Git:

Overview:

- Git itself can be imagined as something that sits on top of your file system and manipulates files. Even better, you can imagine Git as a tree structure where each commit creates a new node in that tree.
- Nearly all Git commands actually serve to navigate on this tree and to manipulate it accordingly.
- Git is just as legitimate in the enterprise as any other VCS.
- Installation instructions can be found here:

  https://linode.com/docs/development/version-control/how-to-install-git-on-linux-mac-and-windows/

The full documentation can be found at:

https://git-scm.com/docs

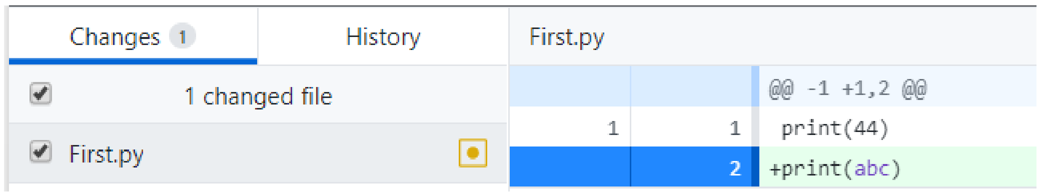
Terminology:

- **master -** the repository's main branch. Depending on the work flow it is the one people work on or the one where the integration happens
- **clone -** copies an existing git repository, normally from some remote location to your local environment.
- **commit -** submitting files to the repository (the local one); in other VCS it is often referred to as "checkin"
- **fetch or pull -** is like "update" or "get latest" in other VCS. The difference between fetch and pull is that pull combines both, fetching the latest code from a remote repo as well as performs the merging.
- **push -** is used to submit the code to a remote repository
- **remote -** these are "remote" locations of your repository, normally on some central server.
- **SHA -** every commit or node in the Git tree is identified by a unique SHA key. You can use them in various commands in order to manipulate a specific node.
- **head -** is a reference to the node to which our working space of the repository currently points.
- **branch -** is just like in other VCS with the difference that a branch in Git is actually nothing more special than a particular label on a given node. It is not a physical copy of the files as in other popular VCS.
- **Merge** - git-merge - Join two or more development histories together
- **Checkout** - Switch branches or restore working tree files
- **Revert** - Revert some existing commits

Commit:

- In order to make a commit, we will first need to make a change in our project.

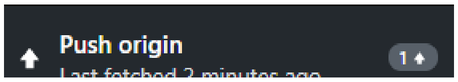
1. Open your code (IDE / notepad / …) and make a small difference (even a comment).
2. Go back to Github Desktop and you will see the change

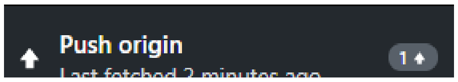


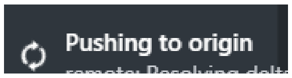
```
@@ -1 +1,2 @@
1  1   print(44)
   2  +print(abc)
```

3. If you want the change to be saved, you will first need make sure file checkbox is checked
4. Add commit **Summary** and **description** (optional) which explains what you changed (please add sense!) and choose **commit** option.
5. If you want the change to not apply, press right click on file → Discard.
6. If you want the change to not be saved, press right click on the file and choose discard changes

Push:

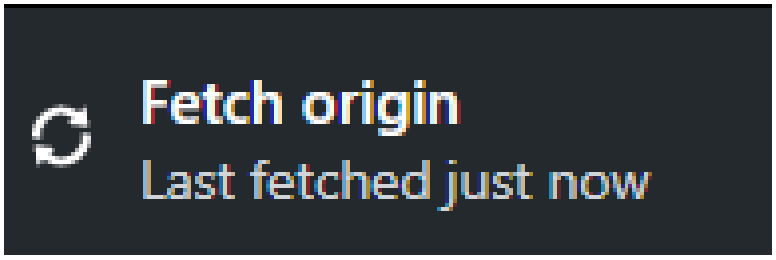
- Currently you code changes are saved locally only!
- Pay attention to the 1 that appears, which tells you that your local is "before" your remote in one change (commit).


- If you want your changes to be added to your remote repository, you will need to push it
- Simply, press on push origin: 

you will see: 

- You can now visit your remote repository (Github) and see the changes.

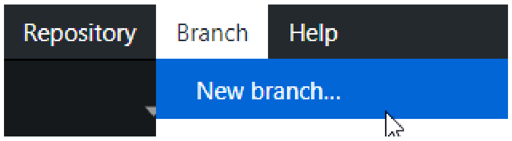
Fetch:

- In order to determine whether your working copy (local) is updated to your remote you can use **Fetch**.
- This will not make any changes! It will only will check.
- Simply press Fetch:


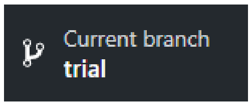
  - Think about it as "Look for updates.."

Branching:

- As we learned before most of the times we will not want to work on our master (production) and we will want to create a new feature as an example.
- What we will do will be creating a new branch, working on our feature, and the adding it into our master.
- To use it we will create a new branch with the following instructions
  - Press branch menu → New branch → Give it a name → press create branch

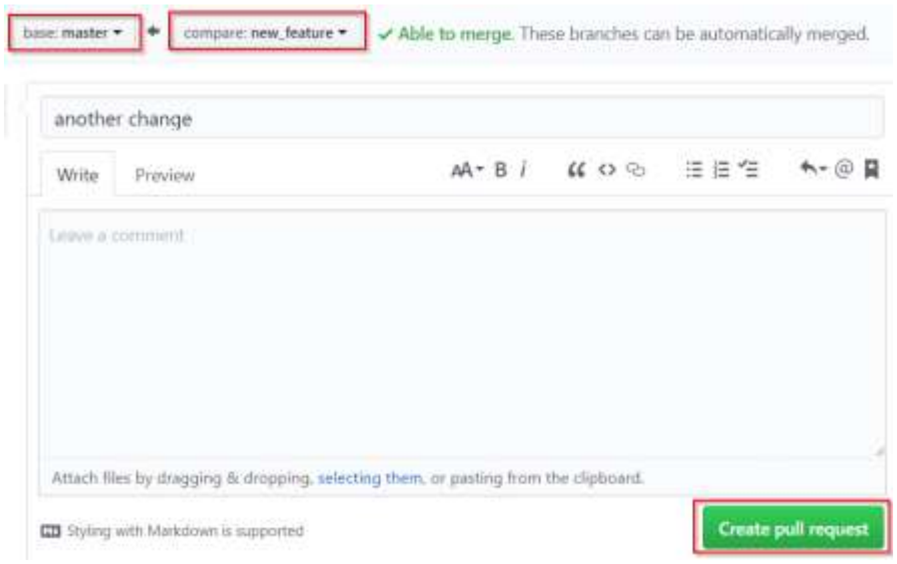
  - Make sure you see your new branch name under current branch:

  - Now let's do another code change, commit and push.

Pull request:

- Enter your Github repository and choose **“New pull request”**
- Choose a branch to pull from and a branch to pull to and press **Create pull request**



These branches can be automatically merged.

another change

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

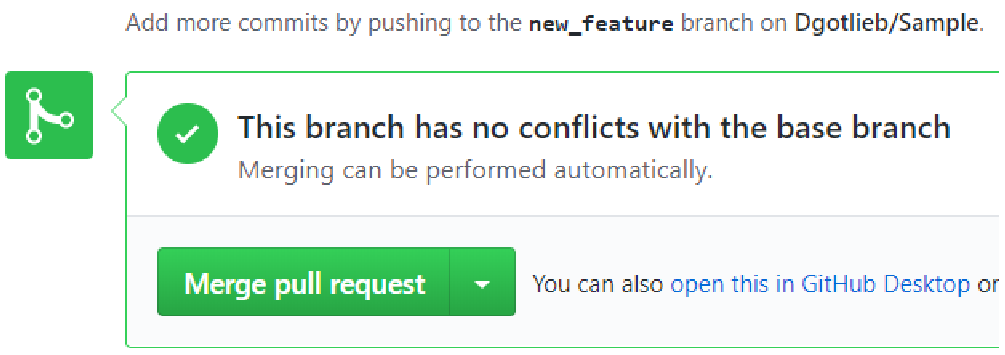
Styling with Markdown is supported

Code review:

- When a developer is finished working on an issue, another developer looks over the code and considers questions like:
  - Are there any obvious logic errors in the code?
  - Looking at the requirements, are all cases fully implemented?
  - Does the new code continuous existing style guidelines?
  - Usually a pull request will only be able to pass after another reviewer approved it.

Merge:

- If your Pull request finished successfully, you will be able to merge your changes:


Add more commits by pushing to the `new_feature` branch on Dgotlieb/Sample.

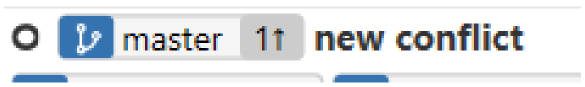
This branch has no conflicts with the base branch
Merging can be performed automatically.

- Now, press **Merge pull request** and then **Confirm merge**

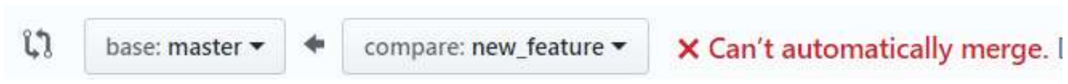
- You will now see a merge confirmation.

Conflicts:

- Sometimes you get merge conflicts when merging or pulling from a branch.
- A merge conflict usually occurs when your current branch and the branch you want to merge into the current branch have diverged.
- That is, you have commits in your current branch which are not in the other branch, and vice versa.
- Git will then tell you about the conflict

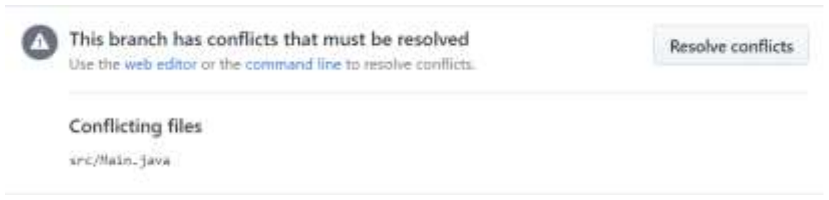


- When you will try merging, you will see the following warning:



✕ Can’t automatically merge.

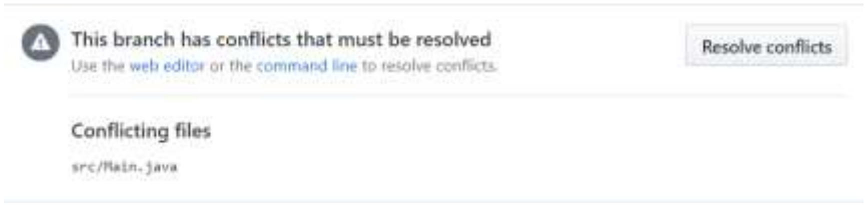
- And after that you will see:


This branch has conflicts that must be resolved
Use the web editor or the command line to resolve conflicts.

Conflicting files
`src/Main.java`

Resolving conflicts:

- To fix the conflict, press resolve conflicts button



This branch has conflicts that must be resolved

Use the web editor or the command line to resolve conflicts.

Conflicting files

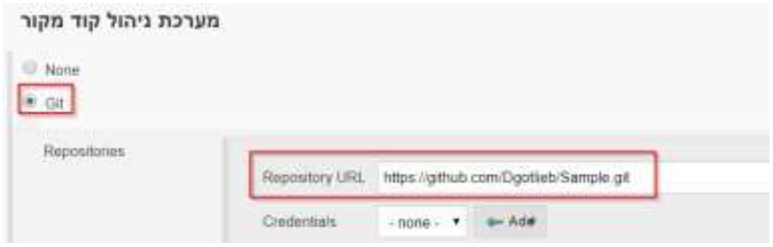
src/Main.java

- You will then be transfer into the conflict class.
- At this point you have 2 options:
    - Manual resolving (which is literally, deleting a piece of code manually).
    - Using a merging tool which can be found online

- Once the conflict will be resolved the merge will happened.

Using Git as a build trigger:

- Many times we will want to start our tests once a new code was pushed.
- For example when a new code is committed to production, we will want our test to run immediately, so issues can be discovered ASAP.
- To achieve it, enter your Jenkins job (remember what it is?) and do the following changes:
    - Under “source code management” check Git option and enter your Git address and credentials of necessary (In Github there are no credentials because it is public).



מערכת ניהול קוד מקוד

( ) None

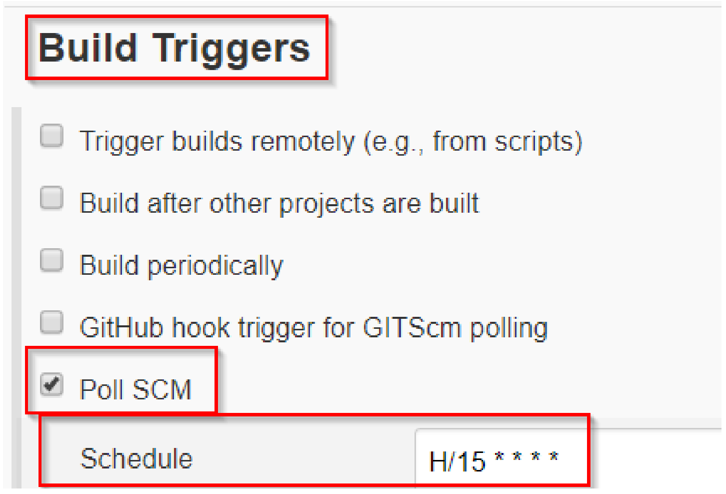
(●) Git

Repositories

Repository URL: https://github.com/Dgotlieb/Sample.git

Credentials: - none -

    - Under build triggers choose “Poll SCM” (Source Control Management)
    - Then choose every how long the job will check for a new commit.
    - The example will basically check every 15 minutes if a commit
- Was performed into your Git repository, if so, your job will run automatically.



Build Triggers

- [ ] Trigger builds remotely (e.g., from scripts)
- [ ] Build after other projects are built
- [ ] Build periodically
- [ ] GitHub hook trigger for GITScm polling
- [x] Poll SCM

Schedule: H/15 * * * *

Using Git in a pipeline:

When using pipeline we can define where the code will come from using **git** command and optionally using the same poll SCM mechanism.

In the below pipeline we will be using Git to pull code from our repository and running the python file which is inside:

```
pipeline {
agent any
   stages {
      stage('checkout') {
         steps {
            git 'https://github.com/<USER_NAME>/<PROJECT_NAME>.git'
         }
   }
      stage('build') {
         steps {
            bat 'python <PYTHON FILE>.py'
         }
               }
        }
}
```