including:

5. Creating an issue from discussion
6. Creating an issue from a project
7. Creating an issue from task list item
8. Creating an issue from a URL query

Tracking Issues

GitHub issues can be used to track the following:



Source: https://7pace.com/blog/github-issues

Tracking Issues

Users can follow these steps to effectively track issues in GitHub:

1. Create issues
2. Assign and label
3. Set milestones
4. Use projects
5. Communicate

Tracking Issues

Create issues

Create issues to report bugs, suggest features, or outline tasks. Provide clear titles and descriptions for each issue.

Assign and label

Assign issues to team members responsible for resolving them. Use labels to categorize the issues by type, priority, or status.

Set milestones

Establish milestones to group related issues together and track progress toward specific goals or deadlines.

Tracking Issues

Use projects

Utilize GitHub projects to organize and visualize issues in a list format. Projects provide a flexible way to manage workflows and track work progress.

Communicate

Communicate updates regularly and track the progress on issues within the GitHub platform.

Assisted Practice

Creating and tracking issues in GitHub **Duration: 10 Min.**

Problem statement:

You have been assigned a task to demonstrate the process of using GitHub issues for planning and tracking work efficiently.

Outcome:

By completing this demo, you will be able to effectively learn how to use GitHub issues for efficient planning and tracking of work.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create an issue
2. Track the issue

Upstream and Downstream

Git upstream and Git downstream refer to the direction of data flow between repositories.

Upstream

- It refers to the original repository from which a clone was made.
- It typically represents the main project repository or the repository that you forked from.
- Changes flow from the upstream repository to your local repository when you pull any update.

Downstream

- Downstream repositories are copies of the upstream repository where changes are made.
- Your local repository and any forks you create are examples of downstream repositories.
- Changes flow from your local repository or forks to the upstream repository through processes like push or pull requests.

Assisted Practice

Working with Git upstream **Duration: 10 Min.**

Problem statement:

You have been assigned a task to set up and manage an upstream connection in Git, enabling contributors to stay synchronized with changes made in the original repository while working on their forked copies.

Outcome:

By completing this demo, you will be able to learn setting up and managing an upstream connection in Git to keep forked copies synchronized with the original repository.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines

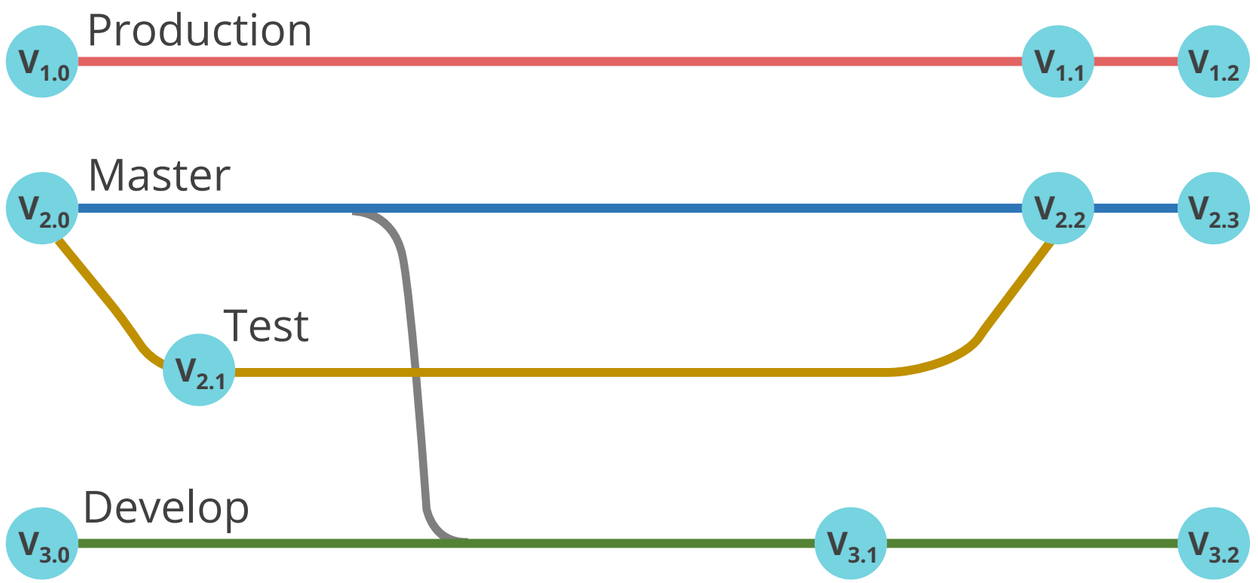
Steps to be followed:

1. Clone your fork
2. Add upstream remote
3. Verify remote

Git Tags

Git tags serve as markers within the Git history, indicating notable occurrences like releases or important commits.

The below figure demonstrates the tags on various branches:



Git Tags: Types

Git supports two types of tags, which are:

Lightweight tags

Annotated tags

Git Tags: Types

Lightweight tags

- These tags are simple pointers to specific commits in the Git history.
- Lightweight tags do not contain additional metadata such as tagger name, email, or tagging message.

Annotated tags

- Annotated tags are more detailed than lightweight tags.
- They include extra information such as the tagger's name, email, creation date, and an optional tagging message.

Git Tags: Operations

Git tags support various operations, including:

Git Create tag

To create a tag, you should start by switching to the branch where you intend to make it.

Git List tag

This command is commonly employed to display all tags present in the repository.

Git Push tag

Users can push tags to a remote server, aiding in locating updates and serving as release markers on the server.

Git Delete tag

Git permits the removal of a tag from the repository at any time.

Git Tags: Operations

Below is the syntax for the **Git Create** tag and **Git List** tag:

Git Create

```
$ git tag <tag name>
```

Git List

```
$ git tag
```

Git Tags: Operations

Below is the syntax for the **Git Push** tag and **Git Delete** tag:

Git Push

```
$ git push origin <tagname>
```

Git Delete

```
$ git tag --delete <tagname>
```

Branching vs. Tagging

Here are the distinctions between branching and tagging in Git:

Aspects	Branching	Tagging
Definition	Branches in Git are pointers to specific commits, allowing for parallel development and isolation of features or fixes	Tags in Git are references to specific points in history, typically used to mark release points or significant milestones
Nature	Branches can move forward as new commits are added	Tags remain fixed at the point they are created
Example	Creating a feature branch to develop a new feature separately from the main branch	Tagging a commit to mark it as a stable release version

Assisted Practice

Creating and deleting tags **Duration: 10 Min.**

Problem statement:

You have been assigned a task to create and delete tags in a Git repository, helping users efficiently manage version control and maintain historical checkpoints within their software development projects.

Outcome:

By completing this demo, you will learn how to create and delete tags in a Git repository, enabling efficient version control and management of historical checkpoints in software development projects.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

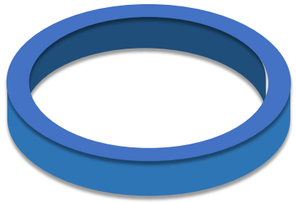
Steps to be followed:

1. Create a tag
2. Delete a tag

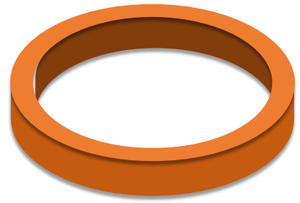
Stashing in Git

Git stash allows developers to temporarily store unfinished changes, which is useful when switching branches or performing operations without committing incomplete work.

Stashing involves two main actions:



Stash



Apply or Pop

Stashing in Git

Stash

The stash command preserves your local changes and resets the working directory to reflect the HEAD commit. It includes both staged and unstaged modifications.

Apply or Pop

Once changes are stashed, you have the option to either apply or pop them back onto your working directory. Applying retains the stash in the stack, whereas popping removes it.

Assisted Practice

Stashing in Git **Duration: 15 Min.**

Problem statement:

You have been assigned a task to use Git stashing and temporarily save uncommitted changes, allowing switch between branches without losing progress.

Outcome:

By completing this demo, you will learn the use of Git stashing to temporarily save uncommitted changes, enabling seamless switching between branches without losing progress.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines

Steps to be followed:

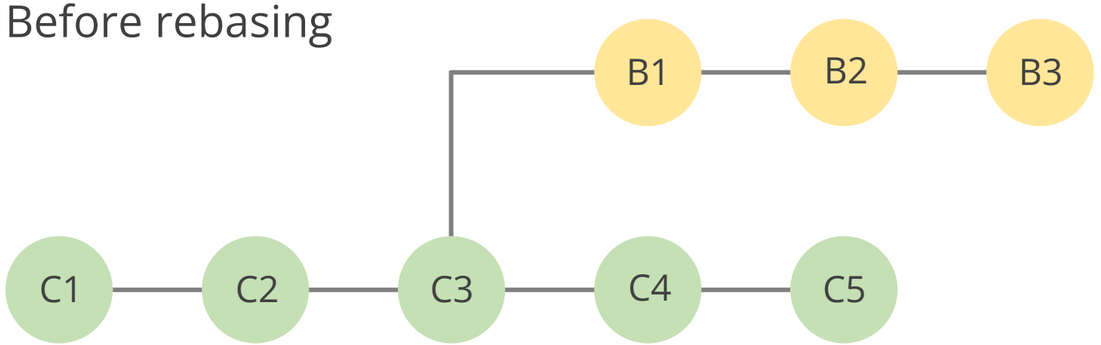
1. Set up the environment
2. Initialize Git and make initial commits
3. Manage branch transitions and changes
4. Stash changes and verify

ASSISTED PRACTICE

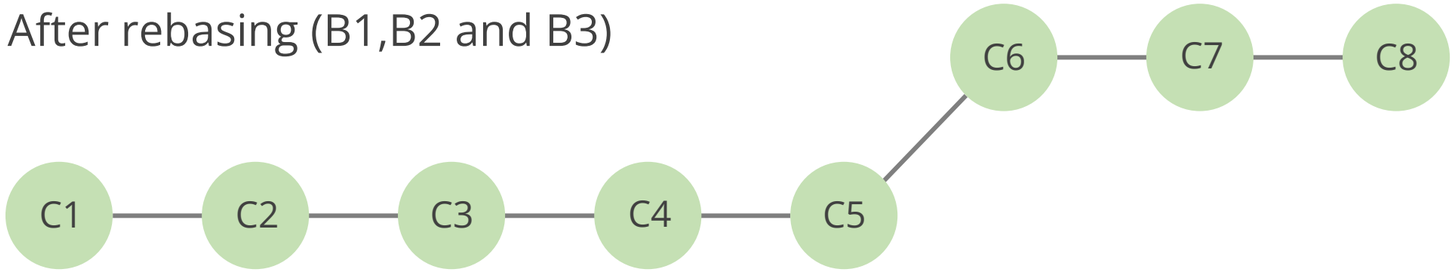
Rebasing in Git

It is a process of integrating a series of commits on top of another base tip. It takes all the commits of a branch and appends them to the commits of a new branch.

Before rebasing



After rebasing (B1,B2 and B3)



Rebasing in Git

It simplifies the project history, enabling seamless tracing from feature completion back to the start.

Syntax:

```
git rebase [-i | --interactive] [ options ] [--exec
cmd] [--onto newbase | --keep-base] [upstream
[branch]]
```

Assisted Practice

Rebasing in Git **Duration: 15 Min.**

Problem statement:

You have been assigned a task to perform rebase in Git for integrating changes from one branch to another while maintaining a linear commit history.

Outcome:

By completing this demo, you will learn how to perform a Git rebase to integrate changes from one branch to another, ensuring a linear commit history.

Assisted Practice: Guidelines

Steps to be followed:

1. Create a repository
2. Clone the Git repository and rebase the branch to integrate the changes

Quick Check

You are working on a feature in a Git project and realize the main branch has new updates. What is the main reason to use rebasing to include these changes in your branch?

A. To merge the changes from one branch into another without creating a merge commit

B. To duplicate a repository's content into a new repository

C. To create a backup of the current repository state

D. To track the changes made by different contributors in a repository

Undoing and Inspecting Changes in Git

Undoing and Inspecting Changes in Git

These are essential tasks for managing project revisions effectively. Below are the methods to undo and inspect changes in Git:

Git Revert

It is a Git command that generates a new commit to revert changes made in a repository, effectively undoing prior commits.

Syntax:

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

- Here, **<commit-hash>** is the identifier of the commit you want to revert.
- This command generates a new commit that reverses the changes made by a specified commit, effectively restoring the previous state while maintaining the commit history.

Git Revert Options

Git revert command provides various options to customize the undo operation according to specific requirements, including:

Commit identifier

To revert changes, specify the commit identifier of the changes you want to undo.

Multiple commits

Git revert allows reverting changes from multiple commits simultaneously by specifying their commit hashes.

Selective revert

You can selectively undo specific changes within a commit by staging or unstaging particular files or lines before executing the revert command.

Git Revert Options

Reverting merge commits

Git revert can handle reverting changes introduced by merge commits as well, ensuring flexibility in handling complex scenarios.

Dry run

Git revert implementations offer a dry run feature, enabling the previewing of changes before committing them to the repository.

Assisted Practice

Reverting to the previous commit **Duration: 15 Min.**

Problem statement:

You have been assigned a task to demonstrate reverting to the previous commit to undo changes and restore the project to a previous state.

Outcome:

By completing this demo, you will learn how to revert to a previous commit in Git, effectively undoing changes and restoring the project to a previous state.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new GitHub repository
2. Clone the GitHub repository
3. Revert to the previous commit

Git Reset

It is the command that users use when they want to move the repository back to a previous commit, discarding any changes made after that commit.

Syntax:

```
git reset [options] [<commit>] [--]

<paths>
```

Here,

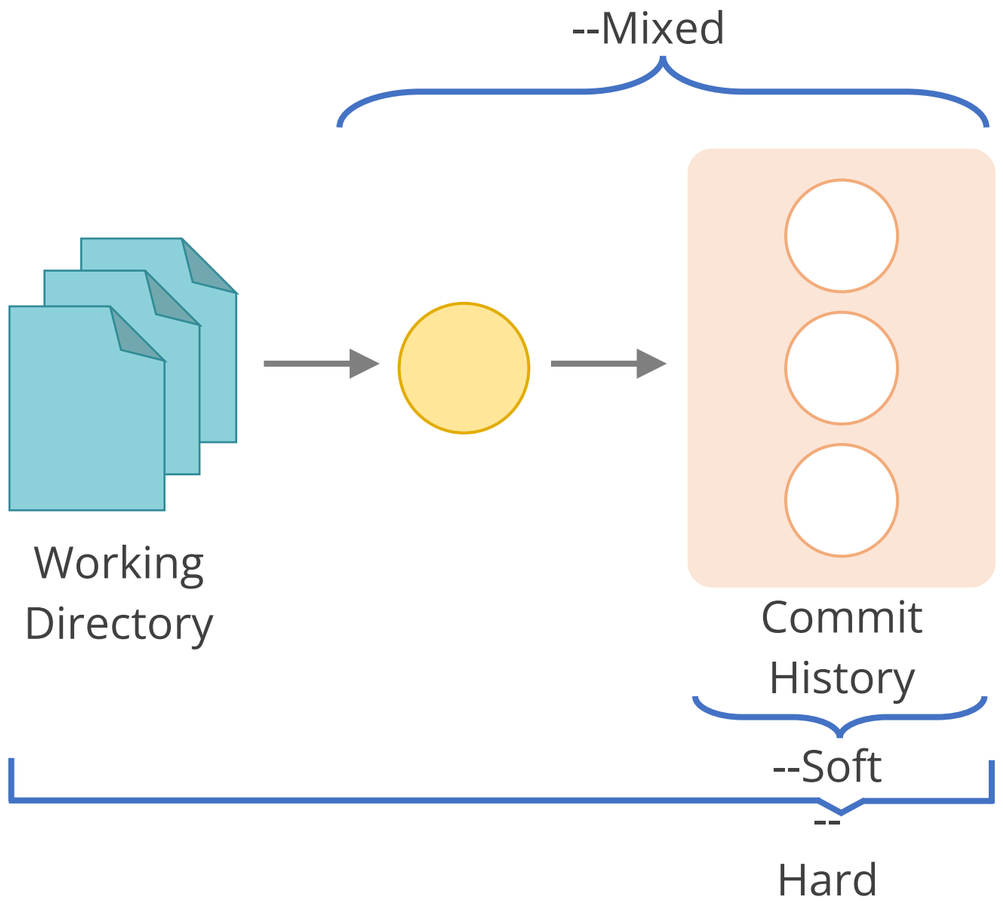
- **options:** Additional options for reset, such as --soft, --mixed, or --hard
- **<commit>:** The commit to which the HEAD should be moved. If omitted, defaults to HEAD
- **<paths>:** Optional path(s) to limit the reset operation to specific files or directories.

Git Reset

Git reset can function on entire commit objects or at a file level. These different reset options impact particular trees managed by Git to handle your file and its contents.

It has three core invocation forms:

1. Soft
2. Mixed
3. Hard



Git Remove

The **git rm** command removes files from both the working directory and the index (staging area) in Git, making it useful for erasing files from a repository's history.

Syntax:

```
git rm [options] <file(s)>
```

Here,

- **[options]:** Additional options for the Git rm command, which can modify its behavior
- **<file(s)>:** Specifies the file(s) to be removed from both the working directory and the index (staging area)

Git Remove

It offers several use cases, which include:

Supporting wildcards

Removing untracked files

Removing files from the index

Staging deletions

Git Diff

This command displays distinctions between the working directory and the staging area, between the staging area and the last commit, and between commits, branches, or other Git objects.

Syntax:

```
git diff [<options>] [<commit>] [--]

[<path>...]
```

Here,

- **options:** Parameters that modify git diff's behavior, like --cached, --stat, and --color
- **commit:** Specifies a commit to compare changes against, including hashes, branch names, or Git references
- **path:** Optional path(s) limiting output to changes in specific files or directories

Git Status

The **git status** command is used to show the status of the Git repository. It displays the state of the local directory and the staging area.

Syntax:

```
$ git status
```

Quick Check

As a developer, you have been updating a web application. You notice that recent changes in your code have caused errors. To revert your files to how they were at the last commit, which Git command should you use?

A. git diff

B. git rm

C. git status

D. git reset --hard

Business Impact Overview: Advanced Git Concepts

Issue tracking in Git ensures that all tasks, bugs, and feature requests are tracked and prioritized effectively in software development projects.

Example: Using GitHub issues to manage and prioritize tasks ensures that critical bugs are fixed promptly, and new features are developed according to business priorities.

Tagging in Git facilitates version control and deployment management by marking specific points in the project history.

Example: Tags are used to mark stable release versions of the web application. This practice allows the DevOps team to quickly identify and deploy stable versions to production.

Business Impact Overview: Advanced Git Concepts

Stashing in Git allows developers to save changes temporarily without committing them, enabling efficient context switching.

Example: When a critical issue arises during feature development, developers can stash their current work, fix the urgent bug, and then reapply the stashed changes to resume their work seamlessly.

Rebasing in Git keeps the project history clean and simplifies the integration of changes from multiple branches, reducing merge conflicts.

Example: By using rebasing, developers can maintain a linear project history, making it easier to integrate features from different branches. This practice reduces the complexity of merges, minimizes conflicts, and ensures that the CI/CD pipeline runs smoothly,

Combining issue tracking, tagging, stashing, and rebasing in a DevOps scenario ensures an efficient, organized, and flexible development process.

Key Takeaways

- Issue tracking in GitHub is a core feature designed to plan, discuss, and track the progress of tasks and bugs related to a project.
- Upstream typically represents the main project repository or the repository that you forked from, and it changes flow from the upstream repository to your local repository when you pull updates.
- Stashing in Git is a functionality enabling developers to temporarily store unfinished changes before committing them, which is beneficial when switching branches or executing operations without committing incomplete work.
- Git revert is a command utilized to reverse alterations in a Git repository by generating a new commit that negates the effects of commits, thereby reversing the modifications introduced by those commits.

Implementing Advance Operations in Git

Duration: 25 Min.

Project agenda: To execute advanced Git operations for enhanced version control and collaboration

Description: Imagine you are a software developer who has been asked to work on the Codex repository on GitHub. The goal is to solve issues using Git. First, create the Codex repo. Then, perform key Git tasks like tagging and branching. Understand Git rebase versus Git revert, explore Git log, and use Git rm with Git status. This project aims to improve teamwork and issue tracking for the Codex repository.

LESSON-END PROJECT

Implementing Advance Operations in Git

Duration: 25 Min.

Perform the following:

1. Create a new repository in Git
2. Create a tag in Git
3. Create a new branch in Git
4. Revert to the previous commit
5. Rebase the branch to integrate the changes
6. Remove the files from the Git index

Expected deliverables: Creation of a GitHub repository named Codex with specified operations executed

LESSON-END PROJECT

Thank You