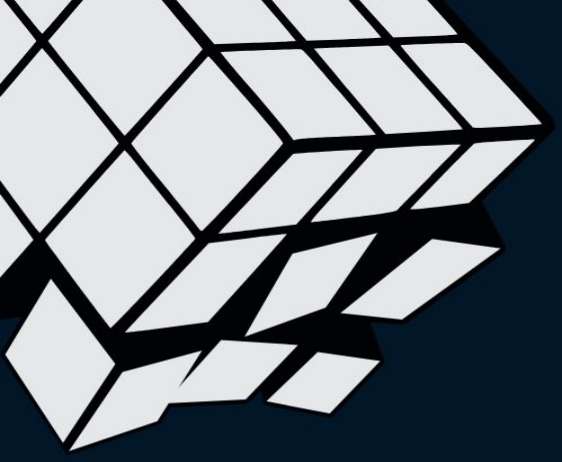


INTRODUCTION TO GIT & GITHUB

Table of Contents

Git Merge

Once the developers are ready with their changes, they can merge their feature branches into the main branch and make those features live on the website.

If you followed the steps from the previous chapter, then your `newFeature` branch is now ahead of the main branch with 1 commit. So in order to get that new changes over to the main branch, we need to merge the `newFeature` branch into our `main` branch.

Merging a branch

You can do that by following these steps:

- First switch to your `main` branch:

```
git checkout main
```

- After that, in order to merge your `newFeature` branch and the changes that we created in the last chapter, run the following `git merge` command:

```
git merge newFeature
```

Output:

```
Updating ab1007b..a281d25
Fast-forward
 feature1.html | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature1.html
```

As you were on the `main` branch when you ran the `git merge` command, Git will take all of the commits from that branch and merge them into the `main` branch.

Now, if you run the `ls` command, you will be able to see the new `feature1.html` file, and if you check the commit history with the `git log` command, you will see the commit from the `newFeature` branch present on your `main` branch.

Before doing the merge, you could again use the `git diff` command to check the differences between your current branch and the branch that you want to merge. For example, if you are currently on the `main` branch, you could use the following:

```
git diff newFeature
```

In this case, the merge went through smoothly as there were no merge conflicts. However, if you are working on a real project with multiple people making changes, there might be some merge conflicts. Essentially this happens when changes are made to the same line of a file, or when one developer edits a file on one branch and another developer deletes the same file.

Resolving conflicts

Let's simulate a conflict. To do so, create a new branch:

```
git checkout -b conflictDemo
```

Then edit the `feature1.html` file:

```
echo "<p>Conflict Demo</p>" >> feature1.html
```

The command above will echo out the `<p>Conflict Demo</p>` string, and thanks to the double grater sign `>>`, the string will be added to the bottom of the `feature1.html` file. You can check the content of the file with the `cat` command:

```
cat feature1.html
```

Output:

```
<h1>My First Feature Branch</h1>
<p>Conflict Demo</p>
```

You can again run `git status` and `git diff` to check what exactly has been modified before committing.

After that, go ahead and commit the change:

```
git commit -am "Conflict Demo 1"
```

Note that we did not run the `git add` command, but instead, we used the `-a` flag, which stands for `add`. You can do that for files that have been added to git and have just been modified. If you've added a new file, then you would have to stage it first with the `git add` command.

Now go switch back to your `main` branch:

```
git checkout main
```

And now, if you check the `feature1.html` file, it will only have the `<h1>My First Feature Branch</h1>` line as the change that we made is still only present on the `conflictDemo` branch.

Now let's go ahead and make a change to the same file:

```
echo "<p>Conflict: change on main branch</p>" >> feature1.html
```

Now we are adding again a line to the bottom of the `feature1.html` file with different content.

Go ahead and stage this and commit the change:

```
git commit -am "Conflict on main"
```

Now your `main` branch and the `conflictDemo` branch have changes to the same file, on the same line. So let's run the `git merge` command and see what happens:

```
git merge conflictDemo
```

Output:

```
Auto-merging feature1.html
CONFLICT (content): Merge conflict in feature1.html
Automatic merge failed; fix conflicts and then commit the
result.
```

As we can see from the output, the merge is failing as there were

changes to the same file on the same line, so Git is unsure which is the correct change.

As always, there are multiple ways to fix conflicts. Here we will go through one.

Now if you were to check the content of the `feature1.html` file you will see the following output:

```
<h1>My First Feature Branch</h1>
<<<<<<< HEAD
<p>Conflict: change on main branch</p>
=======
<p>Conflict Demo</p>
>>>>>>> conflictDemo
```

Initially, it could be a little bit overwhelming, but let's quickly review it:

- `<<<<<<< HEAD`: this part here indicates the start of the changes on your current branch. In our case, the `<p>Conflict: change on main branch</p>` line is present on the `main` branch, which is also the branch that we've currently switched to.
- `=======`: this line indicates where the changes from the current branch end and where the changes from the new branch are coming from. In our case, the change from the new branch is the `<p>Conflict Demo</p>` line.
- `>>>>>>> conflictDemo`: this indicates the name of the branch that the changes are coming from.

You can resolve the conflict by manually removing the lines that are not needed, so at the end, the file will look like this:

```
<h1>My First Feature Branch</h1>
<p>Conflict: change on main branch</p>
```

In case that you are using an IDE like VS Code, for example, it will allow you to choose which changes to keep with a click of a button.

After resolving the conflict, you will need to make another commit as the conflict is now resolved:

```
git commit -am "Resolve merge conflict"
```

Conclusion

Git branches and merges allow you to work on a project together with other people. One important thing to keep in mind is to make sure that you pull the changes to your local `main` branch on a regular basis so that it does not get behind the remote one.

A few more commands which you might find useful once you feel comfortable with what we've covered so far are the `git rebase` command and the `git cherry-pick` command, which lets you pick which commits from a specific branch you would like to carry over to your current branch.

Cloning a repository in VS Code

The good thing is that Visual Studio will auto-detect if you've opened a folder that is a repository. If you've already opened a repository, it will be visible in the Source Control View.

If you haven't opened a folder yet, the Source Control view will give you the option to Open Folder from your local machine or Clone Repository.

If you select Clone Repository, you will be asked for the URL of the remote repository (for example, on GitHub) and the parent directory under which to put the local repository.

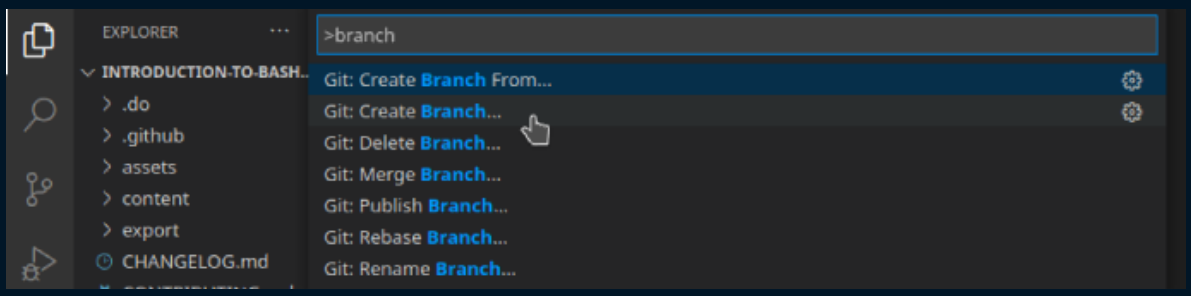
For a GitHub repository, you would find the URL from the GitHub Code dialog.

Create a branch

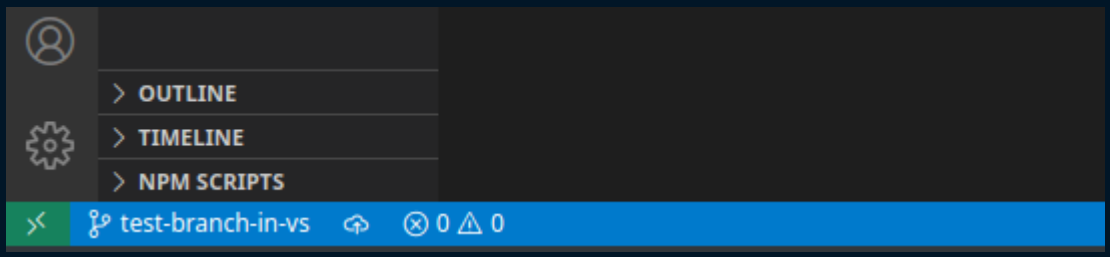
To create a branch open the command pallet:

- Windows: Ctrl + Shift + P
- Linux: Ctrl + Shift _ P
- MacOS: Shift + CMD + P

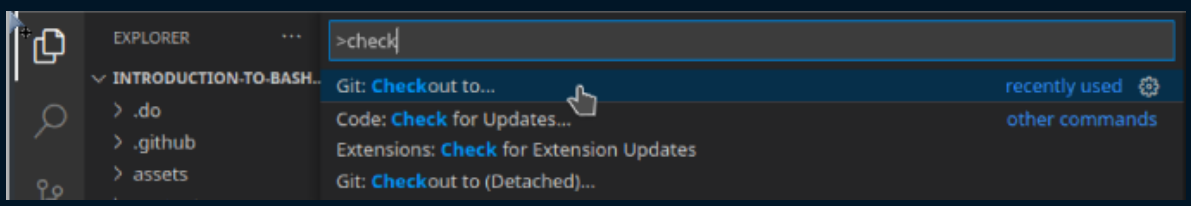
And select `Git Create Branch...`



Then you just need to enter a name for the branch. Keep in mind that in the bottom left corner, you can see in which branch you are. The default one will be the main, and if you successfully create the branch, you should see the name of the newly created branch.



If you want to switch branches, you can open the command pallet and search for `Git checkout to` and then select the main branch or switch to a different branch.



Setup a commit message template

If you want to speed up the process and have a predefined template for your commit messages, you can create a simple file that will contain this information.

To do that, open your terminal if you're on Linux or macOS and create the following file: .gitmessage in your home directory. To create the file, you can open it in your favorite text editor and then simply put the default content you would like and then just save and exit the file. Example content is:

```
cat ~/.gitmessage
```

```
#Title

#Summary of the commit

#Include Co-authored-by for all contributors.
```

To tell Git to use it as the default message that appears in your editor when you run `git commit` and set the `commit.template` configuration value:

```
$ git config --global commit.template ~/.gitmessage
$ git commit
```

Conclusion

If you prefer to code in Visual Studio Code and you also use version control, I will recommend you to give it a go and interact with the repositories in VS code. I believe that everyone has their own style, and they might do things differently depending on their mood as well. As long as you can add/modify your code and then commit your changes to the repository, there is no exactly correct/wrong way to achieve this. For example, you can edit your code in vim and push the changes using the git client in your terminal or do the coding in Visual Studio and then commit the changes using the terminal as well. You're free to do it the way you want it and the way you find it more convenient as well. I believe that using git within VS code can make your workflow more efficient and robust.