GIT STEPS

Creating a repository in Github

1. Go to **github**.
2. Log in to your account.
3. Click the new **repository** button in the top-right.
4. Click the “**Create repository**” button.

**rickardljung/bootcamp\_course** repository is created

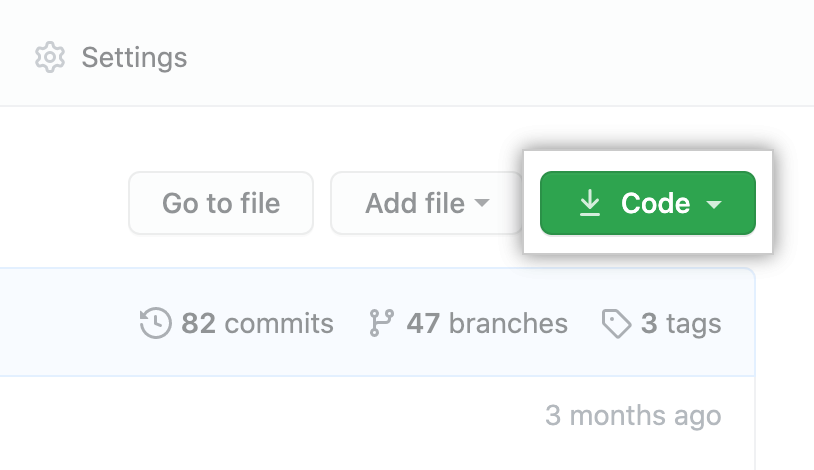
Invite the project members as Collaborators to the repository so as to add their contributions to the project. This can be done in Manage access

Rickard Ljung has invited Marcin, Jinsi and Ludwig as collaborators to   
**rickardljung/bootcamp\_course** repository

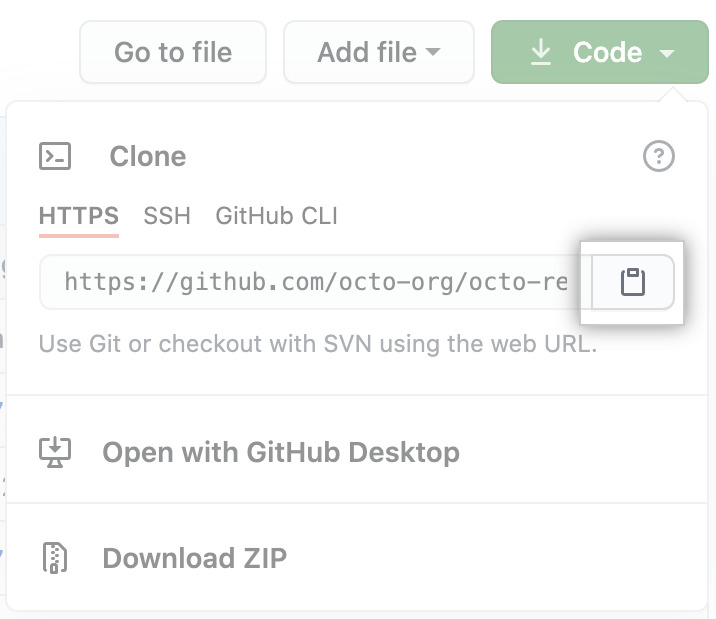
## **First submission**

### **[Cloning a repository using the command line](https://docs.github.com/en/free-pro-team@latest/github/creating-cloning-and-archiving-repositories/cloning-a-repository" \l "cloning-a-repository-using-the-command-line)**

1. On GitHub, navigate to the main page of the repository.
2. Above the list of files, click  **Code**.



1. To clone the repository using HTTPS, under "Clone with HTTPS", click . Copy the URL



1. Open Git Bash.
2. Change the current working directory to the location where you want the cloned directory.
3. Type git clone, and then paste the URL you copied earlier.

$ git clone [https://github.com/*YOUR-USERNAME*/*YOUR-REPOSITORY*](https://github.com/YOUR-USERNAME/YOUR-REPOSITORY)

Press **Enter** to create your local clone.

## Tracking changes with git add and git commit

### Edit a file in your local repo

Save your changes.

Once you are happy with your changes and have saved them, go back to your terminal window and type git status and hit return to execute the command.

* git add: takes a modified file in your working directory and places the modified version in a staging area.
* git commit takes everything from the staging area and makes a permanent snapshot of the current state of your repository that is associated with a unique identifier. The git commit command requires a **commit message** that describes the snapshot / changes that you made in that commit.

A commit message should outline what changed and why. These messages

1. help collaborators and your future self understand what was changed and why
2. allow you and your collaborators to find (and undo if necessary) changes that were previously made.

If you are not committing a lot of changes, you can create a short one line commit message using the -m flag:

* Alternatively, if you are committing many changes, or a small number of changes that require explanation, you’ll want to write a [detailed multi-line commit message](http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html) using a text editor.
* Here’s a model Git commit message:
* Capitalized, short (50 chars or less) summary
* More detailed explanatory text, if necessary. Wrap it to about 72
* characters or so. In some contexts, the first line is treated as the
* subject of an email and the rest of the text as the body. The blank
* line separating the summary from the body is critical (unless you omit
* the body entirely); tools like rebase can get confused if you run the
* two together.
* Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
* or "Fixes bug." This convention matches up with commit messages generated
* by commands like git merge and git revert.
* Further paragraphs come after blank lines.
* - Bullet points are okay, too
* - Typically a hyphen or asterisk is used for the bullet, followed by a
* single space, with blank lines in between, but conventions vary here
* - Use a hanging indent

After you’ve made a few commits, check out the output of the git log command. You should see the history of your repository, including all of the commit messages!

### Push changes to GitHub

So far we have only modified our local copy of the repository. To add the changes to your git repo files on your computer to the version of your repository on GitHub, you need to **push** them GitHub.

You can push your changes to GitHub with:

git push

The way **git**, and **GitHub**, manage this timeline — especially when more than one person is working in the **project** and making changes — is by using **branches**. A **branch** is essentially is a unique set of code changes with a unique name. Each **repository** can have one or more **branches**.

## **Default branch**

The default branch is considered the “base” branch in your repository, against which all pull requests and code commits are automatically made, unless you specify a different branch.

 The default branch name in Git is master. As you start making commits, you’re given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

In **rickardljung/bootcamp\_course** repository, “main” is created as the default branch

There is another branch “master” created for intermediate milestone merges

How does Git know what branch you’re currently on? It keeps a special pointer called HEAD.

In Git, this is a pointer to the local branch you’re currently on.

You can easily see this by running a simple git log command that shows you where the branch pointers are pointing. This option is called --decorate.

$ git log --oneline --decorate

### Switching Branches

To switch to an existing branch, you run the git checkout command.

$ git checkout master

This moves HEAD to point to the master branch.

To show commit history for the desired branch you have to explicitly specify it: git log master or git log main. To show all of the branches, add --all to your git log command.

**Switching branches changes files in your working directory**

It’s important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch. If Git cannot do it cleanly, it will not let you switch at all.

You can also see this easily with the git log command. If you run git log --oneline --decorate --graph --all it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

**Creating a new branch and switching to it at the same time**

It’s typical to create a new branch and want to switch to that new branch at the same time — this can be done in one operation with git checkout -b <newbranchname>.

From Git version 2.23 onwards you can use git switch instead of git checkout to:

* Switch to an existing branch: git switch testing-branch.
* Create a new branch and switch to it: git switch -c new-branch. The -c flag stands for create, you can also use the full flag: --create.
* Return to your previously checked out branch: git switch -.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you’re checking out, Git won’t let you switch branches. It’s best to have a clean working state when you switch branches.

# Git Branching - Basic Branching and Merging

# To create multiple branches for the main

$ git branch inputhandler

# To switch to the branch

$ git checkout inputhandler

# Merging

# You can run your tests, make sure the hotfix is what you want, and finally merge the inputhandler branch back into your main branch to deploy to production. You do this with the git merge command

$ git merge inputhandler

# Delete a branch

# You can delete a branch with the -d option to git branch: Since the inputhandler branch is merged to the main, after testing, it can be deleted.

$ git branch -d inputhandler

# Final Merge to Master

# To merge main to master

# Check out from main to master

$ git checkout master

# git merge master