K.Madhusudhan Achary

[Madhu427/DevOpsClassCodes: skillfy test](https://github.com/Madhu427/DevOpsClassCodes)

Course code:102504

ASSIGNMENT GIT

A screenshot of a computer

AI-generated content may be incorrect.

**Task 1 & Task 2:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

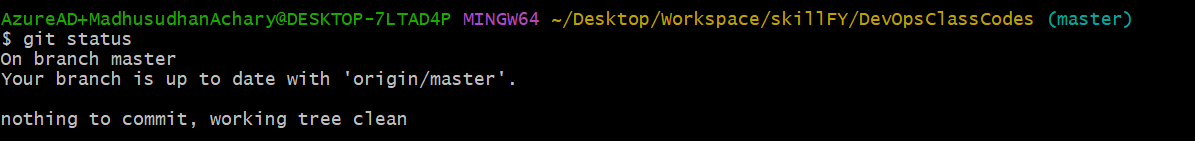
AI-generated content may be incorrect.

**Task 3: Branching Strategy**

The scenario states your team uses the **GitFlow branching strategy**. This strategy is a structured approach to managing your codebase.

**Step-by-step instructions:**

1. **Ensure you are on the develop branch:** The Git Flow model uses a main develop branch for all feature integration. Before you start, make sure your local repository is up-to-date with the remote develop branch.
   * First, switch to the develop branch: git checkout develop.
   * Then, pull the latest changes from the remote: git pull origin develop.



1. **Create a new feature branch:** All new work is done on a dedicated feature branch to keep it isolated from the main codebase. According to GitFlow, these branches are typically named feature/<feature-name>.
   * Create and switch to your new branch with one command: git checkout -b feature/your-feature-name.

A computer screen with text on it

AI-generated content may be incorrect.

1. **Start coding and committing:** Now you can begin working on the new feature. Make changes to the files and commit your work regularly.

* After making changes, add the files to the staging area: git add .
* Commit the changes with a clear message: git commit -m "f1.txt file added".

A screen shot of a computer screen

AI-generated content may be incorrect.

A black background with yellow text

AI-generated content may be incorrect.

1. **Push your feature branch to the remote repository:** To share your work with your team, you'll need to push your feature branch to the remote repository.

* Use the command: git push -u origin feature/your-feature-name. This sets up your local branch to track the remote branch.

A screen shot of a computer

AI-generated content may be incorrect.

**Task 4: Collaboration and Pull Requests**

Now that your feature is complete on its own branch, you need to get it reviewed and merged into the main codebase. This is done through a Pull Request (PR).

**Step-by-step instructions:**

1. **Go to your Git hosting service (e.g., GitHub):** After pushing your branch, the hosting service will often show a notification to create a new pull request. If not, navigate to the "Pull Requests" or "Merge Requests" tab.

A yellow rectangle with black text

AI-generated content may be incorrect.

1. **Create a new Pull Request:**
   * Select your feature branch (feature/your-feature-name) as the "source" or "compare" branch.
   * Select the develop branch as the "target" or "base" branch.
   * Provide a clear and descriptive title for your PR (e.g., "new file f1.txt added").
   * Write a detailed description explaining the changes you made, why they were needed, and what a reviewer should look for.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Request a code review:**
   * Assign the PR to one or more teammates for a code review.
   * Collaborate with your teammates, addressing any comments or suggestions they have by making new commits to your feature branch.
   * Once the reviewers approve your changes, the PR can be merged.
2. **Merge the Pull Request:**
   * Once the code is approved and passes all necessary checks (like CI tests), an authorized person can click the "Merge" button.
   * This will merge your feature branch's changes into the develop branch.
   * After the merge, you can safely delete the feature branch both locally and on the remote.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Task 5: Handling Merge Conflicts**

Merge conflicts occur when changes made in two different branches clash. This scenario happens when a teammate pushes conflicting changes to main while you're

**Step-by-step instructions:**

1. **Update your local main or develop branch:**
   * Before merging your feature branch, always ensure your target branch (master) is up to date.
   * Switch to the develop branch: git checkout develop.
   * Pull the latest changes: git pull origin develop.
2. **Merge develop into your feature branch:**
   * Switch back to your feature branch: git checkout feature/your-feature-name.
   * Merge the latest changes from develop into your feature branch: git merge develop.
   * \* If Git can't automatically merge, it will notify you of a conflict in a specific file. The file will contain conflict markers like `<<<<<<<`, `=======`, and `>>>>>>>`.
3. **Resolve the conflicts manually:**
   * Open the conflicted file in your code editor.
   * The markers show the changes from your current branch (<<<<<<< HEAD) and the incoming changes (>>>>>>> develop).
   * Manually edit the file to keep the correct code, which might be one or both sets of changes.
   * Delete the conflict markers.
4. **Commit the resolved changes:**
   * After resolving the conflicts in the file, add the file to the staging area: git add <conflicted-file-name>.
   * Commit the merge: git commit -m "Resolve merge conflict in <file-name>".
   * Push your branch with the resolved conflicts to the remote: git push origin feature/your-feature-name.

**Task 6: Creating a Release and Tagging**

This task requires you to create a **tag** for a specific version of your application, which is a common practice for marking release points.

**Step-by-step instructions:**

1. **Ensure you are on the correct branch:** Releases are typically created from a stable branch, such as main or master. First, switch to that branch and make sure it's up to date.
   * git checkout master
   * git pull origin master
2. **Create a new tag:** You can create a lightweight or annotated tag. It is best practice to use **annotated tags** for releases, as they store more metadata like the tagger name, date, and a message.
   * Use the command: git tag -a v1.0 -m "Release version 1.0".
   * The -a flag creates an annotated tag, and the -m flag adds a message. A common naming convention is to use a v prefix followed by the version number, like v1.0.
3. **Push the tag to the remote repository:** By default, git push does not push tags. You must explicitly push them to the remote repository.
   * Push the specific tag you just created: git push origin v1.0.

A black screen with colorful text

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**A screenshot of a computer

AI-generated content may be incorrect.**

**Task 7: Hotfix in Production**

1. **Create a hotfix branch:** Hotfixes are worked on in a dedicated branch, usually created directly from the stable release point (v1.0 in this case). This prevents the hotfix from including any new features that are currently in development.
   * git checkout -b hotfix/critical-bug v1.0
   * This command creates a new branch named hotfix/critical-bug and checks it out, with its starting point at the v1.0 tag.

A computer screen with yellow text

AI-generated content may be incorrect.

1. **Make and commit the fix:**
   * Make the necessary code changes to fix the bug.
   * Add the changes to the staging area: git add .
   * Commit the fix with a clear message: git commit -m "Hotfix line added in Read me"

A screen shot of a computer code

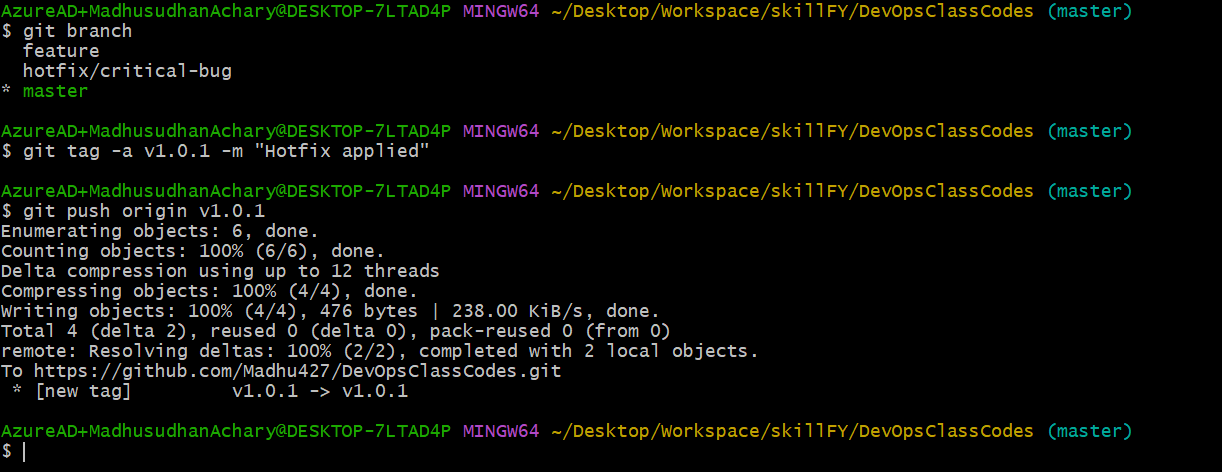
AI-generated content may be incorrect.

1. **Merge the hotfix back to the stable branch:** After the fix is complete, you must merge it into your stable branch (e.g., main).
   * git checkout master
   * git merge hotfix/critical-bug

A computer code on a black background

AI-generated content may be incorrect.

1. **Tag the new release:** After applying the hotfix, you should create a new tag to mark this new version. Use a patch version number, such as v1.0.1.
   * git tag -a v1.0.1 -m "Hotfix applied"
   * Push the new tag to the remote repository: git push origin v1.0.1



1. **Clean up:** Once the hotfix has been merged into all necessary branches and tagged, you can delete the hotfix branch.
   * git branch -d hotfix/critical-bug (local)

A screen shot of a computer code

AI-generated content may be incorrect.

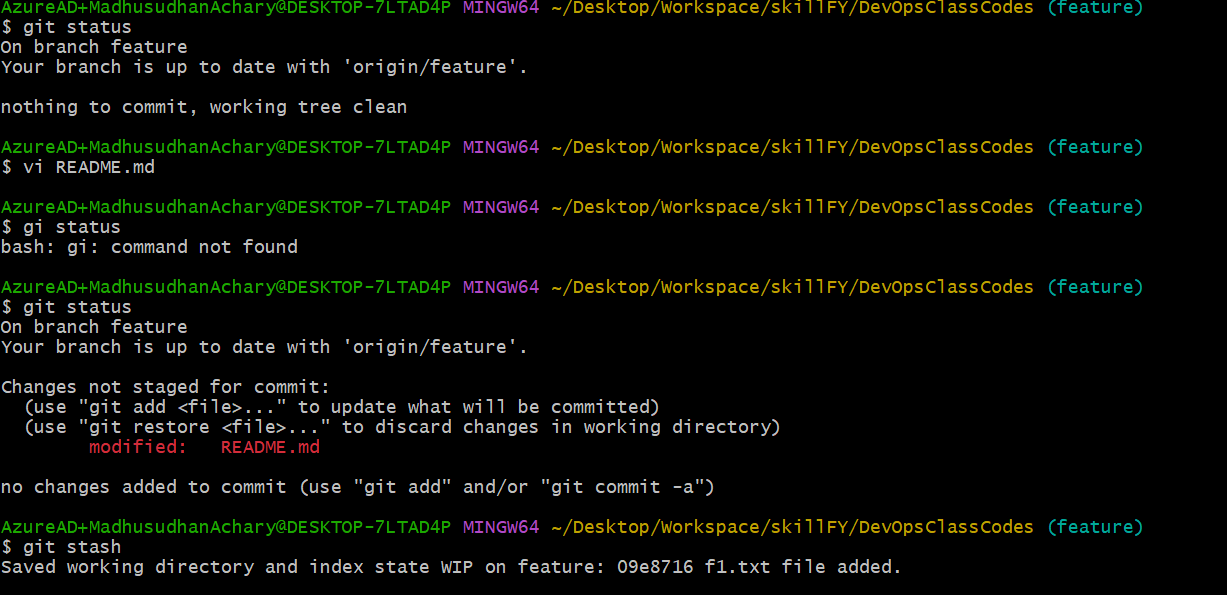
A screenshot of a computer

AI-generated content may be incorrect.

**Task 8: Stashing Changes**

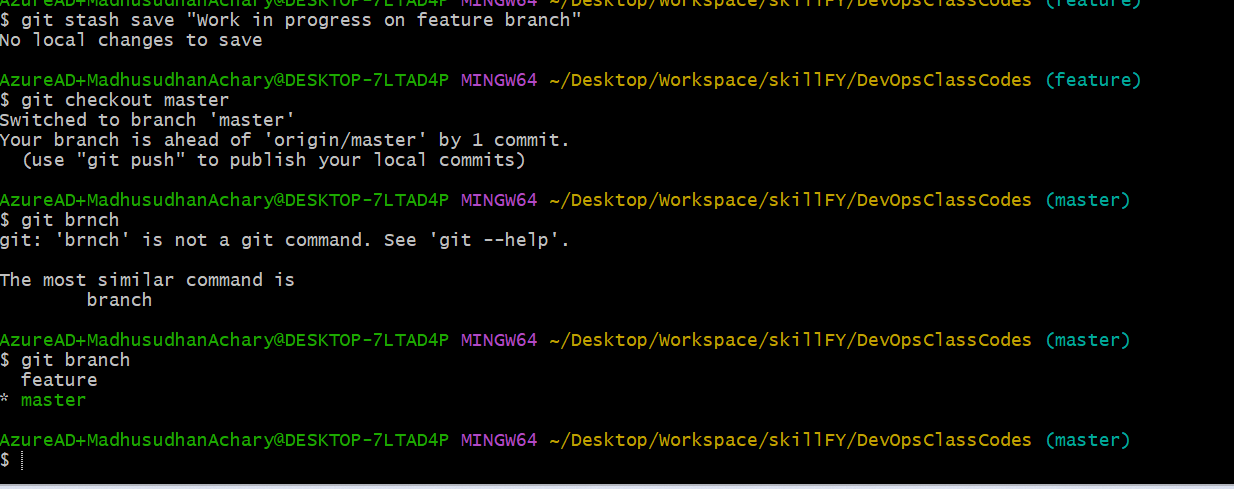
Step-by-step instructions:

1. Check your status: Before you do anything, check the status of your working directory to see which files you have modified.
   * Use the command: git status
2. Stash your changes: To save your uncommitted changes, use the git stash command. This cleans your working directory and adds all modified files to a temporary "stash" list.
   * Use the command: git stash save "Work in progress on feature branch"
   * The optional message helps you remember what the stash contains, which is especially useful if you have multiple stashes.

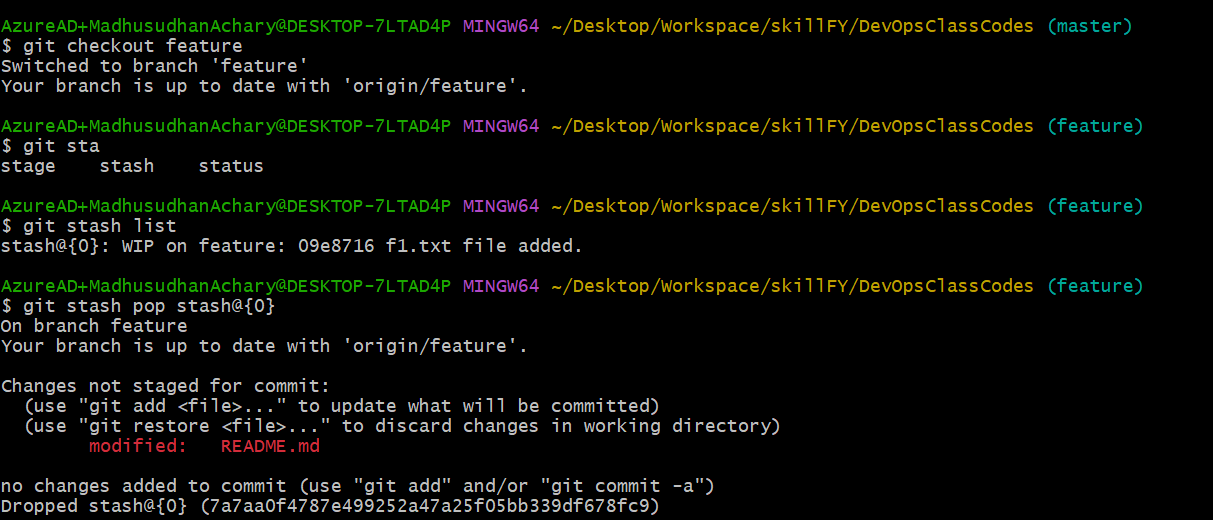


1. **Switch branches and handle the urgent issue:** Now that your working directory is clean, you can safely switch to the new branch to address the urgent issue.

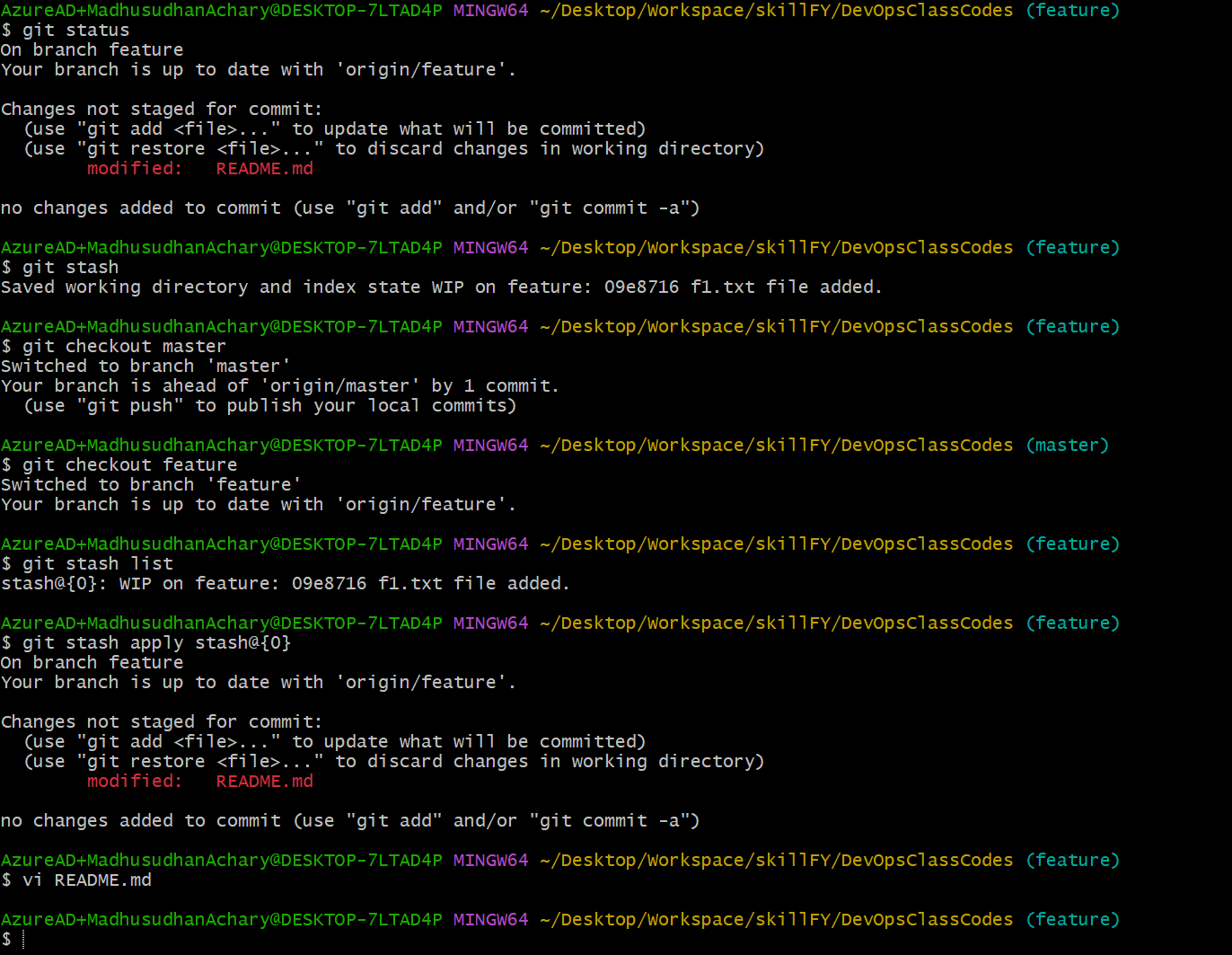
* Use the command: git checkout master
* Make the necessary changes, commit them, and push them to the remote repository.



1. **Return to your original branch and restore your work:** After you've finished with the urgent issue, you can switch back to your original branch and restore your stashed changes.
   * Switch back: git checkout feature
   * To see your stash(es): git stash list
   * To restore your latest stashed changes: git stash pop
   * git stash pop applies the changes and removes them from the stash list. If you need to keep the changes in the stash, you can use git stash apply instead.

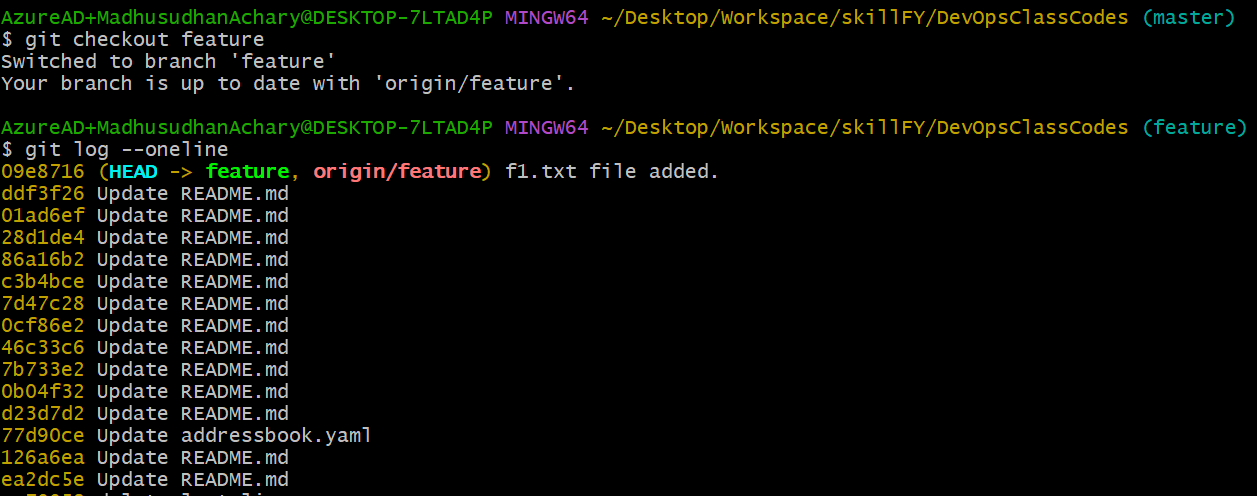


Git stash apply

****

**Task 9: Cherry-Picking a Commit**

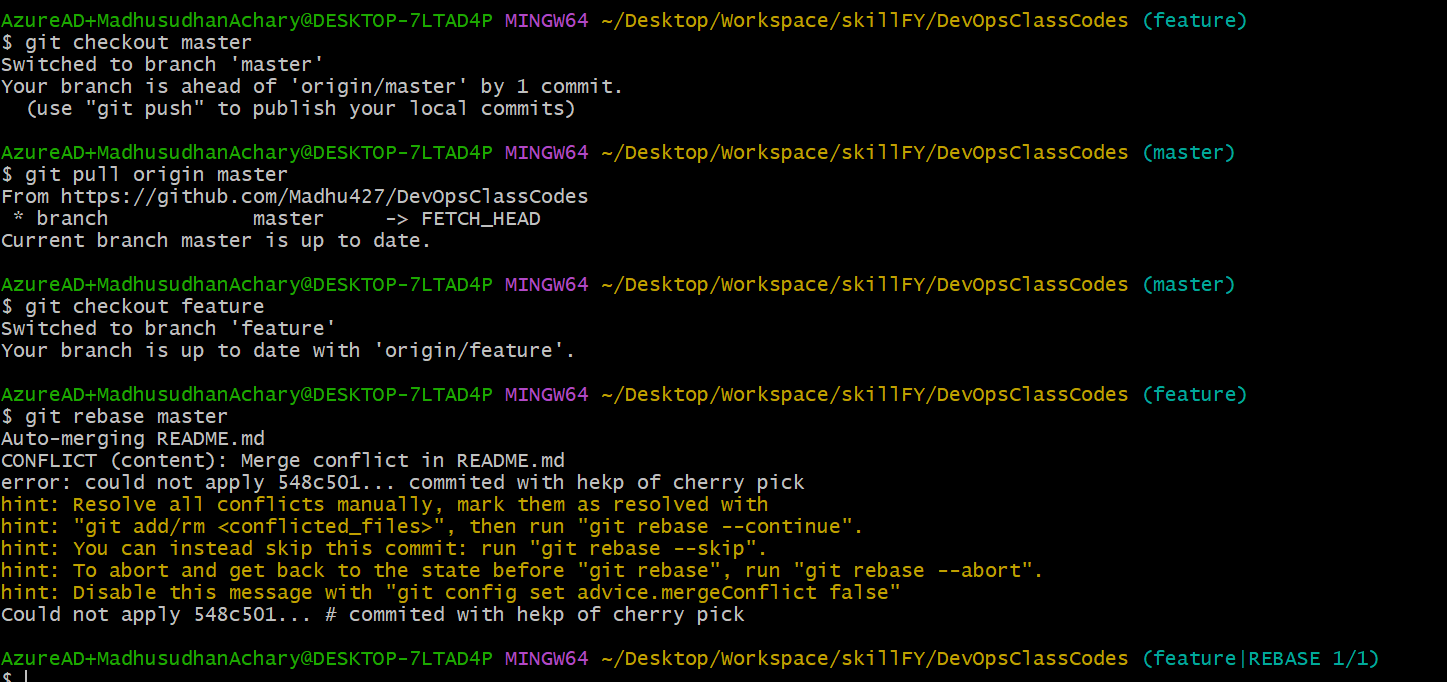
1. Find the commit hash: First, you need to find the unique identifier (commit hash) of the commit you want to copy. You can do this by looking at the commit history of the other branch.
   * Switch to the branch where the desired commit is located: git checkout <other-branch-name>
   * View the commit history with git log --oneline. This will show you a list of commits with their abbreviated hashes. \* Copy the hash of the commit you want to cherry-pick.
2. Switch to your target branch: Now, switch to the branch where you need the fix to be applied.
   * Use the command: git checkout feature
3. Cherry-pick the commit: Use the git cherry-pick command followed by the commit hash you copied in the first step.
   * Use the command: git cherry-pick <commit-hash>
   * Git will now apply that single commit to your current branch. This may cause a merge conflict if the changes clash with your current work. If a conflict occurs, you must resolve it manually before you can proceed.
4. Push the new commit: After cherry-picking, the new commit will be part of your current branch's history. You can now push your changes to the remote repository.
   * Use the command: git push origin feature



**A screenshot of a computer program

AI-generated content may be incorrect.**

1. Ensure your main branch is up to date: Before you rebase, you need to have the latest changes from the remote main branch.
   * git checkout master
   * git pull origin master
2. Switch to your feature branch: Now, switch back to your feature branch where you've been working.
   * git checkout feature
3. Perform the rebase: Run the git rebase command to reapply your branch's commits on top of the main branch's latest commit.
   * git rebase master
   * This command moves your entire feature branch to start at the latest point of master, making it look as though you started your work from there. This creates a linear, clean history without extra merge commits.

****

**Task 12: Squashing Commits**

1. Start an interactive rebase: You will use an interactive rebase to "squash" the commits. You need to know how many commits you want to combine.
   * Let's say you have the last 3 commits you want to squash. Use the following command:
   * git rebase -i HEAD~3
   * The -i flag starts the interactive session. HEAD~3 tells Git to look at the last three commits from your current position.



**A screenshot of a computer program

AI-generated content may be incorrect.**