

Revert Code to Previous

February 19, 2024

Anonymous

I have a Bitbucket repo with a dev-django branch.
I created a new branch using `git checkout -b aws_functions`.
Then I did some changes to the code.
Then `git add`
Then `git commit -m "Created function"`
`git push origin aws_functions`

Then I opened my Bitbucket repo and created a pull request to merge the code to dev-django branch. I merged the pull request. Now the new code is in the dev_django branch.

The new code turned out to be problematic so I need to go back to the old state of my code. How to do that, explain in detail?

ChatGPT

Sure, here's a step-by-step guide to revert your code back to the state before you made the changes:

- Check Out the Branch: First, ensure you're on the branch where you want to revert the changes. In your case, it seems like you want to revert the changes in the "dev-django" branch. You can do this by running:

```
bash git checkout dev-django
```

- Find the Commit to Revert: Next, you need to find the commit that introduced the changes you want to revert. You can do this by looking at the commit history using:

```
bash git log
```

This will show you a list of commits. Find the commit that corresponds to the changes you want to revert. Each commit will have a unique commit hash (a long string of characters), and you'll need this hash for the next step.

- Revert the Commit: Once you've identified the commit you want to revert, you can use the `git revert` command followed by the commit hash. For example:

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

This command will create a new commit that undoes the changes introduced by the specified commit.

- Resolve Conflicts (if any): Git may prompt you to resolve conflicts if the changes you're reverting conflict with other changes in your codebase. If this happens, you'll need to manually resolve the conflicts in the affected files.

- Commit the Revert: After resolving any conflicts, you need to commit the revert changes:

```
bash git commit -m "Revert changes introduced by commit <commit-hash>"
```

- Push the Reverted Changes: Finally, push the reverted changes to your Bitbucket repository:

```
bash git push origin dev-django
```

After completing these steps, your "dev-django" branch should be reverted to its previous state, undoing the changes introduced by the problematic commit. Make sure to thoroughly test your code to ensure everything is working as expected. If you need further assistance or encounter any issues, feel free to ask!

Anonymous

but initially when I merged the pull request, I also deleted the aws_functions branch. How to proceed in that case?

ChatGPT

If you've already merged the changes from the "aws_functions" branch into the "dev-django" branch and then deleted the "aws_functions" branch, you can still revert the changes using a similar process. Here's how you can do it:

- Identify the Reverted Commit: Since you've merged the changes from "aws_functions" into "dev-django", you need to find the commit that merged those changes. You can use `git log` to view the commit history and find the merge commit. The merge commit will have two parent commits, one from "dev-django" and one from "aws_functions".
- Revert the Merge Commit: Once you've identified the merge commit, you can revert it using `git revert`. Use the commit hash of the merge commit when reverting:

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

- Resolve Conflicts (if any): As before, resolve any conflicts that arise during the revert process.

- Commit the Revert: After resolving conflicts, commit the revert changes:

```
bash git commit -m "Revert merge commit <merge-commit-hash>"
```

- Push the Reverted Changes: Finally, push the reverted changes to your Bitbucket repository:

```
bash git push origin dev-django
```

This process will effectively undo the changes introduced by the merge commit, effectively reverting your code back to its state before merging the "aws_functions" branch into "dev-django". Once done, ensure to thoroughly test your code to ensure everything is working as expected. Let me know if you need further clarification or assistance!

Anonymous

can you explain this step in detail with real world example and resolution
Resolve Conflicts (if any): As before, resolve any conflicts that arise during the revert process.

ChatGPT

Certainly! When you revert a commit, Git will attempt to apply the changes introduced by that commit in reverse. However, conflicts may arise if Git cannot cleanly apply the changes due to subsequent changes made in the codebase.

Let's walk through an example:

- Identify the Conflict: Suppose you're reverting a commit that added a function to a file, but since then, someone else has made changes to the same file, including modifications to the function you're reverting.
- Git Status: After running `git revert <commit-hash>`, Git may report that there are conflicts that need to be resolved. You can check the status of your repository using `git status`.
- Resolve Conflicts: Open the conflicted file in your code editor. You'll see sections of code surrounded by `<<<<<`, `>>>>>`, and `>>>>>` markers. These sections represent the conflicting changes. Your goal is to manually edit the file to resolve these conflicts.

For example:

```
jsreact <<<<< HEAD
```

```
// Code from your current branch (the changes you want to keep)
function foo() {
    // Original implementation
}

// Code from the commit you're reverting (the changes you want to remove)
function foo() {
    // New implementation added in the reverted commit
    console.log('New implementation');
}>>>>> <commit-hash>
```

You'll need to decide which changes to keep. In this case, you may want to keep the original implementation and remove the new implementation added by the reverted commit. So, you would edit the file to remove the lines added by the reverted commit and keep the original implementation.

- Mark as Resolved: After editing the conflicted file(s), you need to mark them as resolved. You can do this by staging the changes using `git add <file>`.

- Complete Revert Commit: Once all conflicts are resolved, you can complete the revert commit by running `git commit`. Git will automatically generate a commit message indicating that you've resolved conflicts during the revert process.

- Continue Revert Process: If there are multiple conflicts in different files, repeat steps 3-6 until all conflicts are resolved.

- Push Reverted Changes: After resolving all conflicts and committing the changes, you can push the reverted changes to the remote repository using `git push origin <branch-name>`.

By following these steps, you'll effectively resolve any conflicts that arise during the revert process, ensuring that your codebase is in a clean and functional state.

Anonymous

can you explain git cherry pick in detailed way with real world problem example and resolution

ChatGPT

Certainly! `git cherry-pick` is a powerful Git command that allows you to apply specific commits

from one branch onto another branch. This can be particularly useful when you want to selectively apply changes from one branch to another without merging the entire branch.

Let's walk through an example to illustrate how 'git cherry-pick' works:

Scenario:

Suppose you have a 'feature-x' branch where you've been working on implementing new features for your project. Meanwhile, there's a critical bug that needs to be fixed on your 'main' branch. Fortunately, someone has already fixed this bug and committed the fix on the 'hotfix' branch.

Problem:

You need to apply the fix from the 'hotfix' branch onto your 'main' branch without merging the entire 'hotfix' branch, as it contains other changes that are not ready to be included in 'main' yet.

Resolution:

1. **Identify the Commit:** First, you need to identify the commit that contains the fix on the 'hotfix' branch. You can do this by checking the commit history using 'git log' or by viewing the branch on a platform like GitHub or Bitbucket.

2. **Check out the Target Branch:** Switch to the branch where you want to apply the fix. In this case, it's the 'main' branch:

```
bash git checkout main
```

Copy code

3. **Cherry-pick the Commit:** Use the 'git cherry-pick' command followed by the commit hash of the fix from the 'hotfix' branch:

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

Copy code

4. **Resolve Conflicts (if any):** Git may prompt you to resolve conflicts if the changes you're cherry-picking conflict with other changes in your codebase. Resolve these conflicts as you would during a merge or revert process.

5. **Commit the Changes:** After resolving conflicts, commit the changes introduced by the cherry-pick:

```
bash git commit
```

Copy code

6. **Push the Changes:** Finally, push the changes to your remote repository:

```
bash git push origin main
```

Copy code

By following these steps, you've successfully applied the fix from the 'hotfix' branch onto your 'main' branch using 'git cherry-pick'. This allows you to selectively incorporate specific changes from one branch into another, helping you manage your codebase more effectively.



Anonymous
great, give some other usecase that is completely different from this



ChatGPT
Certainly! Here's another use case for 'git cherry-pick':

Scenario:

Suppose you're working on a project with multiple feature branches, each implementing a different feature. You've completed a feature on the 'feature-A' branch and merged it into the 'development' branch. However, you realize that one of the commits from 'feature-A' is also needed in another feature branch 'feature-B', but you don't want to merge the entire 'feature-A' branch into 'feature-B'.

Problem:

You need to selectively apply a specific commit from the 'feature-A' branch onto the 'feature-B' branch without merging the entire branch.

Resolution:

1. **Identify the Commit:** Determine the commit hash of the specific commit you want to cherry-pick from the 'feature-A' branch. You can use 'git log' to view the commit history.

2. **Check out the Target Branch:** Switch to the branch where you want to apply the commit. In this case, it's the 'feature-B' branch:

```
bash git checkout feature-B
```

Copy code

3. **Cherry-pick the Commit:** Use the 'git cherry-pick' command followed by the commit hash of the desired commit from the 'feature-A' branch:

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

Copy code

4. **Resolve Conflicts (if any):** If there are conflicts between the changes introduced by the cherry-picked commit and the existing code on the 'feature-B' branch, resolve them as needed.

5. **Commit the Changes:** After resolving conflicts, commit the changes introduced by the cherry-pick:

```
bash git commit
```

Copy code

6. **Push the Changes:** Finally, push the changes to your remote repository:

```
bash git push origin feature-B
```

Copy code

By following these steps, you've successfully cherry-picked a specific commit from the 'feature-A' branch onto the 'feature-B' branch. This allows you to selectively incorporate individual commits across branches, providing flexibility in managing your project's codebase.



Anonymous
give an example of a conflict during the above usecase. explain the problem and give the resolution to that



ChatGPT
Certainly! Let's continue with the previous scenario where we cherry-pick a commit from the 'feature-A' branch into the 'feature-B' branch. Suppose there's a conflict between the changes introduced by the cherry-picked commit and the existing code on the 'feature-B' branch.

Problem:

The commit you're cherry-picking from 'feature-A' modifies a function in a file that has also been modified in 'feature-B' since the branches diverged. As a result, Git cannot automatically apply the changes, and a conflict arises.

Resolution:

1. **Identify the Conflict:**

Let's say the function 'calculate_total()' in a file 'utils.py' was modified in both branches: in the 'feature-A' branch:

```
python def calculate_total(items): total = 0 for item in items: total += item.price return total
```

Copy code

In the 'feature-B' branch:

```
python def calculate_total(items): total = 0 for item in items: total += item.quantity return total
```

Copy code

Git detects a conflict because both branches modified the same function differently.

2. **Resolve the Conflict:**

To resolve the conflict, you need to decide which changes to keep. You might want to combine the changes from both branches or choose one version over the other.

For example, you could modify 'utils.py' to calculate the total based on both the 'price' and 'quantity' attributes:

```
python def calculate_total(items): total = 0 for item in items: total += item.price * item.quantity return total
```

Copy code

After resolving the conflict, save the file.

3. **Mark as Resolved:**

Once you've resolved the conflict in 'utils.py', stage the changes to mark the conflict as resolved:

```
bash git add utils.py
```

Copy code

4. **Continue Cherry-Pick:**

Now that the conflict is resolved, you can continue the cherry-pick process by completing the cherry-

pick commit:

```
git cherry-pick --continue
```

Git will create a new commit with the resolved changes.

5. Push the Changes:

```
git push origin feature-B
```

By resolving the conflict in 'utils.py' and completing the cherry-pick you've successfully applied the changes from 'feature-A' to 'feature-B', ensuring that both branches are up to date with the desired modifications.

Anonymous
for the above example, show what i will get to see during the conflict with <<<<<, =====, and >>>>> markers

ChatGPT
Sure, let's break down the conflict markers ('<<<<<', '=====', and '>>>>>') in the 'utils.py' file during the conflict resolution:

In the 'feature-A' branch:

```
def calculate_total(items):
    total = 0
    for item in items:
        <<<<< HEAD
        total += item.price
        =====
        total += item.price * item.quantity
        >>>>> feature-A
    return total
```

In the 'feature-B' branch:

```
def calculate_total(items):
    total = 0
    for item in items:
        <<<<< HEAD
        total += item.price * item.quantity
        =====
        total += item.price
        >>>>> feature-B
    return total
```

Explanation:

- The '<<<<< HEAD' marker indicates the beginning of the changes from the current branch ('feature-B' in this case).
- The '===== ' marker separates the changes from the current branch ('feature-B') and the changes from the branch being merged in ('feature-A').
- The '>>>>> feature-A' marker indicates the end of the changes from the branch being merged in ('feature-A').
- The lines between '<<<<< HEAD' and '===== ' are changes from the current branch ('feature-B').
- The lines between '===== ' and '>>>>> feature-A' are changes from the branch being merged in ('feature-A').

To resolve the conflict, you'll need to manually edit the file to combine the changes as desired. After resolving the conflict, the conflict markers ('<<<<<', '=====', and '>>>>>') and the lines between them should be removed, leaving you with the desired code.

Report content Terms of use Privacy policy

G Get citation