



GitHub

Usage manual GitHub
through Linux Terminals
ROS-autom 2017[®]

GITHUB

SYSTEM PREPARATION

We make sure that we have the necessary packets

```
sudo apt-get install git  
sudo apt-get install repo
```

Το πιο πιθανο ειναι να μην υπαρχει μονο το repo και να απαιτει εγκατασταση. Afterwards, a connection must be made between the github client of our PC and the website's github, so as for us to be able to make commits, pulls, pushes etc.

We run:

```
git config --global user.email dkatkaridis@gmail.com((double dash))  
git config --global user.name mygithubname
```

Mail and password must perfectly match with what we see on the website.

My mail is dkatkaridis@gmail.com with username dkati.

Then, we must certify on github, which PC will be making Push, through our account.

```
ssh-keygen -t rsa -b 4096 -C
```

Received feedback should look like this

```
Generating public/private rsa key pair.  
Enter file in which to save the key  
παταμε enter  
Enter passphrase  
παλι enter  
Enter passphrase again  
Παλι enter  
Τελικα θα μα δωσει το RSA 4096
```

```
The key's randomart image is:  
+---[RSA 4096]---+  
|  +0o+o . =0+++ |  
|  ...00.=.+.+ . |  
|  oo+*o. o . |  
|  o.+=. o |  
|  .+. oS. |  
|  . ....E+o. |  
|  o ... |  
|  . . . |  
|  ....O.. |  
+---[SHA256]---+
```

This sshkey must be written on ssh-agent.

```
eval "$(ssh-agent -s)"
```

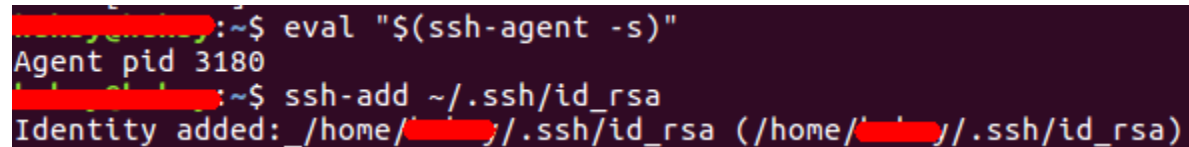
The output must be of this format : **AgentpidXXXXX**

Save the ssh in a local file

```
ssh-add ~/.ssh/id_rsa
```

with output:

Identity added : /home/username/.ssh/id_rsa (home/username/.ssh/id_rsa)

A terminal window with a dark background and light-colored text. The first line shows a prompt followed by the command 'eval "\$(ssh-agent -s)"'. The output is 'Agent pid 3180'. The second line shows a prompt followed by the command 'ssh-add ~/.ssh/id_rsa'. The output is 'Identity added: /home/username/.ssh/id_rsa (/home/username/.ssh/id_rsa)'.

```
username:~$ eval "$(ssh-agent -s)"
Agent pid 3180
username:~$ ssh-add ~/.ssh/id_rsa
Identity added: /home/username/.ssh/id_rsa (/home/username/.ssh/id_rsa)
```

Later on, we must copy the SSH to clipboard. For this purpose, we download the "xclip" tool and we copy the SSH, using xclip, to clipboard.

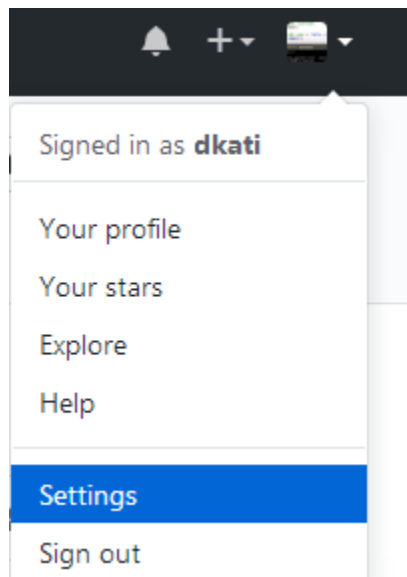
```
sudo apt-get install xclip && xclip -sel clip < ~/.ssh/id_rsa.pub
```

```

ubuntu@ubuntu:~$ sudo apt-get install xclip
[sudo] password for ubuntu:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  xclip
0 upgraded, 1 newly installed, 0 to remove and 3 not upgraded.
Need to get 17,0 kB of archives.
After this operation, 72,7 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu xenial/universe amd64 xclip amd64 0.12+svn84-4 [17,0 kB]
Fetched 17,0 kB in 0s (54,0 kB/s)
Selecting previously unselected package xclip.
(Reading database ... 346421 files and directories currently installed.)
Preparing to unpack .../xclip_0.12+svn84-4_amd64.deb ...
Unpacking xclip (0.12+svn84-4) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up xclip (0.12+svn84-4) ...
ubuntu@ubuntu:~$ xclip -sel clip < ~/.ssh/id_rsa.pub
ubuntu@ubuntu:~$

```

Having done the above, this ssh must be added to our github account. We open our account options:



From the menus, we chose the option SSH and GPG keys and then press the button on the right, that says "New SSH key".



Chose a title for the Title field, and for the Key field we just need to right click and paste clipboard's content from previous step.

Personal settings
Profile
Account
Emails
Notifications
Billing
SSH and GPG keys
Security
Blocked users
Repositories
Organizations
Saved replies
Authorized OAuth Apps

SSH keys

2  [New SSH key](#)

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

	Fingerprint: [redacted]	Delete
SSH	Added on May 10, 2017 by [redacted] Last used within the last 2 weeks	
	Fingerprint: [redacted]	Delete
SSH	Added on Jun 6, 2017 Last used within the last 3 days	

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#).

GPG keys

[New GPG key](#)

There are no GPG keys with access to your account.

Final result should look like this:

Title

Key

```
/PHFDF63YNq1ENBQNwezM5B2xMBPZnZueqI1GMFUkmDcqyRjWADCI0sOJ
/uQd4UMsLZkeO3gSYpxQxRmEMtvJVy45tZnot+EXqB2
/j6WnD3Mv3OyWoK53bj8hIH5A0Ny34qCffHEYsE65cDdaNP9nR6pxeAeKG2sH8QUul+E/efyOYT
/rQEB7zludNyPvCXsogdMg7on7ARjTTMnd9vkXCQZ2QQRaEjYr0LtrZRf3AA6eJD+0fM4nj4j8KNehWziSA
MKgJQV2fk437fGv9qK6+J6r1VyrhKHRmnDbNJHkgybE8chj3OCswGM6fQN3jeMYtrmKEGl3gDk2S4ztOW
jp6Onn2kMeOqurt+y1PIVdCYefqw6lFRoVIEjmbP4AOhK
/JRgmF0pxnQWUFsfzJWjKlegQWHDH0OFYC8+s4UGgHiLVNsKIF2JstOtVvty7UQ==
dkatkaridis@gmail.com
```

Add SSH key

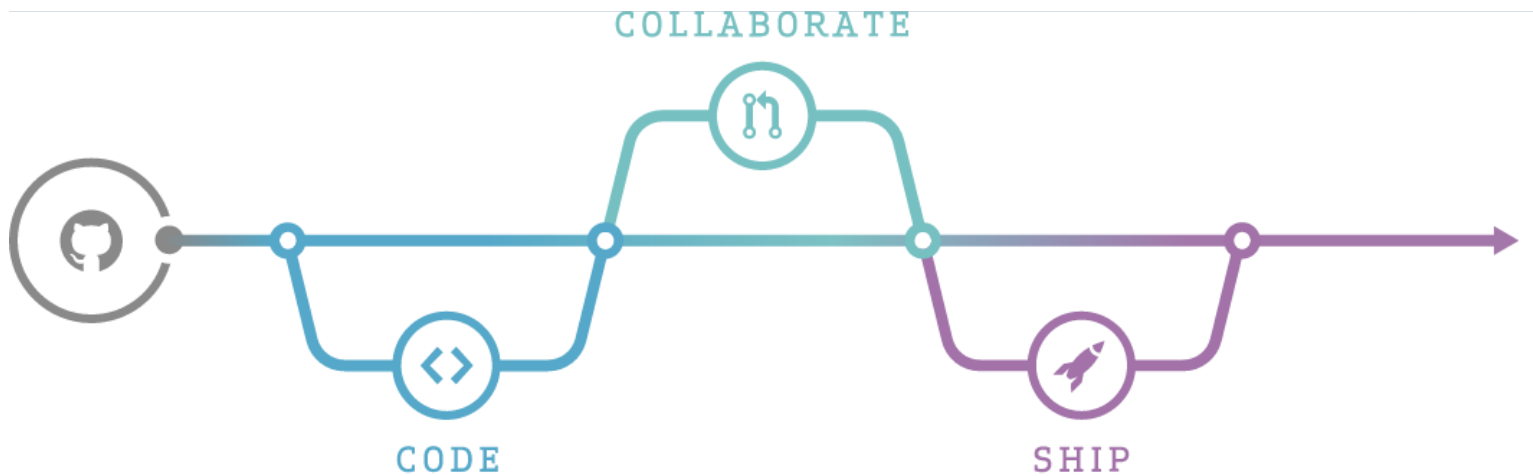
We press AddSSHkey.

There is a chance of being asked to give an access password. In such case, we simply type in our password.

Our computer is now remotely connected with our account on github.

GITHUB

HOW IT WORKS AND PHILOSOPHY



We can imagine github's operation as a tree (with cherries – we will come back to this later) with branches having its own core, and all these based on a timeline. Let's watch the figure below to explain the flowchart of a random application.



What we notice in the figure above are the following:

- 1 field named master
- 1 field named hotfix
- 1 field release
- 1 field develop
- 2 field feature
- Some circles
- 6 dashed lines
- Flow arrows
- 3 small boxes mentioning the application's version

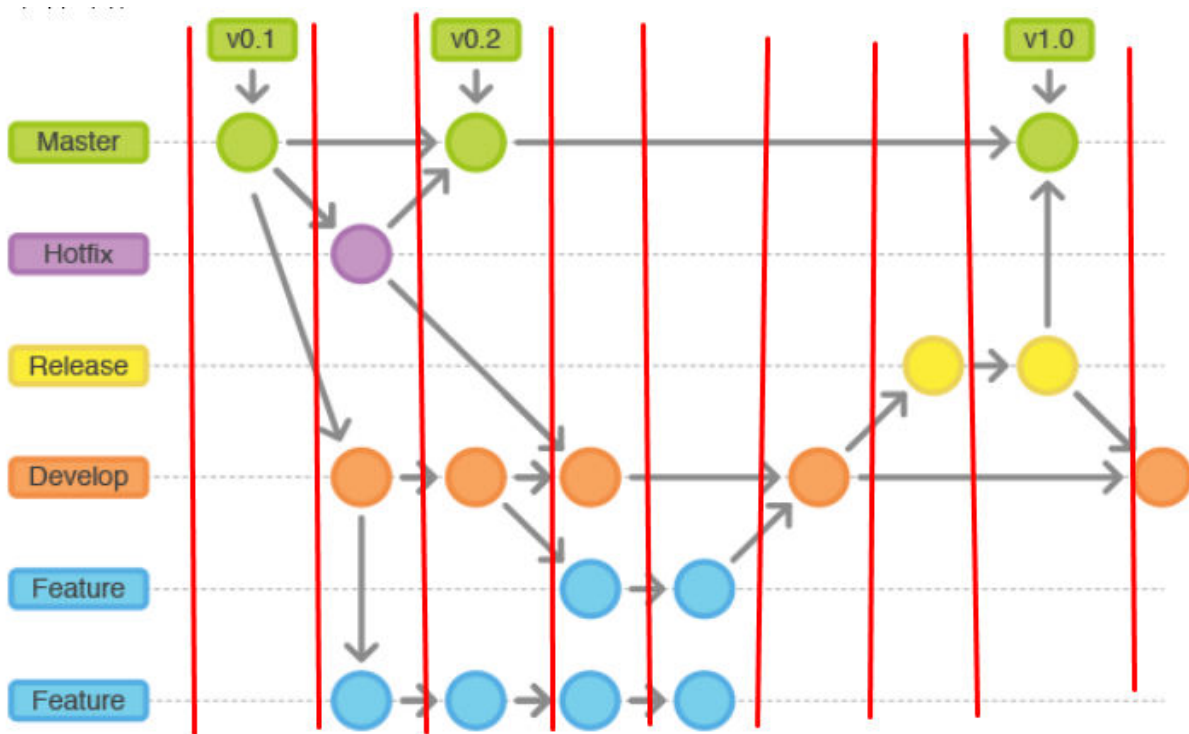
Master is the main frame. This is the first branch to be created and it is the program's main flow. The rest of the frames are the secondary branches.

The horizontal dashed lines show the application's progress over time, directed from left to right.

Flow arrows show the steps that github's flow makes, the structure of which will be analyzed thoroughly in the next further on below.

Circles represent a change (also known as commit) in the code.

It's very important that we notice the vertical alignment of the abovementioned circles



Let's break the last into pieces.

In the first vertical section, there is only the circle of Masterbranch. This states that this is the first initial release of our source code, on github. We might as well imagine Masterbranch as the "formal" source code we would wish someone to see. Usually, this is the stable version of current code.

After that, we see three arrows heading to

1. Hotfix
2. Develop
3. Feature.

This means that we have created three branches from the master. Obvious as it is, these circles are not part of the same vertical section with the master. That's because of previous commit – some change in the code. As a result, we see that the programmer of this code has created three more branches, enabling himself to add a change.

-And what's the reason why he doesn't add that change straight into the masterbranch?

-So as he can check it. If it works fine, then he can add it to the masterbranch (we will see how he does that, below).

In the third column, hotfix gets into master, meaning that hotfix was possibly a bugfix and, so, the programmer adjusted it into the main branch. Because of this change, the programmer also changed the program's version to vo.2.

In the same column (third), we should also keep an eye on the other circles of develop and feature.

Every circle is a commit (a change) in the code. This means that if the circle, located in a column, also exists in another branch, then the very same part of code exists in that branch.

The whole figure should be read using this logic. Arrows with a diagonal direction state the creation of a new branch from that commit. This might be confusing but we will analyze all these later on.

Remarks

- Commits are saved to Github's history. If something is written in history, it cannot be deleted.
- Github's history can be reset or reverted.
 - Revert: rolls the code back to a previous state
e.g. If a variable's declaration is deleted, then reverting the code will bring the declaration back.
- The whole of source code, together with all branches, is called repository (repo).

- Pushing a commit does not mean that changing branch is necessary. The reason why I change branch is because I want to make tests on my code without affecting the basic code which is taken for stable
- For every repository, we can add contributors and grant them privileges to change the code. So, next time I want to make a commit, I will have to first “pull” other contributors’ changes.

GITHUB

LINUX TERMINAL AND COMMANDS

Upon Github’s creation, the only way to use it was through Linux terminals. Later, there was a Desktop application for Windows created. Here we are going to make a brief Linux terminals “how to”.

One getting familiar with Github through terminals, will help himself learn how to use the Windows Desktop application faster. **But not vice versa.**

(Windows application also provides terminals to use, but it’s aimed for advanced users)

There are two ways to upload/download my code to/from Github:

- Upload/Download code to a specific branch, in my Github, to edit. A disadvantage of this method is that sometimes it is necessary to download the whole code from Github instead of the last commits, that another contributor has made to the project.
- Manifest creation that has pre-downloaded all branches to a .tmp folder, thus making them always available without any chance of getting deleted. The only drawback of this method is its complexity.



Below, we are going to refer to both ways but thoroughly analyze the second one, which we consider to be safer. Nevertheless, the first method makes it easier for someone to get whole repositories from others.

STARTING A NEW REPOSITORY TO UPLOAD CODE OF ONE'S OWN PROJECT

We move to our profile on Github.com. After having successfully logged in, chose **+** from top right and then chose New repository to add information of the initial repository. Upon creation, it will be empty and later we are going to add our code as an initial release. Insert the repository name, a brief description (optional), chose the Public option and tick the “Initialize this repository with a README” field so as our first file to be created. License file will be added later.

Owner

Repository name

 **dkati** ▾ / 

Great repository names are short and memorable. Need inspiration? How about **laughing-bassoon**.

Description (optional)

This repo is made for tutorial purposes



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.



Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

Add a license: **None** ▾



Create repository

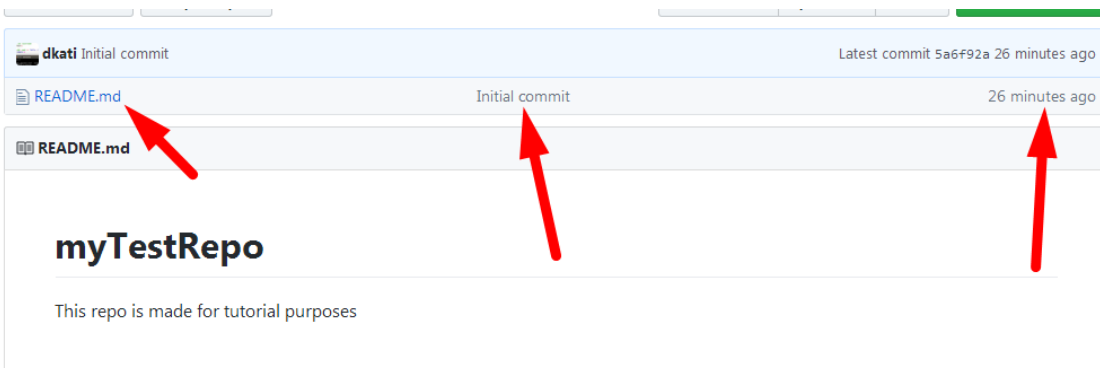
Clicking the “Create repository” button should bring you to the following menu, as shown in the figure below:

The screenshot shows the GitHub interface for a repository named 'myTestRepo' by user 'dkati'. At the top, the repository name and owner are displayed. To the right, there are buttons for 'Unwatch' (with a dropdown arrow), '1' star, '0' star, and '0' fork. Below this is a navigation bar with tabs for 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Settings', and 'Insights'. A message states 'This repo is made for tutorial purposes' with an 'Add topics' link and an 'Edit' button. Below the message, statistics show '1 commit', '1 branch', '0 releases', and '1 contributor'. A row of buttons includes 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The commit history shows 'Initial commit' by 'dkati' with the latest commit '5a6f92a' 'just now'. The file list shows 'README.md' as the 'Initial commit' 'just now'. The main content area displays the 'README.md' file, which contains the text 'myTestRepo' and 'This repo is made for tutorial purposes'.

There are many things to notice here, so let's take them one-by-one:

- Username and repository name are both shown on the top left of our screen (dkati / myTestRepo)
- Beneath that, there is a menu consisting of many tabs:
 - Code : The main and most important menu
 - Issues : In this tab, it's possible for any user (even if one is not a contributor) to report issues for contributors to see, answer and solve them
 - Pull requests : In this tab there are 3rd party-user made suggestions regarding the code, all gathered, in order for contributors to see the changes. Then, contributors can allow the authors, of the suggestions, to implement their changes, just by pressing one button, pushing the commit into the repository. As already mentioned, this process allows someone who is not part of the contributors' team of a repository, to push their commits to a specific repository, as long as contributors approve them.
 - Wiki : Traditional, let's say, wiki where contributors provide instructions and information regarding the source code.
 - Settings : Repository settings (nothing to do with the source code)
- Next, there is the description which was given during the repository creating process.

- Moving on, there's another menu of the following tabs:
 - 1 Commit: Pressing on this menu, we are shown every commit made, sorted by date of creation. Details about commits will be provided later.
 - 1 branch: Shows branches.
 - 0 releases: This is about releases made by contributors.
 - 1 contributor: Contains information about every person who has contributed to the code development, whether they are members of the repository or have used its code.
- Next, we find the branch switching button. Clicking on it, enables the user to navigate through branches and see the code, as well as the commits of it, contained on the branch that the user is currently switched to.
- New pull request: Button through which a 3rd party user can add code.
- Create new file / Upload files : Manual file creation / upload (not recommended) (**Not recommended**)
- Clone / Download : Source code download under .zip format (Not recommended). Reason why this is not preferable is because in Linux environments, symlinks can be deleted during the extraction of the .zip file.
- **dkati** Initial commit : This option refers to the last commit to be made. Github, automatically, created a commit when we create a repository and add a "readme" file. Syntax of the commit title is as follows: <github username><commit title>.
- Directory's files and their information are shown in the menu below the one that we described above.



Left arrow points to the files contained in the specific directory.

Middle arrow points at the last commit to be made and has , also, affected the file which the left arrow is pointing at.

Right arrow shows the time elapsed since the last commit occurred.

Such information makes it easier to keep large sized source codes organized.

Given the fact that we have comprehended the structure of our repository, let's now create **one more repository and name it "myTestRepo2"**, which will be our 2nd tool. Afterwards, we are going to explain and also create the manifest that links all these together.

What we call Manifest is a special repository that contains one or more xml files, which include Github links from projects that we need in order to set up a whole source code.

It is noticeable that many source codes from professional programs are separated into many parts, every each of which is a tool. For instance, there are many tools in ROS (Gazebo, ROScore, RVIZ) which we want to have then working together.

So, we are going to make, at least, 2 repositories for each tool we wish to make our own / custom changes.

The purpose of the manifest, is to define to git service, which sources must be downloaded.

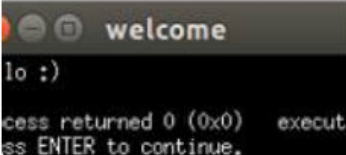
So instead of downloading one-by-one every project of ours, we place them in one directory and split it into parts, through the manifest, for keeping it better organized.

CREATING A MANIFEST

Make a new public repository by the name "myproject-manifest" with a readme file. Before moving on, let's take a look on the main menu of our profile.

```
include <iostream>
main()

std::cout << "Hello :)
return 0;
```



Dimitris Katikaridis
dkati

[Add a bio](#)

✉ dkatikaridis@gmail.com

Overview Repositories **41** Stars **0** Followers **15** Following **6**

Search repositories...

Type: **All** ▾

Language: **All** ▾

 **New**

myproject-manifest

Updated 13 seconds ago

myTestRepo2

Updated 6 minutes ago

myTestRepo

This repo is made for tutorial purposes

Updated an hour ago

What we first notice is that we now have three repositories. One is myTestRepo which might be one of my source code tools. Second if myTestRepo2 which is a 2nd tool of mine and the last one is the manifest.

Press on myproject-manifest and chose, the option on the right, "Create new file".
In the name chosing field, type «default.xml». Set, as content, the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>

<remote name="github"
fetch="https://github.com/" />

<project path="myTestRepoLocalDir" name="dkati/myTestRepo" remote="github" revision = "master" />
<project path="myTestRepo2LocalDir" name="dkati/myTestRepo2" remote="github" revision = "master" />

</manifest>
```

Let's explain the code.

First line is about XML format.

Remote is the xml member that defines where git service will pull all the sources from.

Path project's member is the local directory where the downloaded source code will be stored.

Name is the Github name of the repository.

Revision is the branch that will be shown in our local folder and the branch from/to we will be downloading/pushing.

Under the headline Commit new file, we chose the Commit title.

Here we type the commit title which will be appearing in the commits. Commit titles must be treated with respect since they are what a 3rd person will be able to see. In other words, it's considered wise to take care of the image we show to the "outsiders". A typical commit title is as follows:

<dir></subdir> : <Changes made>

E.g. If I have changed a class inside a file located at directory mysource/src/libs/mylib.cpp then the commit title should be something like this: src/lib: Do not expose _var from mylib

In general, what we try when it comes to commit titles, is to be brief and to the point.

If there's anything to describe, we can do it by placing our explanations in the commit description.

Having done the above, we press Commit new file.

After the file creation, open the readme.md and now we can start editing it, by choosing the pencil icon on the right. Edit will be the writing of the main command which downloads, and sets up, the manifest to our computer.


Command:



repo init -u git://github.com/dkati/myproject-manifest.git -b master


After doing that, we must syntax a commit title.

A proper title might be ->**readme:Add our repo init command**

Next, having finished with the commit, we go back to myproject-manifest. The edited readme.md file appears to be in the front “side” of the repository. **This only applies for readme files** and future file changes will not be shown like this.

 **dkati** committed on **GitHub** readme:Add our repo init command Latest commit 7eb7f47 an hour ago


 README.md	readme:Add our repo init command	an hour ago
 default.xml	Create default.xml	an hour ago

 **README.md**

myproject-manifest

Get our manifest

```
repo init -u git://github.com/dkati/myproject-manifest.git -b master
```



MANIFEST AND SOURCE CODES INITIALIZATION

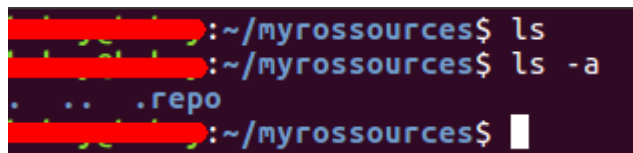
Once we have the repositories and the manifest ready, we can now start saving these repositories in our computer.

```
cd
```

```
mkdir myrossources && cd myrossources
```

Should we receive any message about account colors, we press 'Y' and move on. The created folder will appear to be empty until we use the following command to reveal hidden files:

```
ls -a
```



```
~/myrossources$ ls
~/myrossources$ ls -a
.  ..  .repo
~/myrossources$
```

In this folder, there is the symlink manifest.xml that points at `.repo/manifests/default.xml` which is the manifest we wrote before.

Folder contains all the necessary files for the git/reposervice to work but we are not going to mess with them.

So, we switch to the previous directory (`cd ..`) and run:

```
reposync
```

ATTENTION.

repo sync must always be executed in the same directory that folder .repo exists.

If, even by accident, repo sync command is executed in another directory, there will be a problem with all source code.

```

[redacted]~/myrossources$ ls
[redacted]~/myrossources$ ls -a
.  ..  .repo
[redacted]~/myrossources$ cd .repo
[redacted]~/myrossources/.repo$ ls
manifests manifests.git manifest.xml repo
[redacted]~/myrossources/.repo$ cd ..
[redacted]~/myrossources$ repo sync

... A new repo command ( 1.23) is available.
... You should upgrade soon:

cp /home/[redacted]/myrossources/.repo/repo/repo /usr/bin/repo

Fetching project dkati/myTestRepo
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %         %         Dload  Upload   Total   Spent    Left   Speed
0         0     0     0     0     0      0      0  --:--:--  --:--:--  --:--:--    0
curl: (22) The requested URL returned error: 404 Not Found
Server does not provide clone.bundle; ignoring.
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
From https://github.com/dkati/myTestRepo
* [new branch]      master    -> github/master
Fetching projects: 50% (1/2) Fetching project dkati/myTestRepo2
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %         %         Dload  Upload   Total   Spent    Left   Speed
0         0     0     0     0     0      0      0  --:--:--  --:--:--  --:--:--    0
curl: (22) The requested URL returned error: 404 Not Found
Server does not provide clone.bundle; ignoring.
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
From https://github.com/dkati/myTestRepo2
* [new branch]      master    -> github/master
Fetching projects: 100% (2/2), done.

[redacted]~/myrossources$

```

After the execution of the previous command, we have now downloaded all the source code to our computer.

We can also download only one repository using the following command:

Repo sync myTestRepoLocalDir

This will refresh the local source code and replace it with the source code located on Github. Doing so, helps in cases when we want to discard changes we have made.

It is sensible to always keep our sources on Github updated, in order to be able to safely make local changes. So if there is anything wrong, we can just erase the folder we're working on and then execute the repo sync command for the repository we wish.

Repo sync command can receive as arguments the number of threads to use while it's syncing.

reposync -j4

But why do we care about the that number of threads since everything is done using the internet?

We do care, because repo downloads the source code which is compressed, extracts it and through a process called “diff check”, it checks which files have been locally changed and replaces them. So, if this process takes long to complete, increasing the `-j` can result in faster performance during repo.

It is worth mentioning that `-j` also affects the network load. So, practically, the higher the number of threads, the higher the download speed, increasing the chances of an error to occur. This differs from network to network.

What has been so far described, needs to be done only once. What follows is what one must need to know in order to get familiar with github and the way it works.

FIRST COMMIT AND FIRST PUSH

The most important part is to understand what we want to push and when we should do it. As already mentioned, a commit must be clean. This means that each commit:

- Must have a clear title as to what it describes and what it does
- If need be, it should provide a description with details
- It should not contain file changes that are irrelevant to the main commit change and its purpose
- Να μην είναι αντιγραφή από άλλο commit. Αν θελω το commit καποιου τριτου θα το κανω με τον νομιμο τροπο που θα δουμε παρακατω

Let's explain the 3rd bullet.

When one is getting familiar with Github, faulty pushes are very much likely to be made.

For example, assume that we wish to push a change made to a folder, which contains three files, for a specific purpose. The wise thing to do would be to not change any other file that does not have anything to do with the main editing.

Example

In a file, 2 variables are changed from int to double while in another file, I change the name of a class.

What I type for commit title is "src:mylib Switch 2 vars from int to double". Nevertheless, the class name change is also one change, and it will be pushed too. This is something that we would not like to happen.

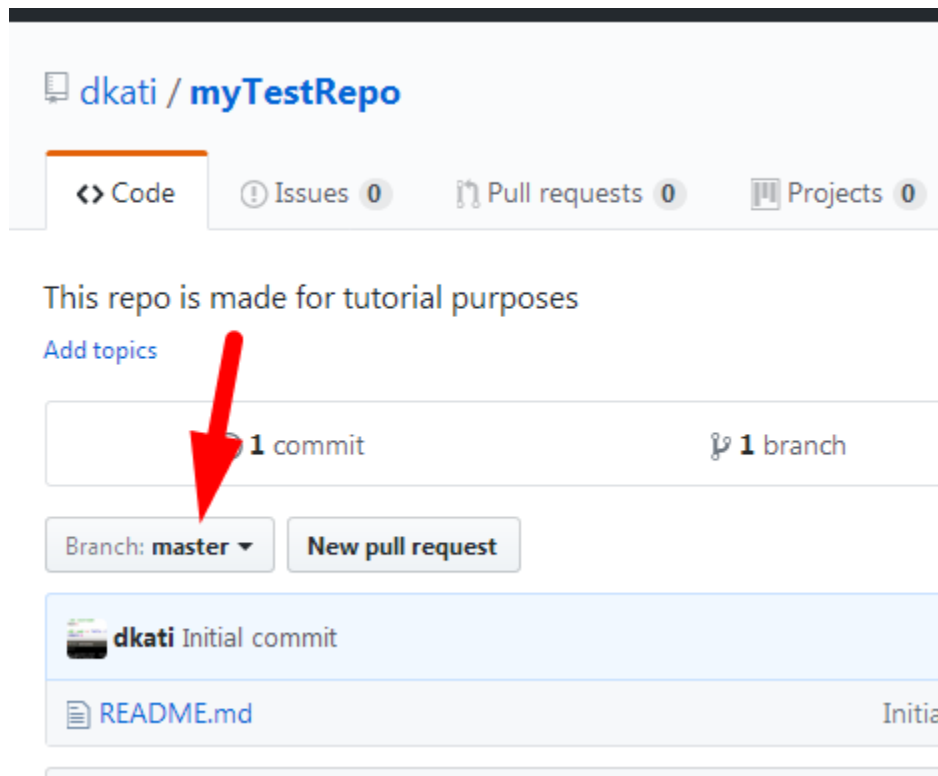
So let's go and make our 1st change which, for the sake of our example, will be made on myTestRepo. The local folder of this repository, is myTestRepoLocalDir. So....

```
cd myTestRepoLocalDir
```

Now that we are in the directory, we must state which branch we are working on. First we check if, by coincidence, we are already on a branch. If this is the case, it can be the result of a past process of ours in this folder. We execute:

```
git branch
```

And the message we should get is (nonbranch), meaning that we are not on any branch right now. We can find the name of our branch from our repository page on Github.



If I intend to send any changes to this branch, I must first execute the following commands:

```
git branch master
```

```
git checkout master
```

or simply

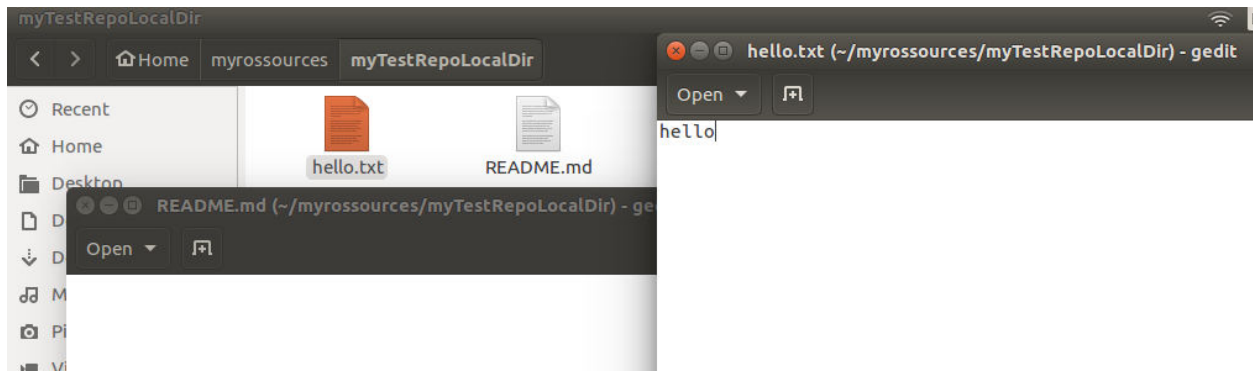
```
git checkout -b master
```

```
myrossources/myTestRepoLocalDir$ git branch
* (no branch)
myrossources/myTestRepoLocalDir$ git branch master
myrossources/myTestRepoLocalDir$ git checkout master
Switched to branch 'master'
myrossources/myTestRepoLocalDir$
```

Once I have changed my branch, I can either keep the terminal open or close it. There are no things running in the background.

I make my changes, locally.

We will add a .txt file, named hello.txt having for content a simple word "Hello" and then we will delete whatever is inside the readme file.



Assume that this is the change we want to make. Last thing to do is to add this to the commit.

git add -A
git commit

Once the above are done, a nano window should automatically open in our terminal, on which we are editing our commit.

First line is always about the commit title. Description is added in the third line (one empty line separates title from description field)

```
My first commit!
thats my first commit!YaY!
# Please enter the commit message for your
# with '#' will be ignored, and an empty me
# On branch master
# Changes to be committed:
#   modified:   README.md
#   new file:   hello.txt
#
```

Notice the following two lines

modified: README.md
new file: hello.txt

By watching these lines, we can validate that we didn't affect any file that should not be part of our commit. If everything is correct, we can move on.

Otherwise, we stop the editing process, without completing our commit title and description, by pressing `ctrl+x` and then "Y". We head to the folder where `.repo` is located (in other words, `myrossources` root) and either we delete the folder, followed by a `repo sync`, or we push our commit and make a fix commit afterwards (not recommended). Another method is to keep, in a different folder, the files we have edited, do a `repo sync` and then paste these edited files.

Press `ctrl+x` and then “Y” for the commit to close. Inside the terminal, we can see the commit title, the two changed files (one addition and one deletion). Additions and deletions refer to code lines. It also shows us that the `hello.txt` file was created, with mod `o644`.

Once the commit is done, we must push.

```
git push origin master
```

This command tries to send our commit to the branch master.

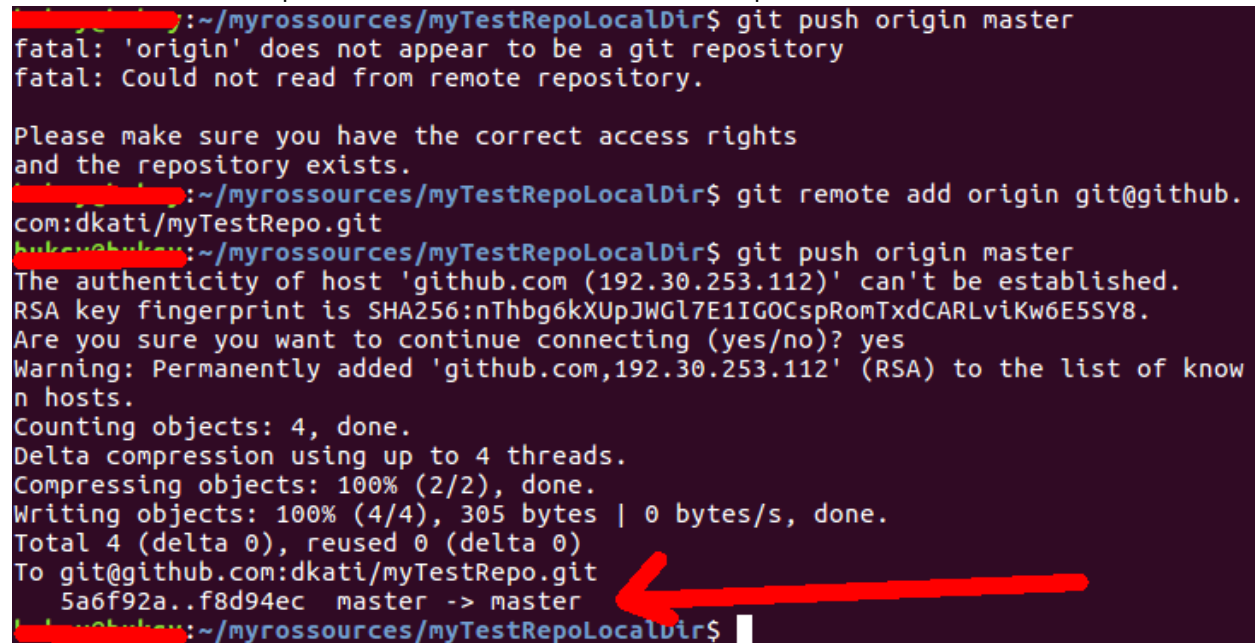
In our case, this try gets some fatal errors. The reason why we get these errors is that, first, we need to create the remote (only needed during the 1st push). The remote is responsible for sending the local commit to our profile.

```
git remote add origin git@github.com:dkati/myTestRepo.git
```

And then, again:

```
git push origin master
```

We will be asked if we really want the push to be done (this question will not be repeated). Press “yes” and it should start the process. The result should look like the picture below.



```
huker@huker:~/myrossources/myTestRepoLocalDir$ git push origin master
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
huker@huker:~/myrossources/myTestRepoLocalDir$ git remote add origin git@github.com:dkati/myTestRepo.git
huker@huker:~/myrossources/myTestRepoLocalDir$ git push origin master
The authenticity of host 'github.com (192.30.253.112)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGL7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,192.30.253.112' (RSA) to the list of known hosts.
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 305 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To git@github.com:dkati/myTestRepo.git
  5a6f92a..f8d94ec master -> master
```

The change I have made can be found in the project’s commits.

https://github.com/dkati/myTestRepo/commits/master

Αναζήτηση

This repository Search Pull requests Issues Marketplace Gist

dkati / myTestRepo Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Settings Insights

Branch: master

Commits on Jun 9, 2017

My first commit! f8d94ec
dkati committed 18 minutes ago

Initial commit 5a6f92a
dkati committed 5 hours ago

Clicking on the commit will show relevant information.

https://github.com/dkati/myTestRepo/commit/f8d94ec598e8e94ccfb5c61ca0c5c973ca61cc1e

Αναζήτηση

This repository Search Pull requests Issues Marketplace Gist

dkati / myTestRepo Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Settings Insights

My first commit! Browse files
thats my first commit!YaY!

master

dkati committed 19 minutes ago 1 parent 5a6f92a commit f8d94ec598e8e94ccfb5c61ca0c5c973ca61cc1e

Showing 2 changed files with 1 addition and 2 deletions. Unified Split

2 README.md View

```
... @@ -1,2 +0,0 @@  
1 -# myTestRepo  
2 -This repo is made for tutorial purposes
```

1 hello.txt View

```
... @@ -0,0 +1 @@  
1 +hello
```

0 comments on commit f8d94ec Lock conversation

Such information is the title, our changes and the given description.

Red lines show what is deleted from the file, and green ones show additions. If every line of a file is marked as green, this means that this file was created during the commit.

COMMIT INVERSION

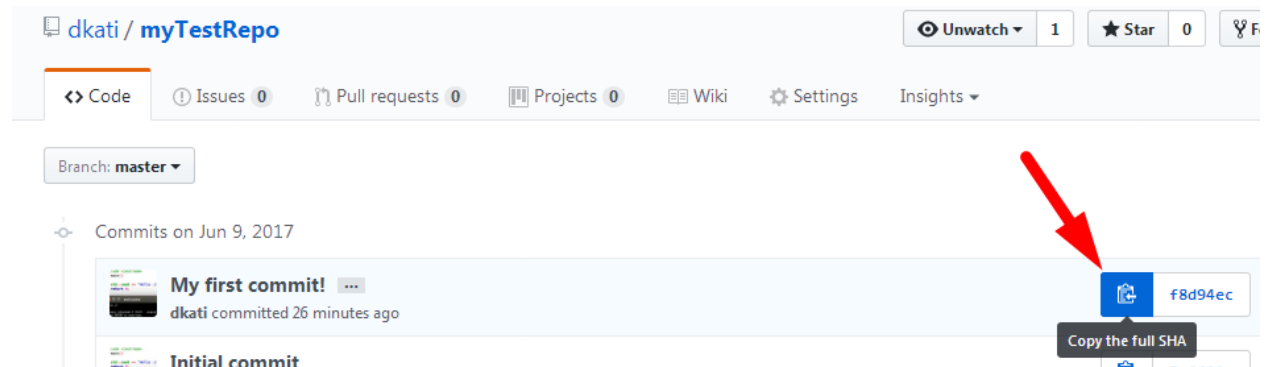
We have already mentioned we cannot delete commits on Github. Its history keeps whatever we push. Nevertheless, we can revert a commit using only one command.

git revert SHA_CODE

SHA is each commit unique key and it can be found either in a commit itself or in the history.



So, either we collect it by copy-paste from there, or by clicking on the corresponding icon as shown below:



7fb76ff5c019adfo4bf2369583fbcee346a1c4dd

This automatically saves the commit SHA, on clipboard.

git revert 7fb76ff5c019adfo4bf2369583fbcee346a1c4dd

Reverting, the commit message window is shown again. Usually, commit title is left untouched and we only change the description explaining the reason of revert.

After completing our changes, exit using ctrl+x and "y".

```
Revert "My first commit!"

i believe its not needed.breaks the build
This reverts commit f8d94ec598e8e94ccfb5c61ca0c5c973ca61cc1e.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   modified:   README.md
#   deleted:    hello.txt
#
```

And then push

`git push origin master`

After pushing, commit is shown in the commit history

dkati / myTestRepo

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Settings Insights

Branch: master

Commits on Jun 9, 2017

	Revert "My first commit!" ...		7fb76ff	
dkati committed 4 minutes ago				
	My first commit! ...		f8d94ec	
dkati committed 32 minutes ago				
	Initial commit		5a6f92a	
dkati committed 5 hours ago				

CHERRY-PICKING. THE «LEGAL» CODE COPY

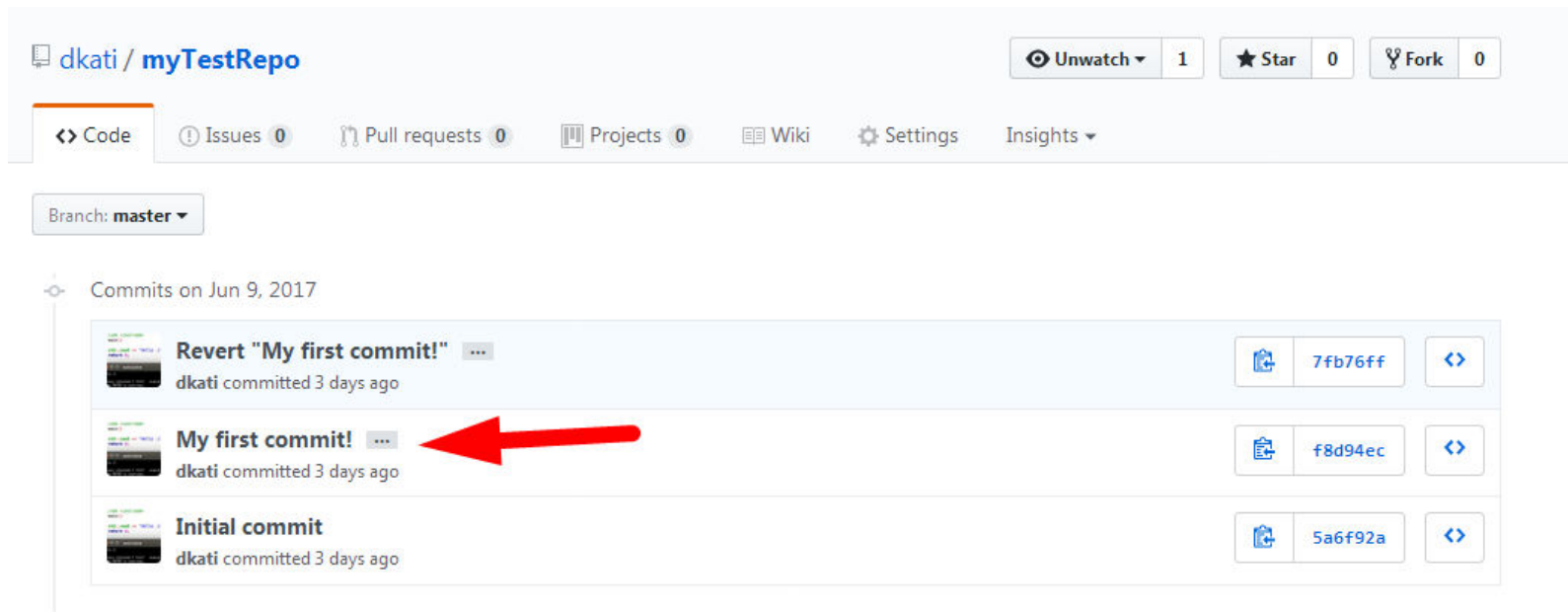
Every code on Github can be secured by an international license (apache, MIT etc.).

Licenses protect authors from possible code “theft”. Github secures legality through its mechanics and provides users with functions that enable them to copy commits made by other people, as long as necessary credits are given.

Cherry-pick function helps us to update parts of other programmers' code, as long as the proper credits are provided, like mentioned above.

In order to cherry-pick, we must first download the whole source code, with the commits, from which we wish to acquire specific commits.

Assume that we wish to cherry-pick the "my first commit" commit we made, previously, from <https://github.com/dkati/myTestRepo>.



After picking the commit we need, we must, using the terminal, head to the project's directory to which we want to place the commit. For example, if the directory we chose to place the commit is "myTestRepo", the command should look like this:

```
git fetch https://github.com/dkati/myTestRepo master
```

Command's basic syntax is:

```
git fetch <link of project> <branch name>
```

We have just stored the specific repository's commit history.

```
~/myrossources/myTestRepo2LocalDir$ git fetch https://github.com/dkati/mytestrepo master
warning: no common commits
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 8 (delta 0), reused 7 (delta 0), pack-reused 0
Unpacking objects: 100% (8/8), done.
From https://github.com/dkati/mytestrepo
* branch      master      -> FETCH_HEAD
~/myrossources/myTestRepo2LocalDir$
```

We are now ready to “pull” the commit

```
git cherry-pick f8d94ec598e8e94ccfb5c61ca0c5c973ca61cc1e
```

Command’s syntax is:

```
git cherry-pick <SHA code>
```

Running this command, there is an error shown mentioning that the commit cannot be adjusted to our code.

When cherry-picking, such errors are very likely to happen and one must be able to fix them. Command result appears in the picture:

```
~/myrossources/myTestRepo2LocalDir$ git cherry-pick f8d94ec598e8e94ccfb5c61ca0c5c973ca61cc1e
error: could not apply f8d94ec... My first commit!
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
Recorded preimage for 'README.md'
~/myrossources/myTestRepo2LocalDir$
```

Looking at the commit, we notice that two files are changed.

The screenshot shows the GitHub interface for a repository named 'dkati / myTestRepo'. The commit being viewed is titled 'My first commit!' with the message 'thats my first commit!YaY!'. It was committed 19 minutes ago by user 'dkati'. The commit details show 1 parent (5a6f92a) and the commit hash f8d94ec598e8e94ccfb5c61ca0c5c973ca61cc1e. Below the commit message, it states 'Showing 2 changed files with 1 addition and 2 deletions.' The files listed are README.md (2 deletions, 0 additions) and hello.txt (1 addition, 0 deletions). The README.md diff shows lines 1 and 2 being deleted, which were '# myTestRepo' and '-This repo is made for tutorial purposes'. The hello.txt diff shows line 1 being added, which is '+hello'. At the bottom, it says '0 comments on commit f8d94ec' and has a 'Lock conversation' button.

Nevertheless, terminal showed:

Recorded preimage for README.md (*)**. This means that the error exists only in this file.

We open the file and see this:

The screenshot shows a terminal window titled 'README.md (~/.myrossources/myTestRepo2LocalDir) - gedit'. The terminal output is as follows:
<<<<<<< HEAD
myTestRepo2
=====
>>>>>> f8d94ec... My first commit!

Let's analyze the syntax of the following text

```
<<<<<<<< HEAD
```

```
# myTestRepo2
```

```
=====
```

This shows us that the original file contains whatever is included between

```
<<<<<<< and =====
```

Commit change is included in the field starting from ===== until >>>>.

In our case, content between ===== and >>>> is empty, meaning that the commit we are attempting to cherry-pick, removes whatever is inside <<<< HERE ===== (in other words, "HEAD").

So, here, the proper solution is for us to remove everything.

Save the file and after changes have been kept, start proceed to git commit.

```
gitadd -A
gitcommit
```

Commit title and description have already been filled. Press ctrl+ x to exit nano editor, and now we can push our commit.

```
Git push origin master
```

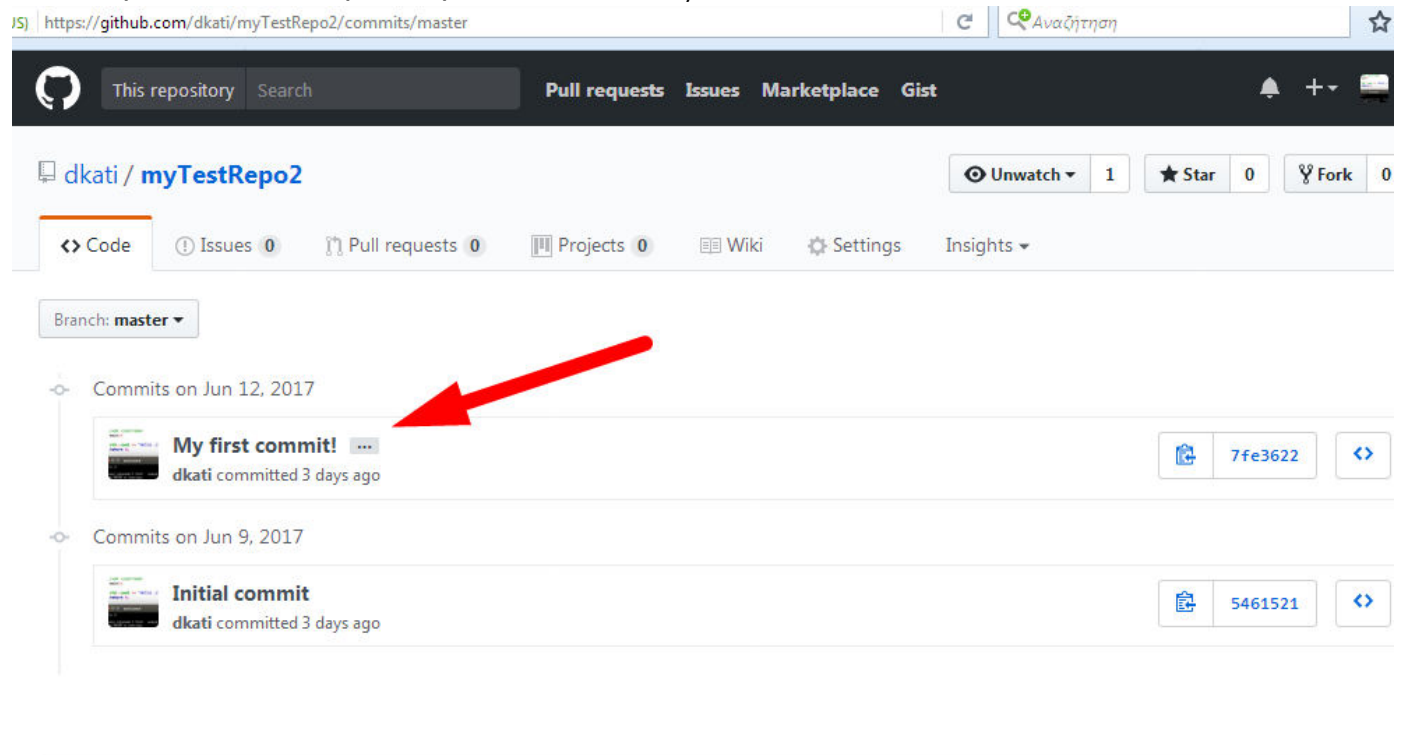
We should receive an error about remote. So, as before...

```
git remote add origin git@github.com:dkati/myTestRepo2.git
git push origin master
```

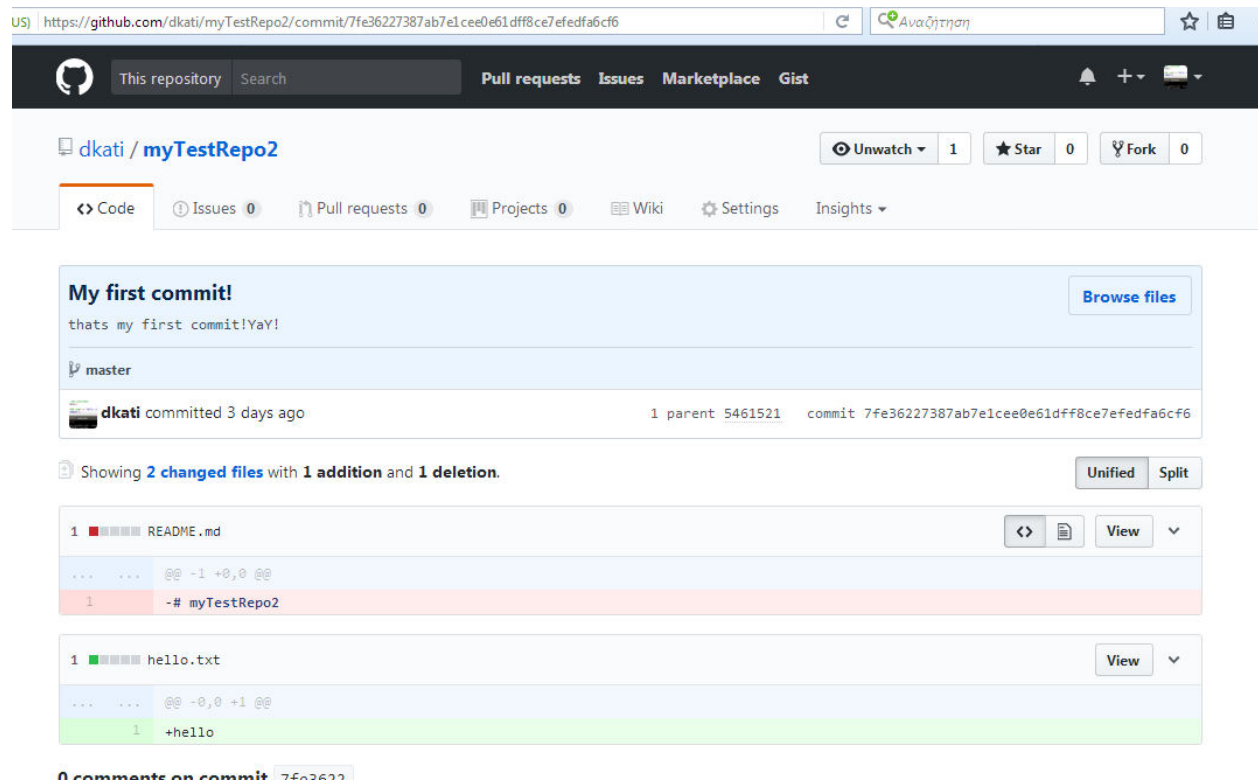
```
~/myrossources/myTestRepo2LocalDir$ git add -A
~/myrossources/myTestRepo2LocalDir$ git commit
Recorded resolution for 'README.md'.
[master 7fe3622] My first commit!
Date: Fri Jun 9 16:22:05 2017 +0300
2 files changed, 1 insertion(+), 1 deletion(-)
create mode 100644 hello.txt
~/myrossources/myTestRepo2LocalDir$ git push origin master
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
~/myrossources/myTestRepo2LocalDir$ git remote add origin git@github.com:dkati/mytestrepo2.git
~/myrossources/myTestRepo2LocalDir$ git push origin master
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 313 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 1 (delta 0)
To git@github.com:dkati/mytestrepo2.git
5461521..7fe3622 master -> master
~/myrossources/myTestRepo2LocalDir$
```

This time, commit has been, indeed, created successfully.



The screenshot shows the GitHub repository page for `dkati / myTestRepo2`. The URL is `https://github.com/dkati/myTestRepo2/commits/master`. The page displays the commit history for the `master` branch. A red arrow points to the commit titled "My first commit!". The commit was made by `dkati` 3 days ago. The commit hash is `7fe3622`. Below it, the "Initial commit" is also shown, made by `dkati` 3 days ago, with hash `5461521`.



The screenshot shows the GitHub commit page for the commit `7fe36227387ab7e1cee0e61dff8ce7efedfa6cf6`. The commit is titled "My first commit!" and was made by `dkati` 3 days ago. The commit message is "thats my first commit!YaY!". The commit is on the `master` branch. The page shows 2 changed files: `README.md` and `hello.txt`. The `README.md` file has 1 addition and 1 deletion. The `hello.txt` file has 1 addition. The commit has 0 comments.

What we notice regarding the two commits, is that there were different changes made in myTestRepo commit and myTestRepo2 commit.

This is reason why we encountered a conflict during cherry-pick. Rarely does it happen for cherry-pick to be executed, successfully, without any conflicts.

Whenever there is a commit pulled through the process of cherry-pick, and the author's name is different from the user's, who is trying to copy that commit, then the final cherry-picked commit appears as shown below:



Here, the original author is **brinlyau** and the one who cherry-picked is **Dmole**.

ΕΠΑΝΕΓΓΡΑΦΗ ΤΗΣ ΙΣΤΟΡΙΑΣ ΤΩΝ COMMIT(ADVANCED USERS)

As it has been made clear, commit deletion is not possible. However, there is a command that restores the whole commit history to an older version, removing any commit created from that point and on. Useful as it may be, it is also considered to be kind of dangerous because of its ability to remove every single commit one has pushed, from the point where the reset has been set to, and further.

Practically, this command does not delete the commits themselves but their presence in the commit history instead. Also, the abovementioned commits are discarded from the locally stored source code. The process of recalling the deleted commits is difficult and requires expertised knowledge in Github shell use.

Command's syntax is:

git reset --hard SHA_CODE (double dash)

e.g. If I wish to "delete" myTestRepo2's last commit, then I need to just reset to the commit before the last one.

```
git reset --hard 5461521f50d472f7950f330608438a70634b435e
```

and then, push is required for the process to complete

```
git push -f origin master
```

Parameter **-f** means «force». Github needs this parameter to be set to **-f** for security reasons. Should we try to push without passing the **-f** parameter, our push request would be rejected.


```
~/myrossources/myTestRepo2LocalDir$ git reset --hard 5461521f50d472f7950f330608438a70634b435e
HEAD is now at 5461521 Initial commit
~/myrossources/myTestRepo2LocalDir$ git push -f origin master
Warning: Permanently added the RSA host key for IP address '192.30.253.113' to the list of known hosts.
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:dkati/mytestrepo2.git
+ 7fe3622...5461521 master -> master (forced update)
~/myrossources/myTestRepo2LocalDir$
```

In case we regret resetting, we can recall the removed commit by, once again, reset (needed code is `7f3622`).

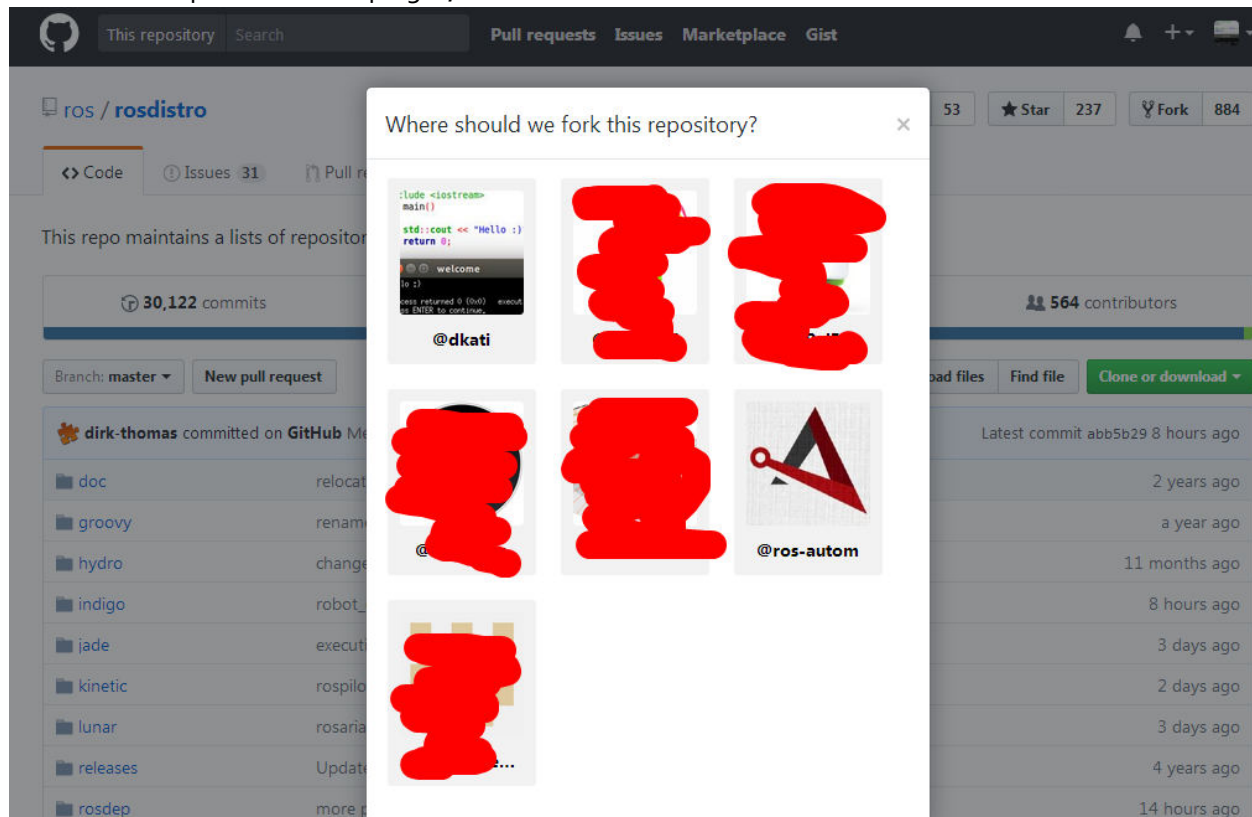
```
git reset --hard 7f3622
```

At this point, it should be clear that git reset acts as an index in the commit history that can be placed on whichever point of the history's timeline we wish to "move".

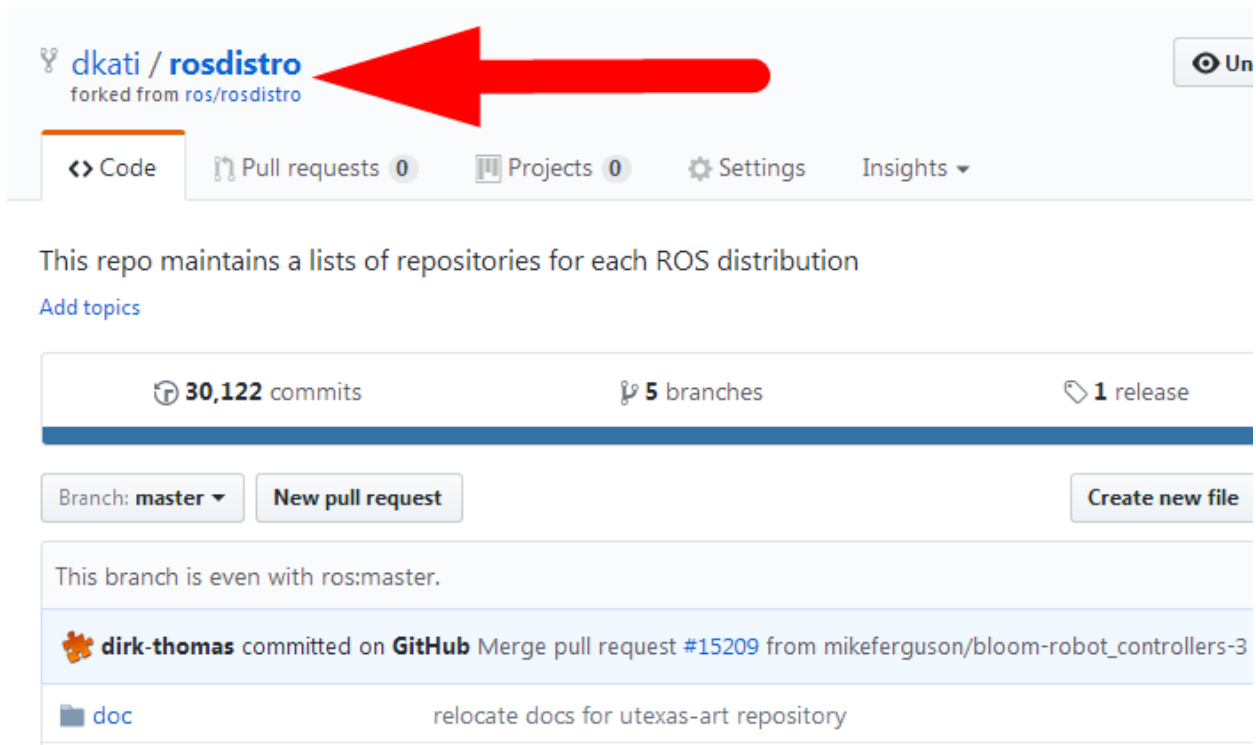
FORKING.LEGALLY COPYING WHOLE REPOSITORIES

Forking is the name of the process during which one's whole repository is copied. It can be done in a few clicks and it is very easy.

First, head to the repository which you want to fork. Assume, this repository is `github.com/ros/rosdistro`. Click the fork option on the top right, and chose which account we want to send it to.



The repository is now available in our profile, as shown below. It is obvious that this repository is part of ros/distro project.



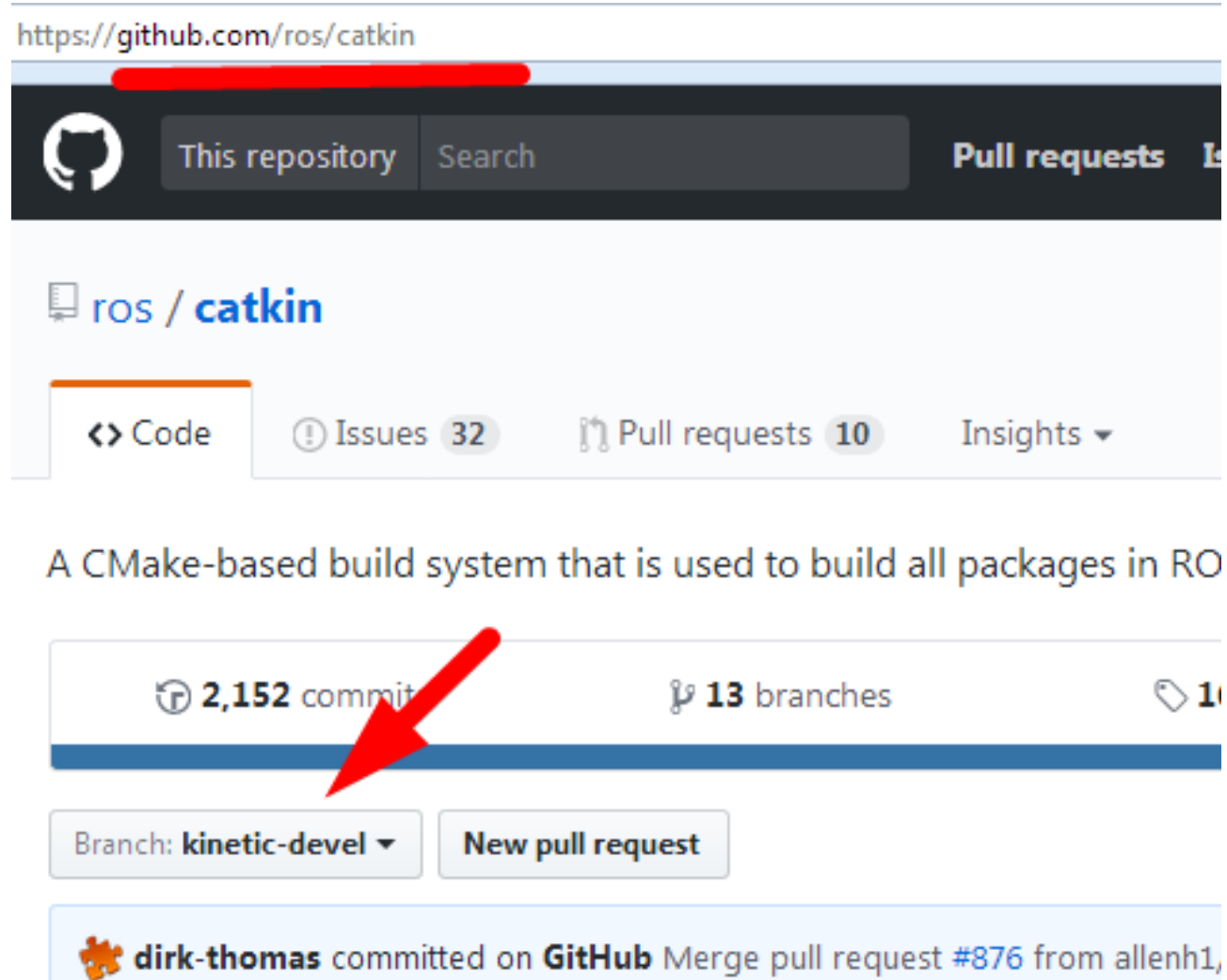
Thereafter, we use the project just like we worked on the two previous repositories of the past examples.

We must always remember that during the 1st push, we must set up the remote (git remote add origin).

WRITING THE WHOLE STORY OF ANOTHER REPOSITORY

Assume that we have a repository with branch master and we want to create a whole branch which will, practically, be a 3rd repository's branch.

So, should I want to create a branch, in myTestRepo, containing kinetic-devel branch from ros/catkin repository..



Obviously, it's not possible to fork, since we do not need all branches. As a result, we must work like this:

- Pull the whole repository
- Navigate to kinetic-devel branch
- Create a local branch which comes from kinetic-devel (picture with arrows, circles etc.)
- Push it

Thereafter, move to myTestRepoLocalDir

```
git clone https://github.com/ros/catkin kinetic-devel
cd kinetic-devel
git branch
```

At this point, we are on the branch that we have downloaded and we are going to create our own branch

```
git branch mykinetic
git branch
```

Now we see that we have two branches and we know mykinetic branch comes from kinetic-devel. Local branch is green colored which we are currently on. Switch branch

```
git checkout mykinetic
```

Since we are on our branch, we push whatever there is on the branch (source and commit history). Trying to git push origin mykinetic, will fail to push because we are not part of the ROS team. What we need is to create a new origin.

```
git remote add myorigin git@github.com:dkati/mytestrepo.git
git push -f myorigin mykinetic
```

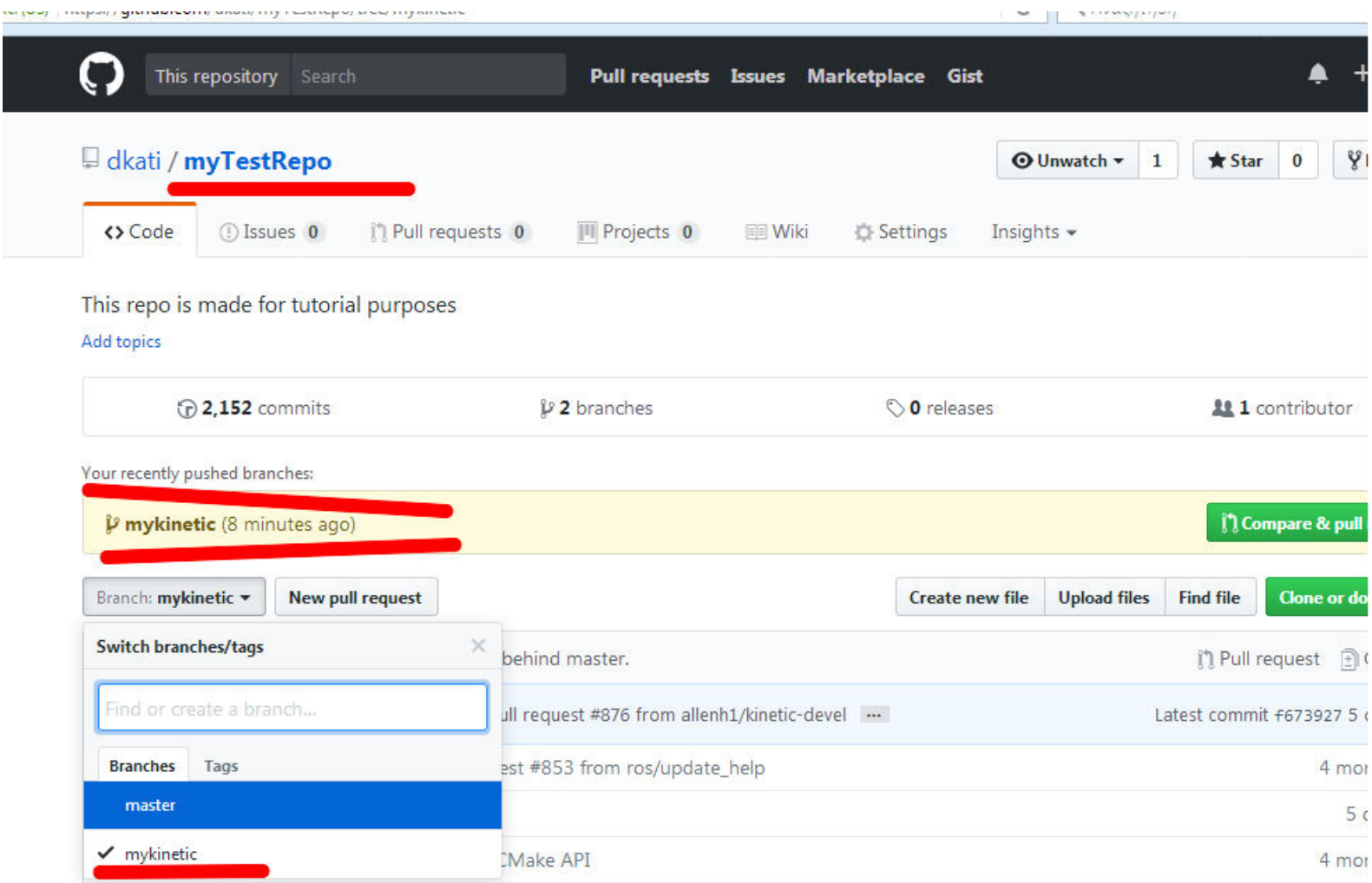
Parameter -f is used in order to force Github to create a new branch on my page.

```

~/myrossources/myTestRepoLocalDir$ git clone https://github.com/ros/catkin kinetic-devel
Cloning into 'kinetic-devel'...
remote: Counting objects: 12110, done.
remote: Total 12110 (delta 0), reused 0 (delta 0), pack-reused 12109
Receiving objects: 100% (12110/12110), 3.34 MiB | 1.26 MiB/s, done.
Resolving deltas: 100% (6467/6467), done.
Checking connectivity... done.
~/myrossources/myTestRepoLocalDir$ cd kinetic-devel
~/myrossources/myTestRepoLocalDir/kinetic-devel$ git branch
* kinetic-devel
~/myrossources/myTestRepoLocalDir/kinetic-devel$ git branch mykinetic
~/myrossources/myTestRepoLocalDir/kinetic-devel$ git branch
* kinetic-devel
  mykinetic
~/myrossources/myTestRepoLocalDir/kinetic-devel$ git checkout mykinetic
Switched to branch 'mykinetic'
~/myrossources/myTestRepoLocalDir/kinetic-devel$ git push -f origin mykinetic
Username for 'https://github.com': dkati
Password for 'https://dkati@github.com':
remote: Permission to ros/catkin.git denied to dkati.
fatal: unable to access 'https://github.com/ros/catkin/': The requested URL returned error: 403
~/myrossources/myTestRepoLocalDir/kinetic-devel$ git remote add myorigin git@github.com:dkati/mytestrepo.git
~/myrossources/myTestRepoLocalDir/kinetic-devel$ git push -f myorigin mykinetic
Counting objects: 11112, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4919/4919), done.
Writing objects: 100% (11112/11112), 3.15 MiB | 1.23 MiB/s, done.
Total 11112 (delta 5839), reused 11108 (delta 5836)
remote: Resolving deltas: 100% (5839/5839), done.
To git@github.com:dkati/mytestrepo.git
 * [new branch]      mykinetic -> mykinetic
~/myrossources/myTestRepoLocalDir/kinetic-devel$

```

Finally, in our repository, we have our new branch which is exactly the same with the original ROS branch.



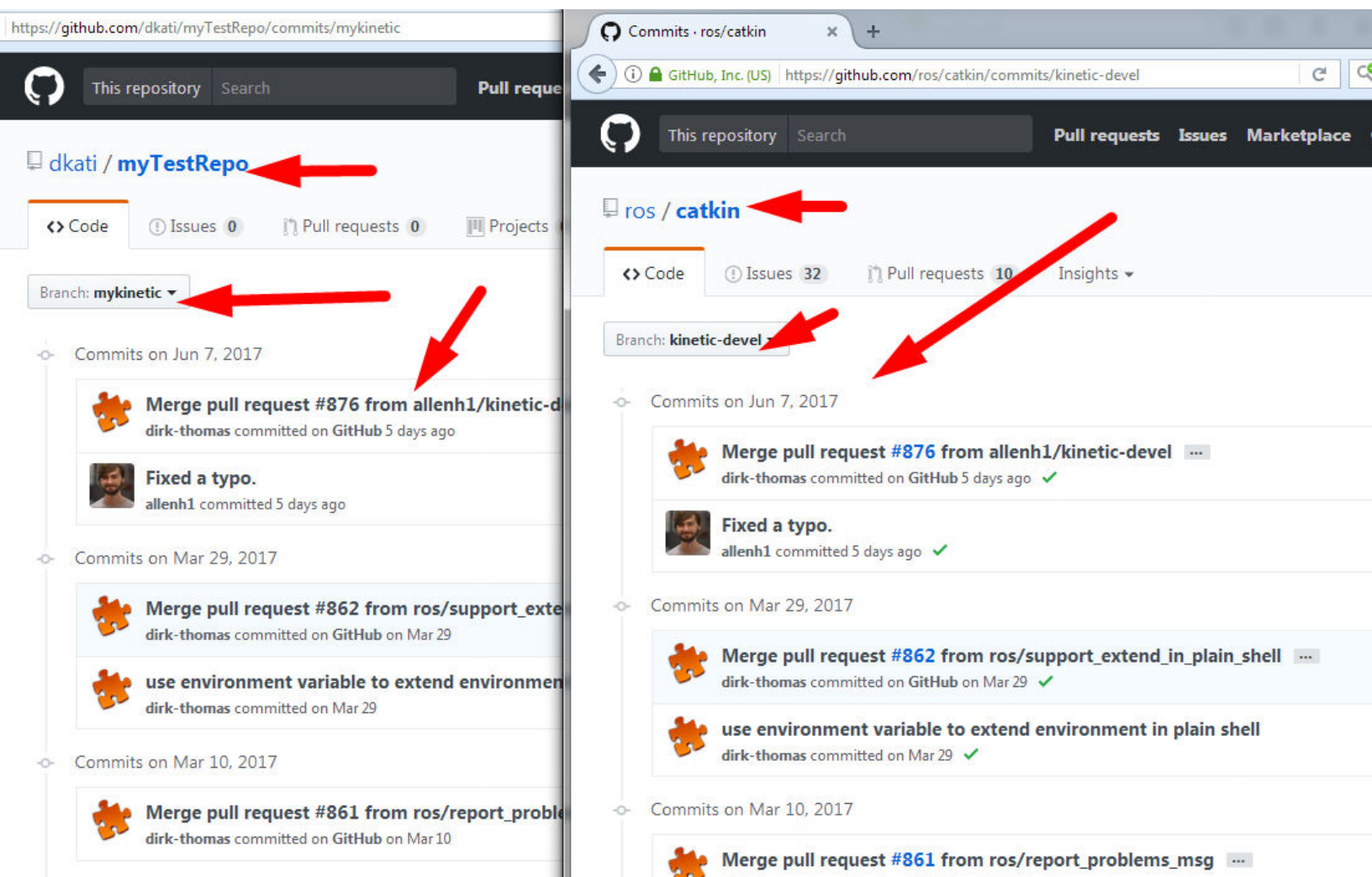
Having finished with Github, let's make a brief cleaning of our code

```
cd ../../  
rm -rf myTestRepoLocalDir  
repo sync myTestRepoLocalDir --force-sync
```

--force-sync:

We use these parameters, during every repo sync, so as to have git replace any possible old errors, in our local source, with the new pulled changes.

It is also worth mentioning that even the commit history is identical with the original ROS repository.



COMMANDS APPENDIX

- `git branch`
Shows all available local branches as well as the branch on which we are currently on (green colored text)
- `git branch <branch name>`
Creates a new branch that comes from the branch we are currently on
- `git checkout <branch name>`
Switches to another branch
- `git checkout -b <branch name>`
Creates a new branch and switches to it
- `git branch -D <branch name>`
Deletes a branch. The branch on which we are currently on, cannot be deleted.
- `git clone <github link from repository> <directory>`
Downloads the default branch from a repository and saves it in a new folder named after the directory's name. The directory in which our terminal aims at, will determine the location of the folder.
- `git clone -b <branch> <github link from repository> <directory>`
Downloads the branch from a repository and saves it in a new folder named after the directory's name. The directory in which our terminal aims at, will determine the location of the folder.
- `git fetch <github link from repository> <branch name>`
Downloads the repository's commit history. Command must be executed in the directory that I plan to cherry-pick to.
- `repo init -u git://<link from manifest>.git -b master`
Manifest's initialization
e.g. `repo init -u git://github.com/dkati/myproject-manifest.git -b master`
- `git cherry-pick <SHA>`
Legally copies a commit with a specific SHA
- `git reset --hard <SHA>`
Set commit history to a specific commit
- `git revert <SHA>`
Reverts a specific commit
- `git commit`
Creates a commit
- `git add -A`
Adds one's changes to the commit
- `git push <remote name> <branch name>`
Uploads a commit, through remote, to a branch
- `git remote add <remote name> git@github.com:<repo name>.git`
Creates remote
- `git remote remove <remote name>`
Deletes remote
- `Repo sync` or `repo sync --force-sync`
Synchronizes local source code with source code on Github

- `-force-sync`:
Parameter used in order to force github to overwrite one's local changes while synchronizing

Every command must be run inside the root directory of each repository (or simply, where .git folder is located)

***=

When Github reports "Recorded preimage", it means that Github has kept both the error and the user's solution to it. This aims at better future automatic trouble shooting for such problems cause during the process of cherry-pick. However, there are times when one does not wish Github to remember how a problem was solved, since the solution may differ from case to case.

Using the following command will have Github "forget" the steps one followed to deal with a conflict

```
git rerere forget *
```