Git Workshop Documentation

Release 0.0.1

Christoph Lange

Feb 25, 2021

CONTENTS

Here are the basic git concepts that we covered in the first workshop, that will be needed in today's workshop. Feel free to take a look, in case you forgot something.

CHAPTER

ONE

GIT BASICS

1.1 Creating a Repository

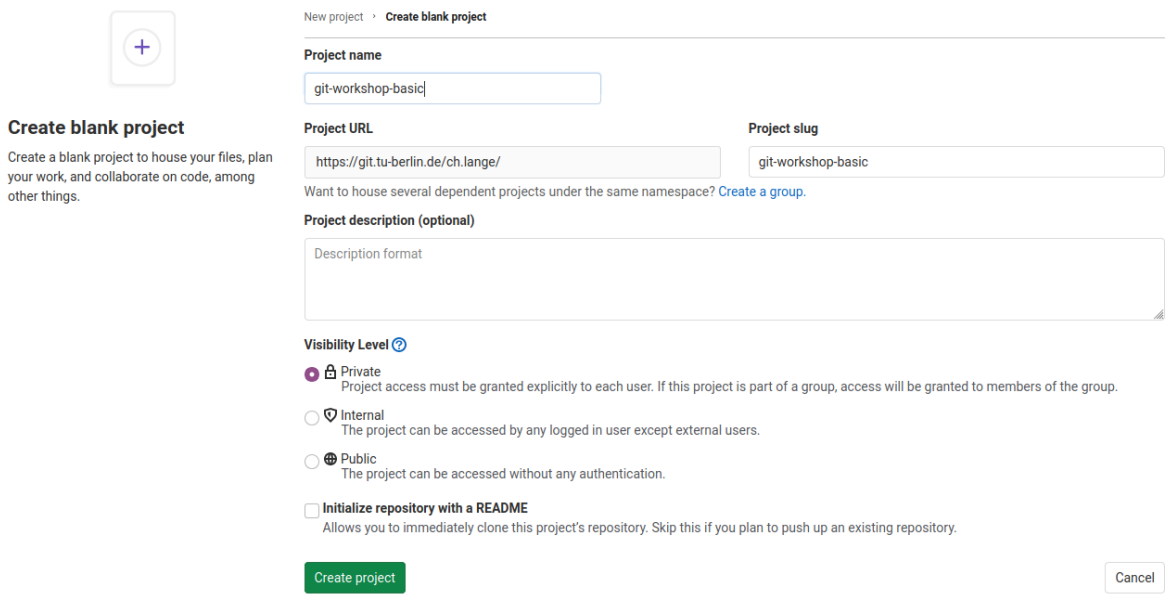
Steps

- *Creating a Repository*
  - *Create Project on GitLab*
  - *Use Project Template*
  - *Sync Local and Remote Repository*

The basic idea is to create a repo on the remote server. Then we create some content for the repository locally and finally we want to sync this content to the remote server.

1.1.1 Create Project on GitLab

First of all you want to create a repository on GitLab/GitHub. Therefore, go to the URL of your GitLab Server, i.e. https://git.tu-berlin.de/kiwi-git-workshops. Then you click on **New Project** and select **Create blank project**. Afterwards you may choose a name for your repository



and click **Create project**. Now we created an empty project on the remote server.

1.1.2 Use Project Template

Now we create a folder with some code on our local machine. Therefore we use a template via the following steps:

1. Open a terminal
2. Install the python package cookiecutter

```
pip3 install cookiecutter
```

3. Use *cd* to navigate to the directory that you want to start a repository.

```
cd path/to/your/git-projects
```

4. Create your python package with

```
cookiecutter https://github.com/spirousschuh/cookiecutter-git-workshop-
↪documentation
```

5. Specify the template parameter. Now you will see

```
author_name [Josephine Doe]:
```

   This is a question. "What should be the name of the author?" and requires your input. You can either press *Enter*, then the author_name is set to the default option Josephine Doe. Or you can enter another name.

6. Answer the questions that will be prompted to you or press *Enter* to choose the default value. You do not need to reveal your real data, as it is a toy project anyway. But you could choose answers like these:

```
(tmp1) christoph@christoph-ThinkPad-P53:~/letter_to_uncle/tmp$ cookiecutter https://github.com/spirousschuh/cookiecutter-git-workshop-basics
You've downloaded /home/christoph/.cookiecutters/cookiecutter-git-workshop-basics before. Is it okay to delete and re-download it? [yes]: yes
author_name [Josephine Doe]: Christoph
author_email [your@address.eu]: mail@to.me
package_name [git_workshop_basic]: my_image_package
package_description [A lightweight python package to practise some git]: This package does simple image manipulations
package_url [https://git.tu-berlin.de/you/your_repo_name]: https://git.tu-berlin.de/ch.lange/my_image_package
(tmp1) christoph@christoph-ThinkPad-P53:~/letter_to_uncle/tmp$
```

Pay attention at the third question. The answer to that question will be the name of the folder where you can find your package later.

Now we created a folder of code locally.

1.1.3 Sync Local and Remote Repository

In this section we will syncronize our local folder with the remote git server. Right know they do not know about each other.

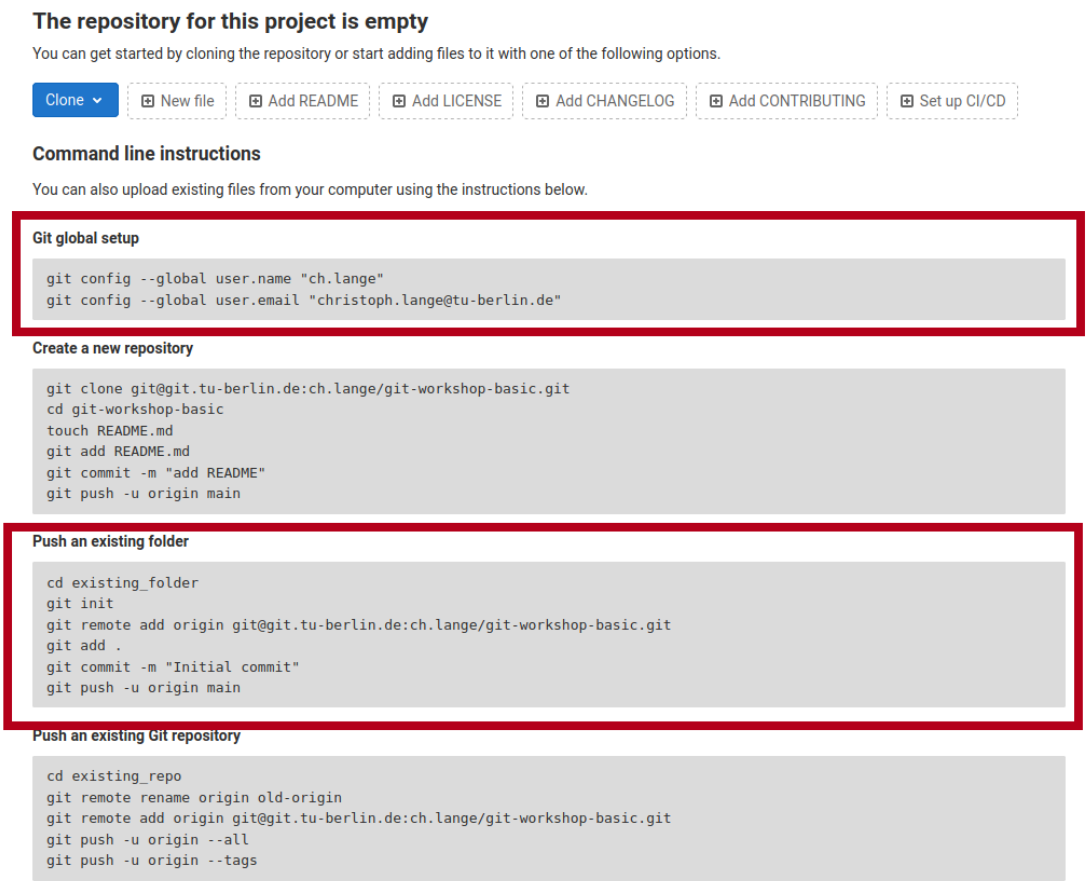
1. Go the folder that you just created in the last step

```
cd my_image_package
```

The name of the folder corresponds to your answer to the question

```
package_name [git_workshop_testing]: my_image_package
```

2. Go back to your browser and open the remote server url (https://git.tu-berlin.de). Then go to the project that you just created in the section *Create Project on GitLab*. As it is an empty project the landing page should look like this:



3. Follow the step that are displayed under **Git global setup** (first red box) one by one, i.e. you copy each line to your terminal and press *Enter*.

4. Follow the steps you find in the section **Push an existing folder** (second red box). You need to replace *cd existing_folder* with the *project-name* you chose in step 6. In case you forgot the package name you can check it with *ls -l* which displays the content of the current directory. (if you get an error like *error: src refspec main does not match any* you need to replace main with master)
5. Install your new package in editable mode

```
pip install -e .
```

6. Go to your project webpage *https://git.tu-berlin.de/your_name/your_project*. When you see a basic README.md file you succeeded.

1.2 Git Workflow

1.2.1 Idea

This is a concise manual to a basic Git workflow. You can find more details here. For each step you can find instructions how to follow that workflow using PyCharm. There is different ways to achieve the same goal without PyCharm. Once you are familiar with the basic concepts you can use any tool you like.

1.2.2 Instructions

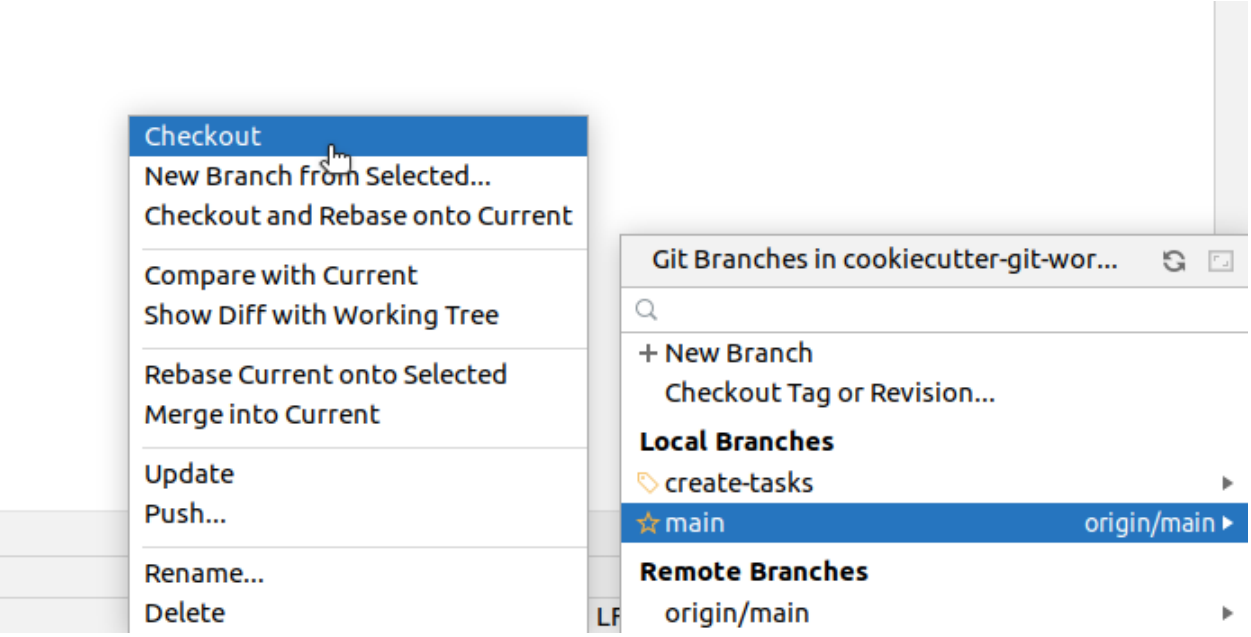
Once you have an idea what you want to achieve the following steps will help you to get there.

Steps

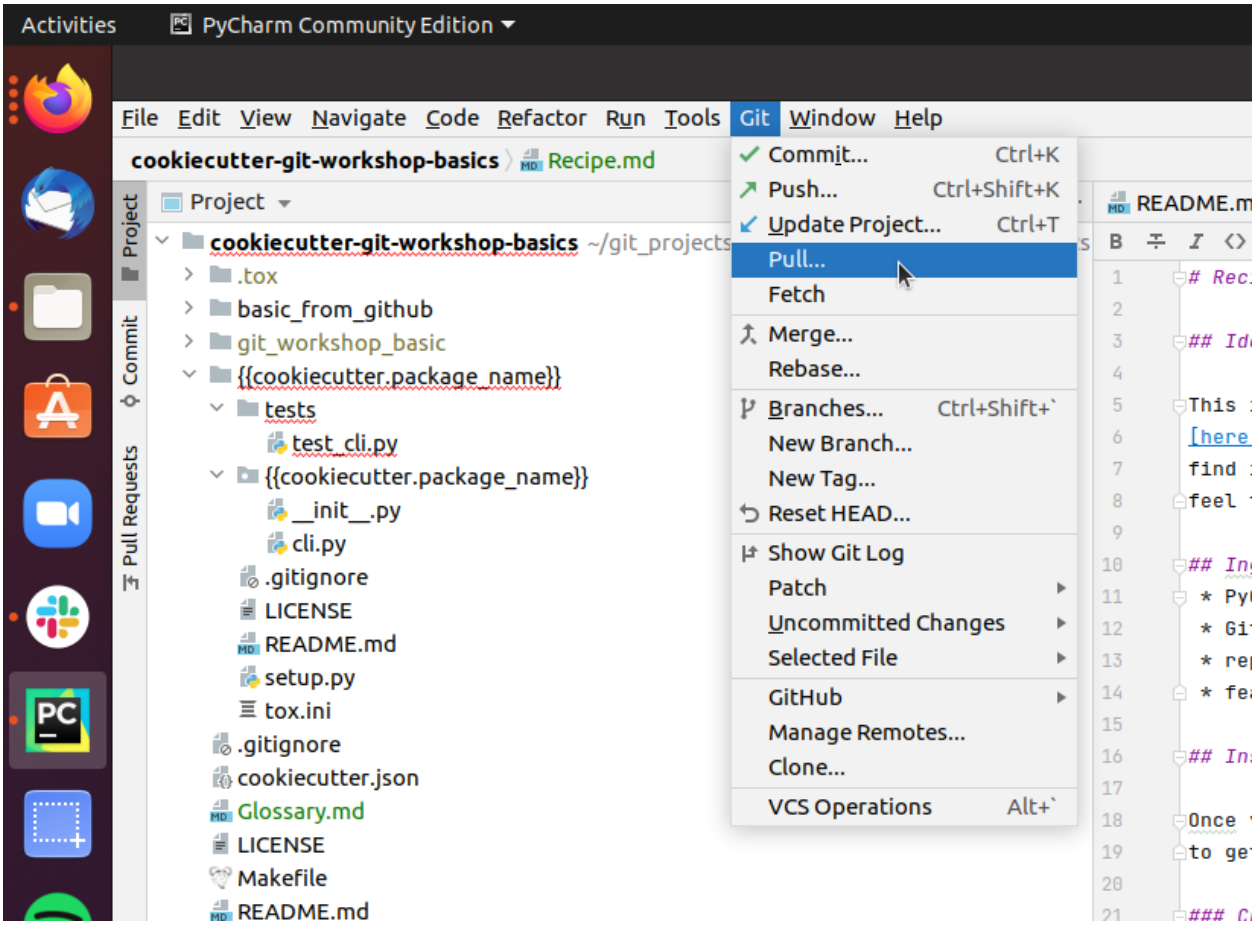
- *Update Local*
- *Create Branch*
- *Add Commits*
- *Push Branch*
- *Merge Request*
- *Discussion*
- *Merge Branch*

Update Local

First we want to make sure to use the newest version of the repositories main branch. Therefore we click on the button in the bottom right corner next to the patlock. Then we see a context menue like this that displays all the local branches.

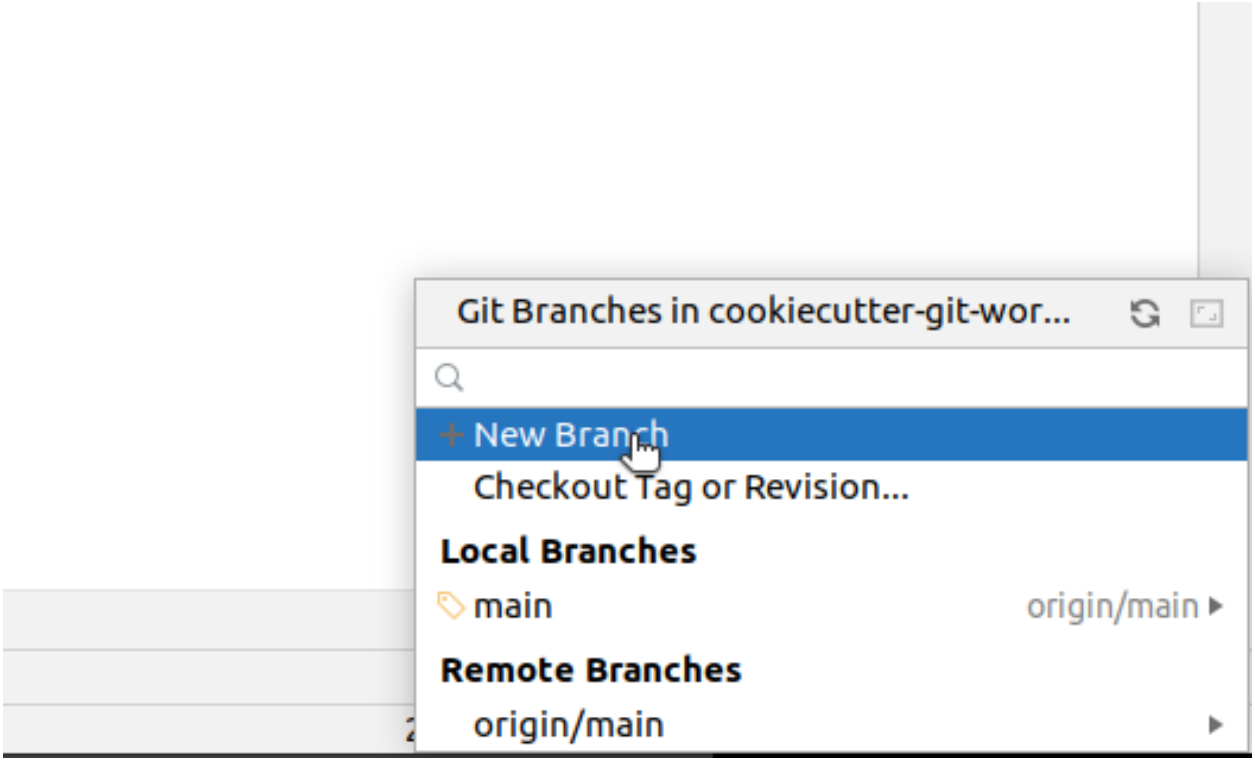


Click on the main/master branch and choose "Checkout" in the second context menue to switch to the main/master branch. Now we need to make sure that your local main/master branch is up to date with the upstream main/master. Therefore we pull the newest state from upstream. In the upper left corner we can find the menue bar, click on "Git" and choose pull in the pull down menue.



Create Branch

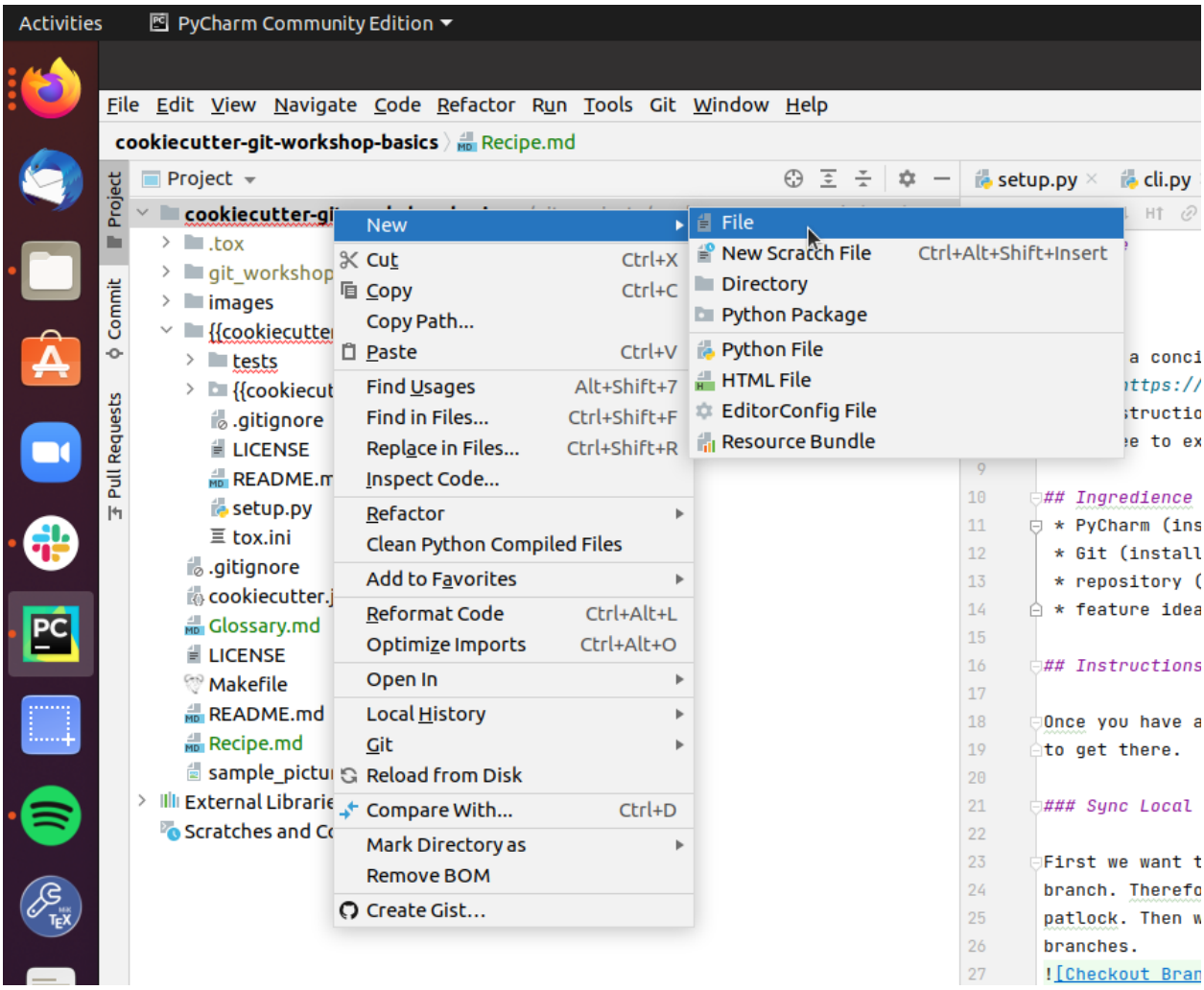
Now we create a branch to implement our feature. In order to do so move your cursor to the buttom right corner and click on your current branch name, which should be main/master, next to the patlock.



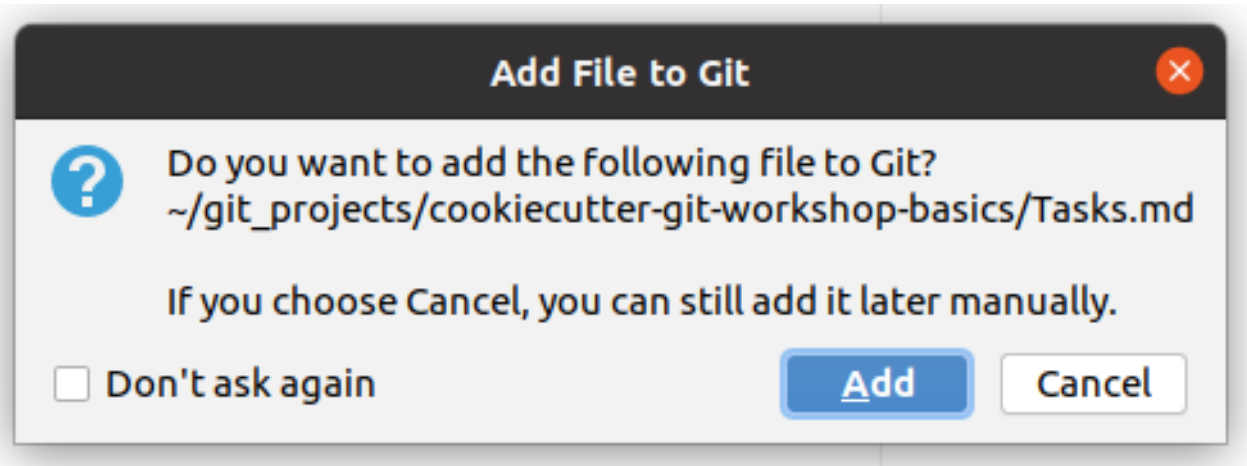
Within the context menue click on "New Branch" and enter a branch name that relates to your feature idea.

Add Commits

Now you need to add, change or delete some content in the repository to achieve your goal. For instance you want to add a new file "Tasks.md". Then you make a right click onto the folder that should contain your new file.



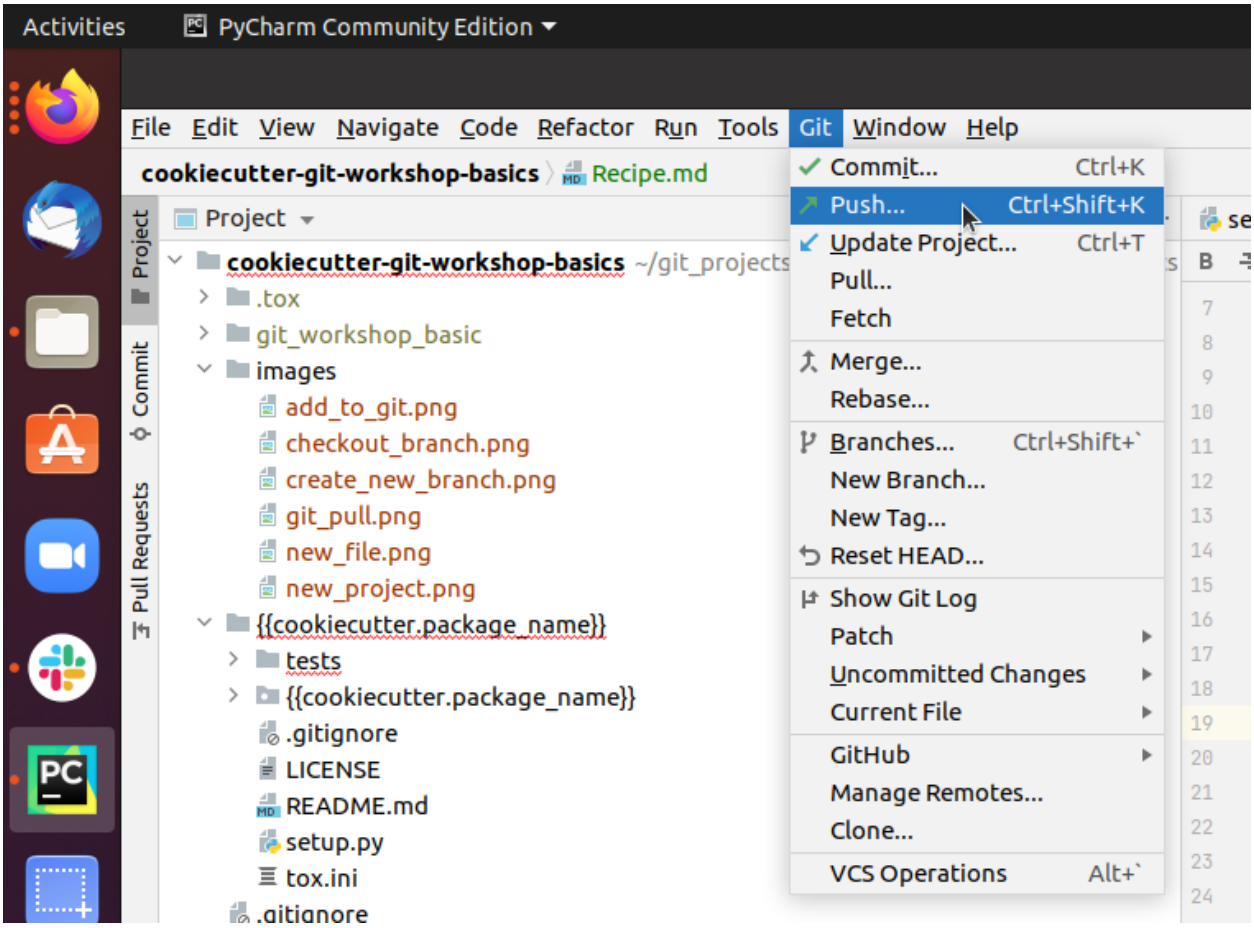
In the context menue select "new" and "File" and enter the filename in the consecutive prompt. Then PyCharm wants to know if Git should look after your new file.



Normally that is a good idea and you shall choose "Add".

Push Branch

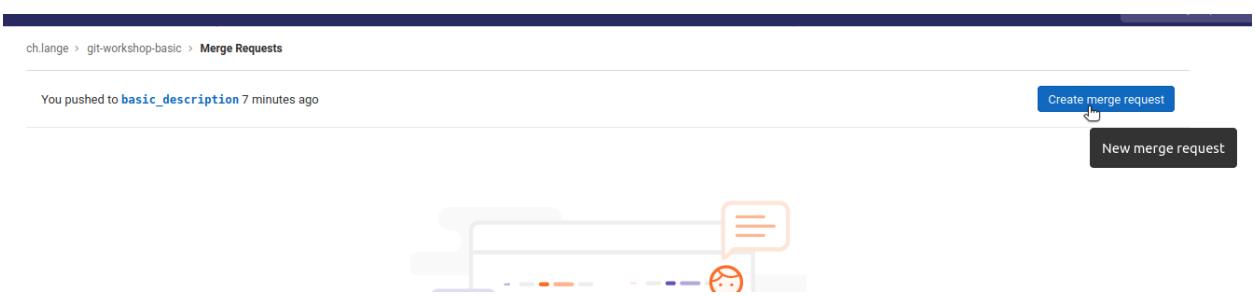
Now you want to push the branch with your changes to the upstream server. This way you create an identical copy of your local branch on the server. To do so



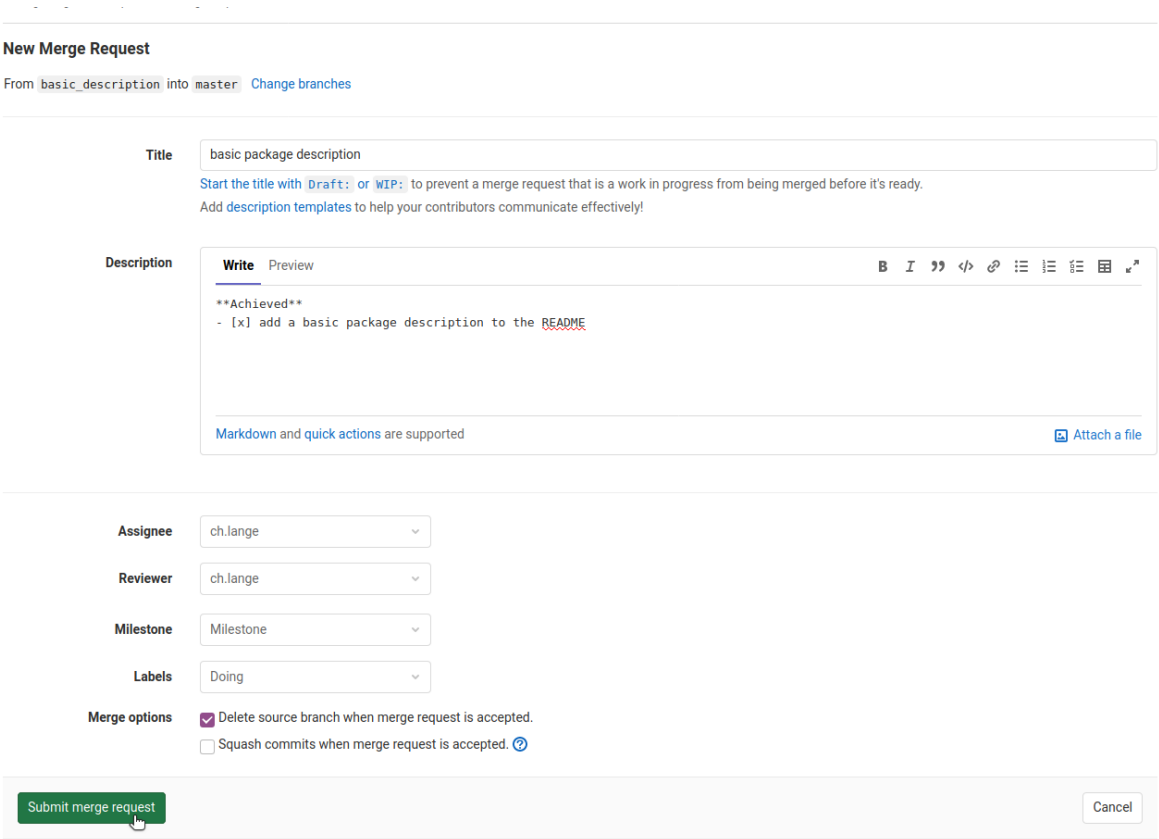
go to the upper left corner where you can find the menue bar and click on *Git* and choose *push* in the pull down menue.

Merge Request

Now that you pushed your local branch to the upstream server, you want to create a merge request on the server. Therefore open your browser and go to *https://git.tu-berlin.de/your_name/your_project/*. On the left hand side you click on *Merge Requests*. Then you get to a page that looks like this:



Here click on *Create merge request* to create a request to merge *your_branch* (here: "basic_description") into main/master. Then you can add a description



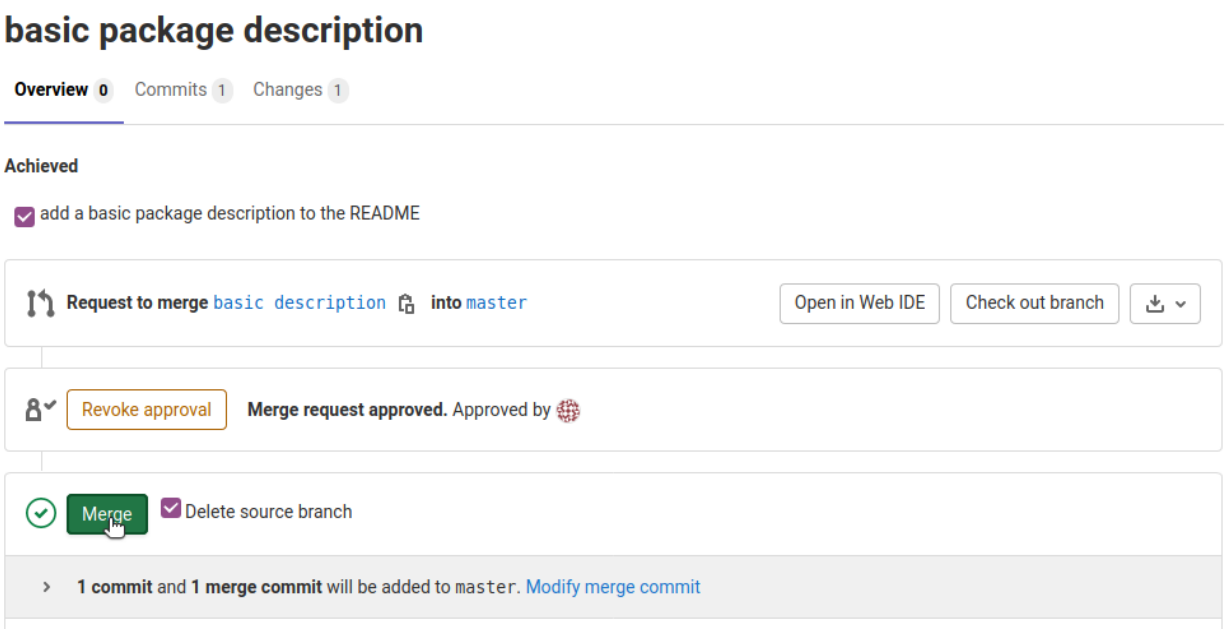
and assign a reviewer. Finally submit you merge request.

Discussion

Now the reviewer of the merge request checks your changes and gives you feedback. After some discussion you might want to go back to step 3 and add additional commits to change the current state. For the sake of practising some interations in the workshop, you can just approve your own merge requests and continue.

Merge Branch

When all discussions are done and you are sure that your changes improve the main/master branch, it is time to merge your branch by



clicking on **Merge**.

Now master on the upstream server is newer than your local branch and its time to start all over again (*Update Local*).

CHAPTER

TWO

DOCUMENTATION BASICS

Today we will learn how to write documentation and publish it on the web. At first we will start with the basics of documentation.

2.1 Documentation 101

The most basic piece of documentation is a Readme.

2.1.1 Readme

When creating a repository it is a good practise to add a Readme right from the start. Its the first point of touch with your repository.

Content

A basic Readme should contain the following:

- the repository's purpose
- an installation guide
- a small code snippet of a typical use-case
- a note on contribution
- how you liscense your project (default choice: https://choosealicense.com/licenses/mit/)

Purpose

I advice you to write documentation, because

- essential if someone else will start using your software
- you reflect your own design choices while explaining them to the user
- in case of multiple users its the most efficient way to handle questions
- great lookup for yourself

2.2 RestructuredText

The standard file format that is used in python to write technical documentation is RestructuredText.

2.2.1 Example

In *RestructuredText* what you type is not what you get. For instance the following snippet

```
Features
________

#. Be awesome
#. Make things faster

Installation
____________

Install **my_project** by running:

.. code-block::

   pip install my_project
```

is rendered to look like the following:

Features

1. Be awesome
2. Make things faster

Installation

Install **my_project** by running:

```
pip install my_project
```

Editing a document is not as straight forward as standard Word Processors, like Libre Office. The idea of using it for documentation is:

2.2.2 Idea

Here some good features of *reStructuredText*:

- automatic formatting
- changes are traceable with git
- auto generated content, i.e. table of content, links
- Speed up writing documentation (once you are familiar with RestructuredText)

2.2.3 Properties

And some things to keep in mind when writing your documentation

- indention is important
- blank lines are very important
- 3 spaces vs. 4 spaces in python
- supports
  - highlighting
  - lists
  - table
  - all sorts of blocks
  - images
  - hyperlinks
  - citations
  - footnotes
  - much more
- can create multiple output formats
  - html
  - LaTEX (pdf)
  - ePub
  - manual pages
  - plain text

For a more detailed introduction on RestructuredText, please take a look at this documentation.

CHAPTER

THREE

DEPLOY DOCUMENTATION

The second session will be about publishing your documentation on the web.

3.1 Sphinx

Sphinx is a tool to build any kind of documenation.

3.1.1 Why shall we use it?

- converts reStructuredText to an output format, i.e. html
- creates links etc. within and amount documents
- supports many customizations
- native python documentation tool widely adapted
  - NumPy
  - SciPy
  - scikit-learn
- change documentation via git

3.1.2 Quick Start

If you want to add documenation to a project, please checkout this guide.

3.1.3 Example

Now you can start writing actual documentation. Each html page corresponds to one `.rst` file. So image we want to document a coffee machine.

Listing 1: index.rst

```
Congrats for buying our new super awesome coffee-machine. Here we provide
more details on the following topics.

.. toctree::
   :maxdepth: 2
```

(continues on next page)

(continued from previous page)

```
   :caption: Table of Content

   safety
   quickstart
```

This table of content will do three things:

- **create links to the heading in `saftety.rst` and `quickstart.rst` right** there up to level 2
- create a navigation menu on the side
- tell sphinx that these documents form a joint assemble

To complete the example we have the two missing files linked in the table of content.

Listing 2: safety.rst

```
Read everything very very carefully.

Safety
======

This coffee machine can only be installed by an electrician. If you do
otherwise, you loose all warranty.


Responsibility (level 1)
========================

We assure you that we don't take any responsibility. Never.

Exception (level 2)
___________________

No exceptions.

Exception (level 3)
*******************

Really?
```

So the last heading `Exception (level 3)` will not appear in the table of content of the index page. All other heading will.

Listing 3: quickstart.rst

```
Instructions
============

Follow the following steps:

#. Take the machine out of its box.
#. PLug the cable into the socket.
#. Switch it on.
```

Afterwards please take a look at the `conf.py` file. This is the place to go to when customizing your documentation.

3.2 Continuous Deployment

3.2.1 Idea

- Once a *Merge Request* is merged everything happens automatically
- typical use-cases are:
  - releasing a new version
  - building the corresponding apps
  - building new documentation

3.2.2 Purpose

- allow developers to be lazy
- automated processes are less error prone
- maintain documentation in Git

3.2.3 Webhook

A Webhook is an Api Endpoint that you can call in order to make a change on a website.

Simple Example

When you post a comment on Reddit, you trigger a webhook as well. In the background some code puts your comment into some reddit database. Afterwards it makes a call to a Reddit webhook that tells it to fetch the newest state from the database.

Elaborate Example: Read the Docs

A similar use-case is to public your documentation to the internet. Therefore we will establish a webhook on Read the Docs. That means our code versioning system (GitLab) will call the webhook when there are changes.

3.3 Read the Docs

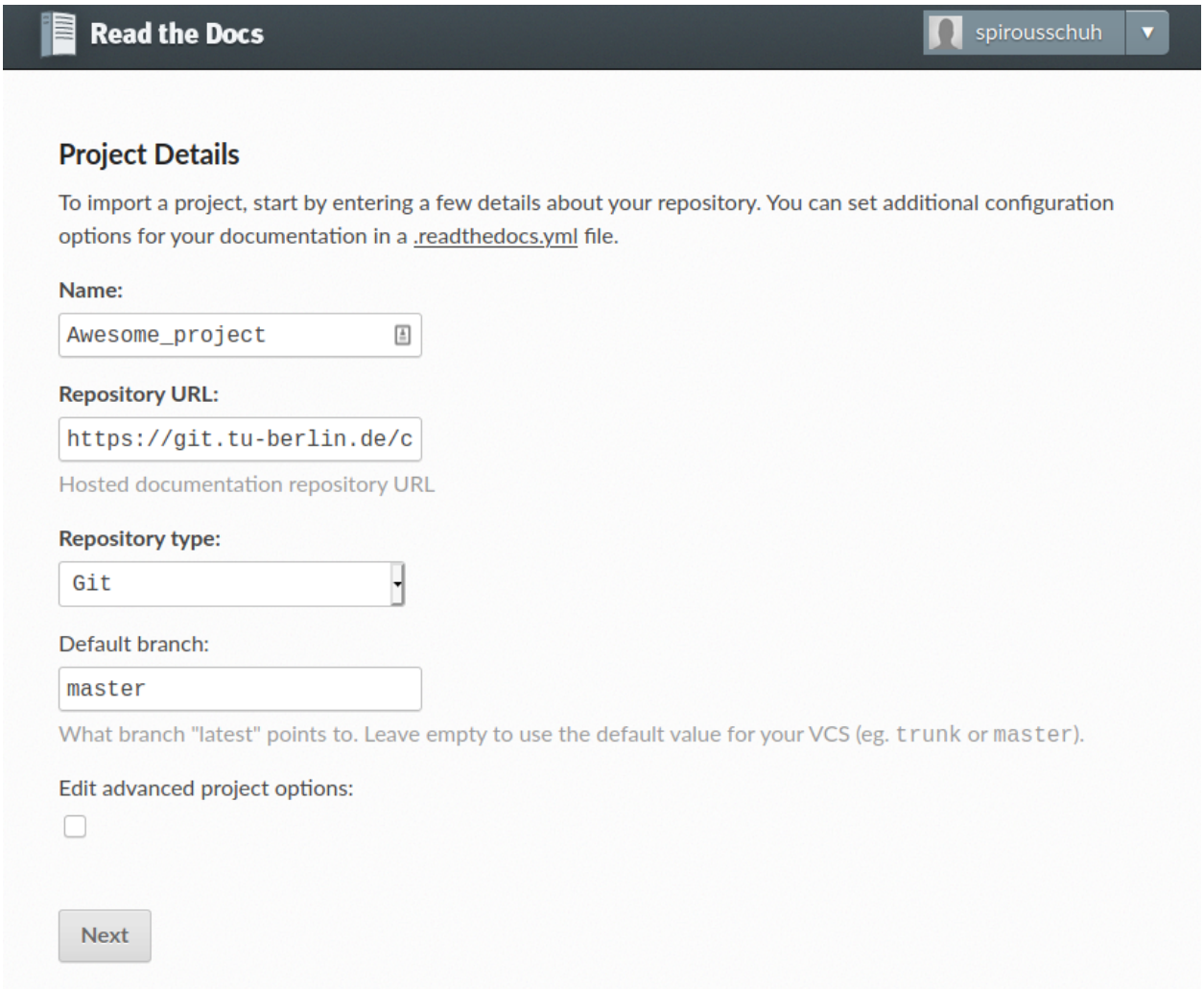
What does Read the Docs do?

- a service that hosts sphinx documentation
- all public repositories are free
- a framework for starting your own documentation server

So here you can find step by step instructions how to publish your documentation using Read the Docs.

3.3.1 Setup Read the Docs

1. Go to https://readthedocs.org/ and create an account.
2. Log in and click on your user-name in the upper right corner.
3. Click on Import a Project
4. Something Manual
5. Then fill the following form



6. Click on **next**.

Now you successfully introduced Read the Docs to your repository. On the next page you see two important things.

On the one hand you can try to build your documentation and on the other hand we did not establish a webhook so far. Let us start with the build.

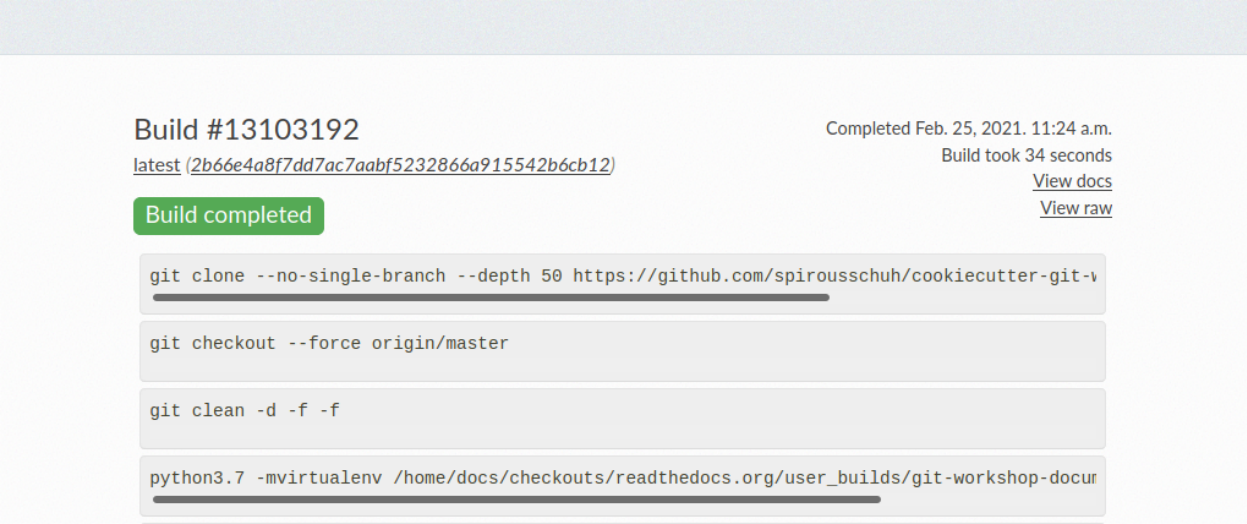
3.3.2 Build Documentation

For the build your repository needs to be public. In case it is already public please continue here.

1. Open your repository `https://git.tu-berlin.de/your/repo/url` in the browser
2. In the bottom left corner click on settings
3. Scroll down to the section *Visibility, project features, permissions*
4. Click on the **Expand** button
5. Underneath the heading *Project Visibility* use the drop down menue to choose *Public*
6. Scroll down and click save changes

Now your repository can be seen by everybody in the internet, in particular by the Read the Docs service, so we can start building the documentation.

1. Open https://readthedocs.org/ in your browser and log in
2. Click on your user-name in the upper right corner.
3. Click on your project name
4. Click on the **Build Version** button
5. Wait for the build to succeed. It should look like this



1. On the right hand side please click on the button **View docs**.

3.3.3 Add Webhook

Now we want our documentation to be updated automatically, once a Merge Request is merged. Therefore we will add a webhook to the repository.

1. Open https://readthedocs.org/ in your browser and log in
2. Click on your user-name in the upper right corner.
3. Click on your project name
4. Click on the **Admin** button
5. In the navigation on the left hand side click on **Integrations**

6. Click on the button **Add integration**
7. In the pull down menue select *GitLab incoming webhook* and click on the **Add integration** button
8. Now right click on the link that you see and that starts with `readthedocs.org/api/v2/webhook/...` to copy it
9. Go back to your repository `https://git.tu-berlin.de/your/repo/url`
10. Click on **Settings>Webhooks**
11. Copy the webhook URL to the *URL* field
12. Scroll down and click on *Add webhook*.

CHAPTER

FOUR

DOCSTRING

And the third session will be about documenting more detailed code interfaces.

4.1 Doc-Strings

4.1.1 Types of Docstrings

There is different entities that posses docstrings. Take a look at the example from Wikipedia:

```
"""The module's docstring"""

class MyClass:
    """The class's docstring"""

     def my_method(self):
         """The method's docstring"""
         pass

def my_function():
    """The function's docstring"""
    pass
```

Most entities have a docstring. You can check it via `my_entity.__doc__`.

4.1.2 Detailed Docstring

You can use it to explain the user how to use your function in more detail:

```
def addition(arg1, arg2):
    """
    This functions adds the first and the second argument.

    :param arg1: the first summand
    :type arg1:  float
    :param arg2: the second and last summand
    :type arg2:  flaot
    :return:     the sum of both summands
    :rtype:      float
    """
    return arg1 + arg2
```

Why shall we document the function right in the code?

- documentation is close to the code
  - explanation right at hand
  - once you change the code you can change the docu right at the same place
- We can still include the docstring into our main documentation

4.1.3 Autodoc

When you stick to the convention above to describe the function arguments, what the function returns as well as the typing, you can use automatically generated documentation.

First thing you need to do is to change the `conf.py` file, to tell Sphinx that it should use *autodoc*.

Therefore go to `conf.py` and append the following string to the *extensions* list.

```
extensions = ['sphinx.ext.autodoc']
```

Now you can use the autofunction feature by adding the following block to your documentation.

```
.. autofunction:: .adding_numbers.addition.addition
```

Please note that `adding_numbers.addition.addition` refers to the function in the same way you would import it. You can see the result here:

Please note that the convention we used here is the one from *reStructuredText*. There is other conventions from Google as well as from Numpy that are fairly common.

CHAPTER

FIVE

EXERCISES

Here you can find some tasks to practise the ideas introduced above.

5.1 Task 0: Create a new repository

Similar to last two times we want to create a new repository that we use for this workshop. Please note that we want to create a repository in the group kiwi-git-workshops.

For step by step instructions on how to create a repository, you can take a look at the creating a *Creating a Repository* page.

5.2 Task 1: Readme

Please create a seperate branch for each task and create a Merge Request every time. You can find detailed instructions on the *Git Workflow* page.

TODO (Task 1): Please add a small Readme in the format reStructuredText. So you should change the file `README.rst`.

It should contain:

- the repository's purpose
- an installation guide
- a small snippet how to use the command line interfact

Please keep in mind:

- indention is 3 spaces
- blank lines are important to seperate blocks from each other

After merging your Merge Request you should see your new `README.rst` at the landing page of your project `https://git.tu-berlin.de/kiwi-git-workshops/your_project`.

5.3 Task 2: Creating Sphinx Documentation

Please create a seperate branch for each of the sub-tasks and create a Merge Request every time. You can find detailed instructions on the *Git Workflow* page.

5.3.1 Setup Sphinx

Please follow this guide in order to create a basic documentation.

5.3.2 Landing Page

Now we want to replace the content of `index.rst` with what we already used for the `Readme.rst`.

You can quickly build your documentation locally to check if everything is rendered accordingly.

```
cd /your/project/docs
make html

# open the html site in a browser of your choice
firefox _build/html/index.html
```

5.4 Task 3: Read the Docs

Now we want to publish the documentation, the one file from task 2, to the internet. Therefore please follow the steps described in *Read the Docs* and apply them to the current repository.

5.5 Task 4: Docstrings

In the last workshop we dealt with a function called `invert_image`. You can find it in this package as well

```
your_package_name.processing.invert_image
```

Please write a docstring for that function. Here you can find more detailed instructions on *Detailed Docstring*.

Once you added a Docstring, please add another file that describes the process of inverting an image to the documentation. Then use the autofunction feature to include the Docstring into the documentation.

If you encounter an error like

```
WARNING: autodoc: failed to import function 'some.module'
```

make sure that you installed the package into your current enviroment.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search