Git Coding Dojo

# Introduction

This coding dojo is designed around the use of git versioning control solution. You will use git to create a local repository, make changes to the repository content, and synchronize work in a team.

The dojo has been designed around the use of git as a command line tool and will not cover usage of third-party software that can be used to manage git repositories (GitKraken, Visual Studio, …).

In this document, you will find all commands that needs to be executed for each chapter. The commands are displayed with the following style:

this is a command that should be executed

The dojo uses files that are provided in a zip archive along these instructions. During some chapters, you will need to retrieve specific files from the archive and copy them in your repository to simulate development of the source code. When files should be copied, you will find instructions with the following style:

Copy files from the **start** folder provided for the dojo.

# Basics

In this chapter, you will perform the following tasks:

* Create your local repository
* Add content to your repository
* Save changes to your repository using version control

## Create & initialize your repository

In this section, you will create an empty local repository which will be used for the first chapters of the dojo. Use the following command to create the repository:

git init git-coding-dojo

cd git-coding-dojo

This last command can be used at any point in your repository to get a summary of the status of the repository.

git status

## First commits

In this section, you will add the basic template of the application and do some changes to the source code, while using version control to track and store the changes.

Copy files from the **start** folder provided for the dojo.

Check the status of the repository and add files to prepare your first commit.

git status

git add index.html

git add style.css

git status

git commit

Enter a message to explain the changes made to the source code and close your editor. Then check the status of the repository. The second command will display a simple graph of the history of your repository.

git status

git log --oneline --graph

Now you will make some changes to the source code.

Copy files from the **1.1** folder provided for the dojo.

You can check the index.html web page in a browser to see your changes. Then, check the status of the repository.

git status

Add the updated index.html to prepare the next commit and use the new commit option to directly provide the commit message.

git add index.html

git commit -m "Add 1.1 paragraph"

git status

git log --oneline --graph

## Amend a commit

This section will show you how you can easily change your last commit if you made a mistake or forgot a file.

Copy files from the **1.2** folder provided for the dojo.

Check the index.html page to visualize the changes made to the web page, then commit the changes to your repository.

git status

git add .

git status

git commit -m "Add 1.2 text"

Note the hash of the commit that was just created, to compare after the next step.

Copy files from the **1.3** folder provided for the dojo.

Check the index.html page to visualize the added changes, then use the commit command with the amend option to modify the latest commit.

git status

git add .

git commit --amend

You can check the commit log and see that no new commit was added to the history. Compare the last commit hash with the one you noted from the previous step and see that it has been updated.

git log --oneline --graph

# Branches

In this chapter, we will start to use branches to organize our work on the source code of the application.

You will perform the following tasks:

* Create a new branch
* Switch branches
* Merge changes from a branch in another branch
* Resolve conflicts between branches

## Working on a branch

In order to organize our work, we will avoid working directly on the master branch. Create a new branch named develop that will be our main branch for new developments.

The branch command allows you to manage branches in your repository. You can list them, create, and delete branches from the repository.

git branch --list --verbose

git branch develop

git branch -lv

Now that the branch exists, change your current working branch to the newly created develop branch.

git checkout develop

git branch -lv

Now, you will do some modifications to the source code on the develop branch.

Copy files from the **2.1** folder provided for the dojo.

Check the index.html page to see your modifications and commit the changes to the repository.

git commit -a -m "Add 2.1 paragraph"

git branch -lv

Observe the differences between the two branches. Your modifications have only been added to the develop branch. You can see the difference by displaying the log for all branches.

git log --all --oneline --graph

You can switch back to the master branch and see that your modifications are not here.

git checkout master

Check the index.html page and see that master still has the version from the previous chapter.

## Merge changes from your branch

Once your modifications are complete, you will want to integrate them in the master branch. This is done with a merge operation. Switch back to the master branch If you didn’t on the previous section.

git checkout master

git merge develop

As the develop branch was only ahead of the master branch, the merge operation is a simple one, called a ‘fast-forward merge’. Check the branches point to the same commit.

git branch -lv

The merge operation has correctly updated the master branch to match the develop branch.

## Merge with conflicts

You will once again add modifications on the develop branch, but this time the master branch will change before the merge, which should make the merge operation more complicated.

Switch back to the develop branch to make your modifications.

git checkout develop

Copy files from the **2.2** folder provided for the dojo.

Check the index.html page to view the modifications then commit your changes. You can again see that the develop branch has been updated.

git commit -m "Add 2.2 paragraph on develop"

git branch -lv

Switch back to the master branch to make different changes.

git checkout master

Copy files from the **2.3** folder provided for the dojo.

Check the index.html page to see the master modifications then commit the changes to the master branch.

git commit -m "Add 2.3 paragraph on master"

git branch -lv

Display the commit log to visualize the two divergent commits on develop and master branch from a shared history.

git log --all --oneline --graph

This time, when doing the merge, you will have some conflicts to resolve as the same file has been modified on both branches.

git merge develop

If you check the repository status, you will see that git has prepared your working folder with changes from both changes that should be resolved before creating a commit for the merge operation.

git status

We have a set of files with the resolution of the conflict ready for this exercise.

Copy files from the **2.4** folder provided for the dojo.

Check the index.html page to see that changes form both branches have been integrated in our new version.

You can then add the resolved version for the next commit, check that all conflicts are resolved, and commit the merge operation to the master branch.

git add index.html

git status

git commit

You can leave the merge commit message as-is or change it if you want.

Display the commit log to see how the merge operation is indicated, with both branches as parents for the new commit.

git log --all --oneline --graph

Once you have finished, delete the local develop branch. You will recreate it later when working with a remote repository.

git branch -d develop

# Remotes

Remotes are a key part of git when working in a team on the same product. They are mainly used as a central shared repository that will allow the team to synchronize its work and run CI/CD pipelines.

In this chapter, you will perform the following tasks:

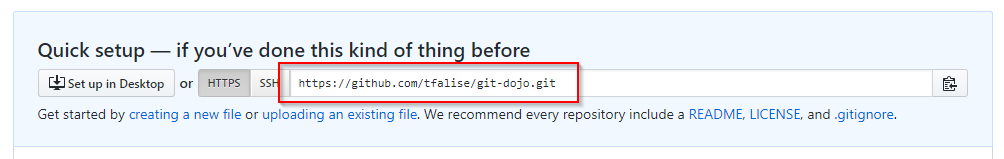
* Setup a remote repository on Github
* Commit some changes locally and share it with the remote repository
* Create branches on the remote repository
* Create local branches to track the remote branches

## Setting up a remote

In this section, you will create a new repository on Github and update it with the current version of your application.

Open Github and create a new account if you don’t have one already. Then, create a new repository for this coding dojo.

Once created, copy the repository url that will be used to setup the remote on your local repository.



Go back to your console and add the remote to your local repository with the following command (replace the @url parameter with the url copied from the Github repository).

git remote add origin @url

git push --set-upstream origin master

If you look at your repository on Github, you should now see that it has been initialized with the latest code from your application.

Check the branches on your local repository and see that you now have a local reference to the branches from the remote repository.

git branch -alv

## Local and remote branches

Now, you will do some changes on your local repository and see how you can push them to the shared remote repository.

Copy files from the **3.1** folder provided for the dojo.

Check the index.html page to visualize the new modifications made to the application, then commit the changes locally.

git commit -a -m "Add 3.1 paragraph"

git status

See on the status of the repository that your local master branch is now ahead of the remote one by 1 commit. You can also see on the branch list and log that the local and remote branches do not point to the same commit.

git branch -alv

git log --all --oneline --graph

If you check on the Github website, you can also see that the repository does not have the changes from your last commit.

Push your changes with the following command and see that your branches are now up to date.

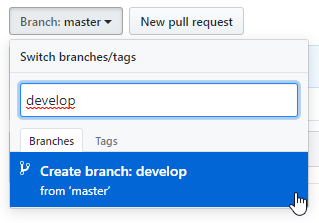
git push origin master

git branch -alv

## Using remote branches

Branches are not always created from your local repository. You will often need to work locally on a branch that exists on the remote, but not in your local version of the repository.

Create a develop branch on the Github repository from the web interface.



Go back to your console and check the list of available branch references on your repository.

git branch -alv

The develop branch has been created on the Github repository but is not yet available locally. You need to update your references from the remote repository. This is called a fetch operation.

git fetch origin

Now that your remote references are up to date, you can again check the available branches and switch to the develop branch.

git branch -alv

git checkout develop

Even if the branch develop does not exist currently, git will automatically create it and setup the branch to track the corresponding remote branch. You can view the tracking information with the following command.

git branch -lvv

# Working in Team

In this chapter, we will simulate two colleagues working on the same source code. Each one of them will use a different local repository.

You will perform the following tasks:

* Clone a remote repository
* Create feature branch for each developer
* Use pull requests to integrate code in the develop branch
* Retrieve commits from the develop branch in your feature branch

## Working in team - Setup

Each developer will need its own local repository to work isolated from the other one. You will first create a repository for Bob, using the clone operation to create a clone of the remote repository locally.

Use the following commands (don’t forget to replace the @url parameter with the url from your Github repository).

cd ..

git clone @url bob-dojo

You will setup this repository with a specific user name and email to differentiate the contributions of both developers on Github.

cd bob-dojo

git config user.name "Bob"

git config user.email "bob@dojo"

Next, create another clone of your Github repository for Alice, the second developer, and configure it with a specific user name and email.

cd ..

git clone @url alice-dojo

cd alice-dojo

git config user.name "Alice"

git config user.email "alice@dojo"

You now have two different repositories tracking the same remote repository.

## Working in team - Bob

You will start by doing some modifications on the source code for Bob. Get into the folder corresponding to Bob’s repository and switch to the develop branch.

cd ..\bob-dojo\

git checkout develop

Create a new branch for your modifications, to isolate your changes from the develop branch, which will serve as the integration branch shared by the team.

git checkout -b feature/bob

The -b option used on the checkout option will automatically create the branch if it does not exist when trying to switch branches.

You can view all branches with the following command.

git branch -lvv

Now, add the modifications for Bob to the feature branch.

Copy files from the **4.1** folder provided for the dojo.

Check the updated index.html page and commit your changes to the local branch.

git commit -a -m "Add bob content in a new paragraph"

Now that your changes have been committed locally, push these to the remote repository. This will automatically create the corresponding branch on the remote repository.

git push --set-upstream origin feature/bob

git branch -lvv

You can see on the branch details after your push that the remote branch has been created and is pointing to the same commit than the local branch.

## Working in team - Alice

You will now do some other changes to the source code from Alice’s repository. Follow the same steps you just did for Bob.

First, create your feature branch from the develop branch.

cd ..\alice-dojo\

git checkout develop

git checkout -b feature/alice

You can retrieve the references from the remote repository and see that Bob’s branch is indeed available form a remote branch reference.

git fetch origin

git log --all --oneline --graph

Now, add the source code changes for Alice.

Copy files from the **4.2** folder provided for the dojo.

Check the index.html page and see the changes made to the page stylesheet. Commit the changes and push them to the remote repository.

git commit -a -m "Update CSS style"

git push --set-upstream origin feature/alice

Have a look at the commit log after your push.

git log --all --oneline --graph

You can see both branches from Bob and Alice, available on the remote repository. See how they diverge from the develop branch.

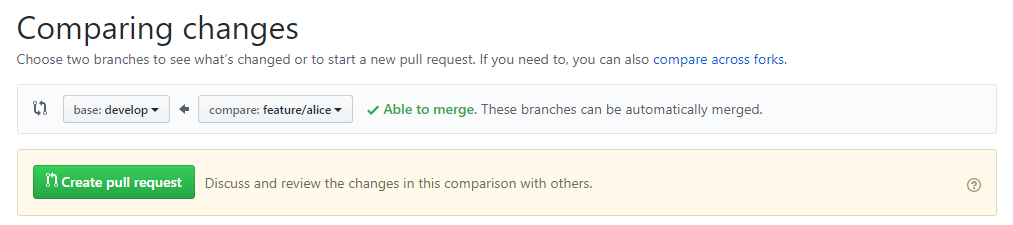
You can also navigate to your Github repository to explore both branches from the web interface.

## Working in team - Merge Alice’s changes

Now that both branches contain different changes, it’s time to merge these changes in the develop branches.

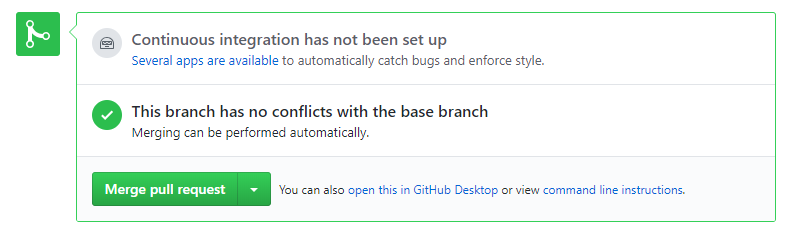
You will use the ‘pull request’ mechanism provided by Github to manage the integration of the source code changes in the develop branch.

Navigate to Github pull request section and create a new pull request, from the feature/alice branch to the develop branch.

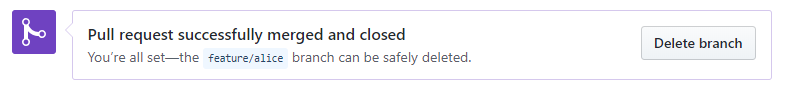


The pull request will serve as a gating mechanism that will allow the review of the modifications before merging them in the develop branch. This is the ideal moment for quality checks, code review, …

As our pull request has only one commit, we can merge it directly in the develop branch.



Once the pull request has been completed and the source code modifications have been merged, you can delete the feature branch on Github.



Now that the branch has been deleted, you need to update your local branch references.

git fetch

git branch -alvv

You can see that even if the remote branch has been deleted, the local branch persists. You can use the following option to remove branches which remote has been deleted.

git fetch --prune

git branch -alvv

You can now go back to your develop branch and update it from the remote.

git checkout develop

git merge origin/develop

As the remote branch is only ahead by 1 commit, the merge operation is a ‘fast-forward’ merge. Now that all branches, local and remote, are up to date, you can delete the local feature branch.

git branch -d feature/alice

Display the commit log and see that the commit from Alice’s branch has been correctly merged into develop branch.

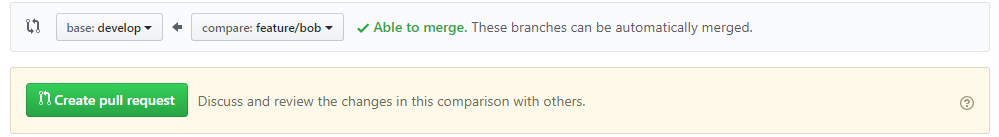
git log --all --oneline --graph

## Working in team - Merge Bob’s changes

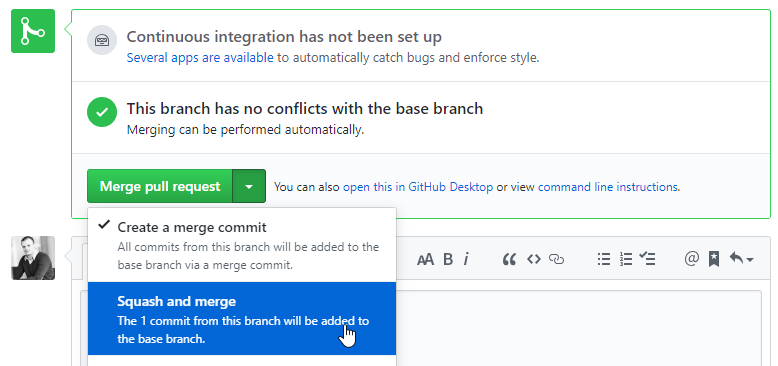
Now that Alice’s work has been integrated into develop branch, you will integrate Bob’s work. Move to Bob’s repository to execute the next steps.

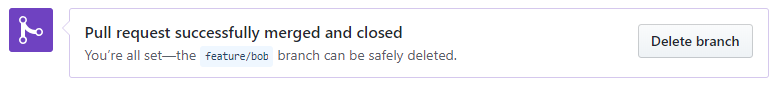
cd ..\bob-dojo\

From the Github web interface, create a new pull request to integrate work from feature/bob branch into develop branch.

There should be no conflict for your pull request, allowing you to complete it right now. When completing the pull request, select the ‘squash’ option for the merge.

The squash will combine all commits that were done on the source branch in a single commit that will be added to the target branch.

Now that the pull request has been completed, you can delete the feature/bob branch from the remote repository using Github web interface.

Go back to your local repository and update your local develop branch.

git checkout develop

git pull origin

git log --all --oneline --graph

The pull command here will execute a fetch command and a merge from the tracked remote branch. Once the squash commit from the remote has been properly retrieved locally, you can delete the local feature branch.

git branch -d feature/bob

Notice that git will not allow you to remove the branch, as the commit on the develop branch is not the same than the one on the feature branch. Git will warn you that you may lose work when deleting the branch.

As we know that the work has been properly integrated with the pull request, you can force the deletion with the following option.

git branch -D feature/bob

You can now check the commit log and observe that all source code changes are properly integrated in the develop branch.

git log --all --oneline --graph

# Release & Production

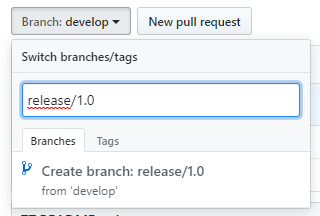
In this chapter, you will execute the following actions:

* Create a dedicated branch to prepare a release of the application
* Do some bugfix on the release branch
* Rebase your commits
* Apply a hotfix in production
* Cherry-pick your hotfix from production into the release branch

## Prepare your first release

Time has come to prepare the first release of your application. To prepare the release without interrupting the development activities, you will create a dedicated branch on Github that will be used for release activities.

On the Github web interface, create a release/1.0 branch from the develop branch.



Go back to Alice’s repository and fetch the branch references from the remote to prepare the work on the release.

cd ..\alice-dojo\

git fetch origin

git checkout release/1.0

Do the same for Bob’s repository

cd..\bob-dojo\

git fetch origin

git checkout release/1.0

Bob will be the first developer to add some last-minute content that should be included in the release.

Copy files from the **5.1** folder provided for the dojo.

Check the index.html web page to view the new content, commit locally and push your changes to the remote repository.

git commit -a -m "Add last content for 1.0 release"

git push origin

## Bugfix on the release branch

Now that Bob has pushed his last-minute modification, Alice will also need to contribute to your first release. Go back to Alice’s working directory for the next steps.

cd ..\alice-dojo\

Add the changes from Alice to the working folder.

Copy files from the **5.2** folder provided for the dojo.

Check the changes made to the stylesheet and commit them locally.

git commit -a -m "Last CSS update for 1.0 release"

Fetch the latest commits from remote repository and have a look at the commit log.

git fetch origin

git log --all --oneline --graph

See that the remote release branch and Alice’s local release branch have diverging commits. Try to push your commits to the remote repository.

git push origin

Git will prevent your push, as your last commit does not have the proper parent. You will need to update your local branch to include the missing commits from the remote branch before pushing.

You can use the rebase operation to update your local branch with the latest commits from the remote. This will execute the following operations:

* Undo your local commits until a common parent is found
* Apply missing remote commits to your local branch
* Redo your local commits on the updated branch

git rebase origin/release/1.0

You can look at the commit log to see that the history of commits has been modified.

git log --all --oneline --graph

Now that Bob’s commit has been integrated into your local branch, you should be able to push Alice’s modifications to the remote repository.

git push origin

## Hotfix in production & Cherry-pick

Unfortunately, while the team was working to prepare the next release, a blocking bug has been found in production and needs to be fixed as soon as possible.

First, create a new branch from the master branch to work on the production code and fix the bug.

git checkout master

git branch hotfix/color

git checkout hotfix/color

Now that you are on the hotfix branch, you can apply the bugfix with the correct files.

Copy files from the 6.1 folder provided for the dojo.

Check the index.html web page to see the stylesheet change and commit your changes locally.

git commit -a -m "Update colors to use new theme"

Note the commit hash that was just generated, as it will be used in the next steps. You can now switch to the master branch and merge the hotfix to be deployed in production.

git checkout master

git merge hotfix/color

Look at the commit log to see how the hotfix has impacted the history.

git log --all --oneline --graph

Now that the bug has been fixed, you can go back to our release preparation. Of course, the hotfix deployed to production should be integrated into the next release.

A merge from the master branch into the release branch is not recommended, as it would be complex to merge, and the master branch may contain other hotfixes that should not be integrated into the release branch (i.e. deprecated features).

Switch to the release branch to prepare the integration of the hotfix.

git checkout release/1.0

The cherry-pick command will allow you to replicate a specific commit on a given branch, without doing a complete merge of the full history.

Use the following command to cherry-pick the hotfix commit into your release branch (replace the @commitId parameter with the commit hash from the previous step).

git cherry-pick @commitId

Finally, have a look at your commit log to ensure that our hotfix has been properly integrated in the release branch.

git log --all --oneline --graph

Your release is now ready to go in production !