PW SKILLS | DevOps and Cloud Computing

Advanced Gift

Objective

- Implement advanced branching and merging techniques.
- Resolve merge conflicts effectively.
- Work with remote repositories for collaboration.
- Compare popular Git repository hosting services (GitHub, GitLab, Bitbucket).
- Understand Git hooks, submodules, and subtrees for enhanced project management.
- Learn the benefits of open source contributions and how to contribute.

PW SKILLS

Demonstrating creating and switching branches

Let's do it

To create and switch branches in Git, you can use either the git branch command followed by git switch or the more efficient git switch command with its -c option.

Method 1: Using git branch and git switch

1. Create a new branch:

```
git branch new-branch
```

2. Switch to the new branch:

```
git switch new-branch
```

Let's do it

Method 2: Using git checkout

You can also create and switch to a new branch in one command using ‘git checkout’ with the -b flag:

```
git checkout -b new-branch
```

Method 3: Using git switch

With newer versions of Git, you can simplify this process using the ‘git switch’ command:

```
git switch -c new-branch
```

PW SKILLS

Explaining Merging Strategies

Fast-forward merge

Fast-forward merge is a type of Git merge that occurs when the branch being merged has no new commits compared to the target branch.

- In this scenario, Git simply moves the pointer of the target branch forward to the latest commit of the source branch, effectively integrating the changes without creating a new merge commit.

Key Points about Fast-Forward Merge:

- Linear History
- No Merge Commit
- When It Occurs: Fast-forward merges happen when:

The target branch has not diverged from the source branch.
The commit history is linear with no conflicting changes

Fast-forward merge

Example of Fast-Forward Merge:

1. Suppose you have a main branch and a feature branch:

```
A---B---C (main)
     \
      D---E (feature)
```

2. If no new commits are added to the main branch while working on the feature branch, merging will result in:

```
A---B---C---D---E (main, feature)
```

Fast-forward merge

Performing a Fast-Forward Merge:

To perform a fast-forward merge, you can use the following commands:

```
git checkout main
git merge feature
```

If fast-forwarding is possible, Git will automatically move the main branch pointer to include all commits from feature

Fast-forward merge

Preventing Fast-Forward Merges:

If you want to ensure that a merge commit is created even when a fast-forward is possible, you can use:

```
git merge --no-ff feature
```

This command forces Git to create a merge commit, providing more context in your project history

Pop Quiz

Q. Which command can be used to perform a fast-forward merge?

A

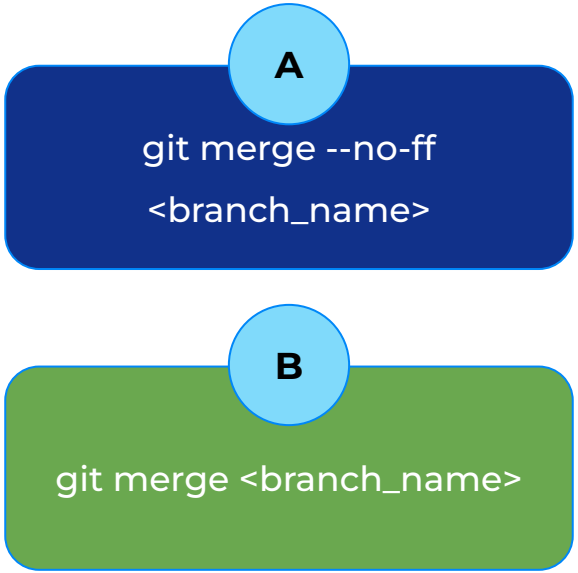
git merge --no-ff <branch_name>

B

git merge <branch_name>

Pop Quiz

Q. Which command can be used to perform a fast-forward merge?



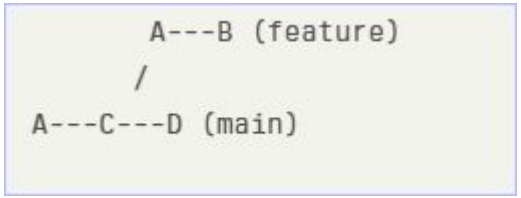
Three-way merge.

A three-way merge in Git is a method used to integrate changes from two branches that have diverged from a common ancestor. This process involves three key commits:

1. Base Commit
2. Source Commit
3. Target Commit

Visual Representation:

A: Common ancestor (base commit)
B: Latest commit on the feature branch
C: Latest commit on the main branch
D: New commit on the main branch after A



Three-way merge.

Merge Process:

When merging the ‘feature’ branch into ‘main’, Git identifies:

- Base: Commit A
- Source: Commit B (from the feature branch)
- Target: Commit D (from the main branch)

The resulting merge commit will combine changes from both branches, creating a new commit that ties them together, resulting in:

Pop Quiz

Q. What are the three key commits involved in a three-way merge?

A

Initial commit, final commit, merged commit

B

Base commit, source commit, target commit

Pop Quiz

Q. What are the three key commits involved in a three-way merge?

A

Initial commit, final commit, merged commit

B

Base commit, source commit, target commit

PW SKILLS

Hands-on examples of merging branches with sample code changes.

Let's do it

Scenario:

- Let's say you are working on a project with two branches: ‘main’ and ‘feature’. You want to add a new feature to your project while ensuring that the main branch remains stable.

Step-by-Step Example:

1. Create and Switch to the Feature Branch:

```
git checkout -b feature
```

2. Make Changes in the Feature Branch:

- Edit a file (e.g., app.py) to add your new feature. For example, you might add a function:

```
# app.py

def new_feature():
    print("This is a new feature!")
```

Let's do it

3. Stage and Commit Your Changes:

```
git add app.py
git commit -m "Add new feature function"
```

4. Switch Back to the Main Branch:

```
git checkout main
```

5. Make Changes in the Main Branch:

```
# app.py

def existing_function():
    print("This is an existing function.")
```

Let's do it

6. Commit Changes in Main Branch:

```
git add app.py
git commit -m "Add existing function"
```

7. Merge the Feature Branch into Main:

```
git merge feature
```

8. Resolve Conflicts (if any):

- If there are no conflicts, Git will automatically create a merge commit.

Let's do it

9. Complete the Merge:

```
git add app.py  # After resolving conflicts
git commit -m "Merge feature branch into main"
```

10. Push Changes to Remote Repository:

- Finally, push your changes to the remote repository.

```
git push origin main
```

PW SKILLS

What causes merge conflicts.

Merge Conflicts

Merge conflicts in Git occur when two or more branches have competing changes that cannot be automatically reconciled by Git. Here are the primary causes of merge conflicts:

1. Simultaneous Edits
2. File Deletion and Modification
3. Divergent Changes
4. Renaming Files
5. Pending Changes

PW SKILLS

Resolving conflicts manually in the command line or with a GUI tool.

Resolving conflicts manually

Resolving Conflicts Manually via Command Line:

1. Create a Merge Conflict:

First, create a scenario that leads to a merge conflict. For example, make changes in two branches.

```
# On the main branch
git checkout -b feature-branch
echo "Line from feature branch" > file.txt
git add file.txt
git commit -m "Add line from feature branch"

# Switch back to main and make conflicting changes
git checkout main
echo "Line from main branch" > file.txt
git add file.txt
git commit -m "Add line from main branch"
```

Resolving conflicts manually

2. Merge the Branches:

Attempt to merge the feature branch into the main branch.

```
git merge feature-branch
```

3. Identify Conflicts:

Git will indicate that there are conflicts. Use git status to see which files are in conflict.

```
git status
```

4. Edit the Conflicted File:

```
<<<<<<< HEAD
Line from main branch
=======
Line from feature branch
>>>>>>> feature-branch
```

Resolving conflicts manually

5. Resolve the Conflict:

```
Line from main branch
Line from feature branch
```

6. Stage and Commit the Resolved File:

After resolving the conflict, stage and commit your changes.

```
 git add file.txt
 git commit -m "Resolved merge conflict between main and
feature-branch"
```

Resolving conflicts manually

Resolving Conflicts Using a GUI Tool:

1. Open Your GUI Tool:
Use a Git GUI client like GitKraken, SourceTree, or any IDE with built-in Git support (e.g., IntelliJ IDEA).

2. Identify Conflicted Files:
The tool will typically show you a list of files with conflicts after attempting a merge.

3. Select a File to Resolve:
Click on the conflicted file to open a diff view that shows both versions of the changes.

Resolving conflicts manually

4. Choose Changes:
Use the GUI options to accept changes from either side (the local version or the incoming version) or manually edit the content as needed.

5. Finalize Resolution:
After resolving all conflicts, save your changes and follow prompts in the GUI to stage and commit your resolved files.

PW SKILLS

The Importance of remote repositories in collaboration.

Importance of remote repositories

Remote repositories are essential for collaboration in software development for several reasons:

1. Centralized Codebase
2. Efficient Collaboration
3. Version Control
4. Backup and Recovery
5. Enhanced Features
6. Support for Open Source Development

PW SKILLS

Demonstrating Pushing and pulling changes using git push and git pull

Pushing and Pulling changes

Pushing Changes with ‘git push’:

1.
```
git add <file>
git commit -m "Describe your changes"
```

2. Push Changes to Remote Repository:

```
git push origin main
```

- This command pushes changes from your local main branch to the remote main branch on the origin remote.

Pushing and Pulling changes

Pulling Changes with ‘git pull’:

1. Pull Latest Changes from Remote:

- Before pushing new changes, it’s good practice to pull the latest changes from the remote repository to ensure your local branch is up-to-date.

```
git pull origin main
```

- This command fetches updates from the remote ‘main’ branch and merges them into your local ‘main’ branch.

Pop Quiz

Q. If you try to push changes to a remote branch that has been updated since your last pull, what will Git do?

A

Reject your push and ask you to pull first.

B

Automatically merge the changes.

Pop Quiz

Q. If you try to push changes to a remote branch that has been updated since your last pull, what will Git do?

A

Reject your push and ask you to pull first.

B

Automatically merge the changes.

PW SKILLS

Fetching changes using ‘git fetch’ and rebasing with ‘git rebase’.

‘git fetch’ & ‘git rebase’

To effectively manage changes in Git, you can use ‘git fetch’ to retrieve updates from a remote repository and ‘git rebase’ to integrate those changes into your local branch.

Using git fetch

‘git fetch’ downloads commits, files, and references from a remote repository without merging them into your local branch. This allows you to review changes before integrating them. To fetch changes, use the following command:

```
git fetch origin
```

‘git fetch’ & ‘git rebase’

```
git log --oneline main..origin/main
```

This command shows the commits that exist in the remote ‘main’ branch but not in your local ‘main’ branch.

Using git rebase

After reviewing the fetched changes, you can use git rebase to apply your local commits on top of the fetched changes. This creates a linear history without merge commits.

```
git rebase origin/main
```

PW SKILLS

Brief example of cloning a remote repository

Cloning a remote repository.

To clone a remote Git repository, you can use the ‘git clone’ command. Here’s a brief example illustrating the process:

1. Copy the Repository URL: Navigate to the repository you want to clone on a platform like GitHub. Click on the Code button and copy the URL (either HTTPS or SSH).

2. Run the git clone Command:
```
git clone <repository-url>
```

3. Change Directory: After cloning, navigate into the newly created directory:
```
cd YOUR-REPOSITORY
```

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Compare popular services (GitHub, GitLab, Bitbucket)

Their unique features, pricing, and community support.

Unique features:

1. GitHub:

- Community-Driven
- GitHub Actions
- Extensive Integrations

2. GitLab:

- Comprehensive DevOps Platform
- Open Source Option
- Robust CI/CD Capabilities

Their unique features, pricing, and community support.

3. Bitbucket:

- Atlassian Integration
- Built-in CI/CD Pipelines
- Focus on Team Collaboration

Pricing

Feature/Plan	GitHub	GitLab	Bitbucket
Free	Unlimited public/private repos, 2,000 automation minutes/month	Unlimited public/private repos, all DevOps stages	Unlimited private repositories for small teams
Premium	Enhanced features for larger teams (pricing varies)	Advanced CI/CD and enterprise features (pricing varies)	Free public repositories; paid plans for additional features
Enterprise	GitHub Enterprise with advanced security (pricing varies)	Advanced security and compliance tools (pricing varies)	Enterprise support through Atlassian suite

Their unique features, pricing, and community support.

Community Support

GitHub:
Has the largest active community with extensive resources available for developers. It is particularly strong in open-source collaborations.

GitLab:
Growing community focused on DevOps practices. Provides robust documentation and forums but is smaller than GitHub's community.

Bitbucket:
Community support primarily comes from Atlassian resources. It has a dedicated user base but is less active compared to GitHub and GitLab.

PW SKILLS

Demonstrate creating and managing repositories on GitHub.

Lets do it

Creating a Repository:

1. Sign In: Log into your GitHub account.
2. Navigate to ‘New Repository’
3. Fill in Repository Details
4. Create Repository

Managing Your Repository:

- Adding Files
- Making Changes
- Branching and Merging
- Collaborating with Others

PW SKILLS

Explaining how to fork, clone, and create pull requests.

Lets do it

Forking a Repository:

1. Navigate to the Repository
2. Click the Fork Button

Cloning Your Fork:

1. Copy the Clone URL
2. Open Terminal
3. Run the Clone Command:

```
git clone https://github.com/yourusername/reponame.git
```

4. Navigate into the Directory:

```
cd reponame
```

Lets do it

Creating a Pull Request:

1. Make Changes Locally: After cloning, create a new branch for your changes:

```
git checkout -b new-feature
```

Make your changes and commit them:

```
git clone https://github.com/yourusername/reponame.git
```

2. Push Changes to Your Fork: Push your changes back to your forked repository on GitHub:

```
cd reponame
```

3. Create a Pull Request: Go to your forked repository on GitHub & Fill in any necessary details and click Create pull request.

Pop Quiz

Q. What is a pull request?

A

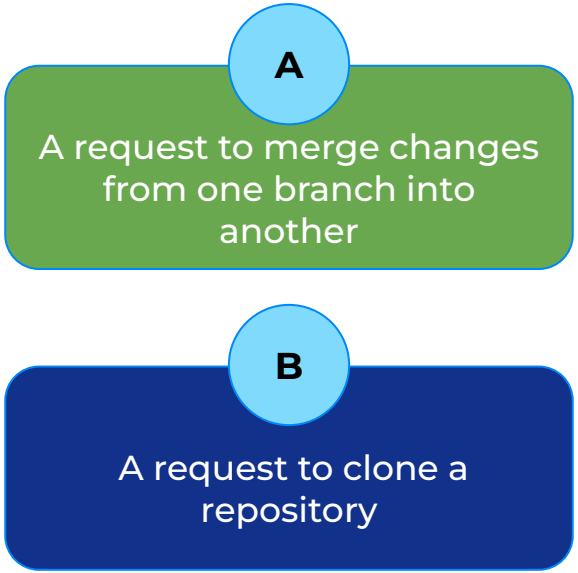
A request to merge changes from one branch into another

B

A request to clone a repository

Pop Quiz

Q. What is a pull request?



PW SKILLS

Introduce Git hooks

Introduction to Git Hooks

Git hooks are powerful scripts that enable automation and customization of Git's behavior at various stages of the version control process. These hooks can be triggered by specific events, allowing developers to enforce policies, automate tasks, and maintain code quality.

What Are Git Hooks?
Git hooks are scripts placed in the .git/hooks directory of a repository that execute automatically before or after certain Git commands, such as commits, merges, or pushes.

pre-commit and post-merge hooks with examples.

Pre-Commit Hook

The pre-commit hook is executed before a commit is finalized. It allows developers to run checks on the code that is about to be committed, ensuring it meets certain standards.

Example of a Pre-Commit Hook:

```
#!/bin/bash

# Run linting on Python files
flake8 .  # Check for style issues
if [ $? -ne 0 ]; then
    echo "Linting failed. Commit aborted."
    exit 1
fi

# Run tests
pytest  # Execute tests
if [ $? -ne 0 ]; then
    echo "Tests failed. Commit aborted."
    exit 1
fi

echo "Pre-commit checks passed."
```

pre-commit and post-merge hooks with examples.

Post-Merge Hook

The post-merge hook is triggered after a successful merge operation. This hook can be used to perform actions such as updating dependencies, running tests, or notifying team members about the merge.

Example of a Post-Merge Hook:

```
#!/bin/bash

# Notify team members about the merge
echo "Merge completed successfully. Notify team."

# Optionally, run tests to ensure everything works after the merge
pytest  # Execute tests after merging
if [ $? -ne 0 ]; then
    echo "Tests failed after merge. Check your changes."
fi
```

Demonstrating creating a simple Git hook script.

Step-by-Step Guide to Create a Pre-Commit Hook:

1. Navigate to Your Repository:

```
cd /path/to/your/repo
```

2. Create the Hook File:

```
touch .git/hooks/pre-commit
```

Demonstrating creating a simple Git hook script.

Step-by-Step Guide to Create a Pre-Commit Hook:

3. Edit the Hook Script:

```
#!/bin/bash

echo "Running pre-commit checks..."

# Check for uncommitted changes
if ! git diff --cached --quiet; then
    echo "You have uncommitted changes. Please commit or stash them
before proceeding."
    exit 1
fi

echo "Pre-commit checks passed."
```

4. Make the Hook Executable:

```
chmod +x .git/hooks/pre-commit
```

PW SKILLS

Explaining submodules

Demonstrating creating a simple Git hook script.

To add a Git submodule and update it, follow these concise steps:

Adding a Submodule

1. Navigate to Your Repository: `cd /path/to/your/repo`

2. Add the Submodule: `git submodule add <submodule_url>`

3. Commit the Changes: `git commit -m "Added submodule"`

Demonstrating creating a simple Git hook script.

Updating the Submodule

To update your submodule to the latest commit from its remote repository:

1. Initialize and Update (if it's a new clone):

```
git submodule init
git submodule update
```

2. Pull Changes from the Submodule:

```
cd path/to/submodule
git pull origin main  # Replace 'main' with the appropriate
name if needed.
```

Demonstrating creating a simple Git hook script.

3. Return to Main Repository:

```
cd ../..  # Adjust based on your directory structure.
```

4. Commit Any Changes:

```
git commit -m "Updated submodule"
```

PW SKILLS

Subtrees for managing dependencies within a repository.

Introducing subtrees

Git subtrees provide a method for managing dependencies within a repository by allowing you to nest one repository inside another as a subdirectory.

Key Features of Git Subtrees:

- Integration
- No Separate History
- Ease of Updates

Introducing subtrees

Basic Workflow for Using Git Subtrees:

1. Adding a Subtree:

```
git subtree add --prefix=<directory> <repository-url> <branch>
```

2. Updating a Subtree:

```
git subtree pull --prefix=<directory> <repository-url> <branch>
```

3. Pushing Changes Back:

```
git subtree push --prefix=<directory> <repository-url> <branch>
```

PW SKILLS

Benefits of: skill enhancement, networking, and portfolio building

Benefits of contributing to open source:

Contributing to open source offers several significant benefits, including skill enhancement, networking opportunities, and portfolio building. Here's a brief overview of each:

1. Skill Enhancement:
- Practical Experience
- Learning New Technologies
- Code Review

Benefits of contributing to open source:

2. Networking Opportunities:
   - Community Engagement
   - Mentorship
   - Visibility

3. Portfolio Building
   - Demonstrable Work
   - Diverse Projects
   - Reputation

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this session
- Review your lecture notes
- Note down questions and queries regarding this session and consult the teaching assistants

Thanks !