# Семинар 1. Git, github, CI/CD.

Основная задача на сегодня: понять, что Git это совокупность файловой системы, хеширования, DAG и несколько алгоритмов взаимодействия с ними.

1. Создайте пустой репозиторий **tmp-repo** на своём аккаунте с помощью Github UI – **ничего не добавляйте в него**. Склонируйте его к себе на локальную машину.
2. Посмотрите на внутренние файлы папки **.git**: есть ли внутри них что-либо (особенно в **.git/objects**)? Посмотрите, что в себе содержит файл **.git/HEAD**. В этом файле хранится строка **refs/heads/main** – адрес файла, в котором лежит хэш наиболее свежего коммита в ветке, на которой мы сейчас находимся:

   ```
   ~/tmp_repo$ cat .git/HEAD
   ref: refs/heads/main
   ```
3. В репозитории нет ни одного коммита, как, впрочем, и ни одного файла. Запишем в файл **first_file.txt** цифру 1:

   ```
   echo "1" > first_file.txt
   ```
4. Сейчас об этом файле знает только ФС вашего компьютера, однако **git** уже в курсе, что вы добавили файл, но не подготовили его к стейджу в коммит (не добавили в индекс). Понять это можно с помощью просмотра вывода команды **git status**. Кроме того, сама директория **.git** не изменилась.

   ```
   ~/tmp-repo$ git status
   On branch main

   No commits yet

   Untracked files:
     (use "git add <file>..." to include in what will be committed)
       first_file.txt

   nothing added to commit but untracked files present (use "git add" to track)
   ```
5. Добавим в индекс наш коммит с помощью **git add first_file.txt**. Проверим, что файл попал в отслеживание **git** с помощью **git status**. Обратим внимание на то, что теперь лежит в **.git**. Разрослась папка **.git/objects**, появился непустой бинарный файл **.git/index**:

   ```
   .git/objects/d0:
   total 4,0K
   -r--r--r-- 1 17 jan 16 20:52 0491fd7e5bb6fa28c517a0bb32b8b506539d4d

   -rw-rw-r-- 1 112 jan 16 20:52 .git/index
   ```

   В объектах появился слепок файла **first_file.txt**. Показать, что это именно он мы можем с помощью команды **git hash-object first_file.txt**.
6. Проверим, что лежит в индексе, то есть, за какими файлами следит **git**. Сделать это можно с помощью команды **git ls-files -s** (для полного вывода).
7. Подвяжем BLOB файла к tree-объекту, который будет отправлен в коммит.
   Обычно это автоматически выполняется за вас когда вы формируете коммит

командой **`git commit`**, однако сейчас, т.к. мы ничего не сломаем, мы можем руками подвязать BLOB к tree командой **`git write-tree`**.

```
git write-tree
9782c59213a234b40728f75846f0148b9e4f5d5e
```

Обращаем внимание: теперь в репозитории два объекта из модели **`git`**: blob и дерево. Это отображается и в том, что теперь появилось в **`.git/objects`**: запакованный в packfile, там теперь хранится также и файл дерева. Посмотрите, что теперь лежит внутри файла tree с помощью **`git cat-file -p 97….5e`**.

8. Продолжим низкоуровнево взаимодействовать с **`git`**. Теперь, после того, как мы создали файл tree 97 из индекса, мы можем сделать коммит. Однако, мы пойдём сложным путём и не будем писать **`git commit`**. Для этого, как мы обсуждали на лекции, нам потребуется сделать ещё 2 действия: создать объект-коммита, вручную его хешируя, и закоммитить дерево с этим объектом.

   Получим объект-хеш дерева, передадим в **`git commit-tree`** сообщение о коммите и перекинем ссылку. Вот как это выглядит (важное уточнение: на этом этапе наши с вашими хешами разойдутся, т.к. в информацию о коммите хешируется также и информация об авторе, **`user.name`** и **`user.email`**):

   ```
   # результат git write-tree не будет меняться если не менялась ФС
   tree=$(git write-tree)

   # пишем сообщение коммита и передаём его на stdin в git commit-tree
   commit=$(echo "Commit Message" | git commit-tree $tree)

   # обновляем ссылку HEAD на новый коммит
   git update-ref refs/heads/main $commit
   ```

   Первые две строки понятны: мы получаем хеш объекта-дерева, которое ссылается на blob файла **`first_file.txt`**, а затем создаём объект коммита. Зачем нужна третья строка?

   Благодаря ней мы двигаемся по истории разработки вперёд. Теперь в дереве коммитов появился первый коммит (у меня его хеш: **`7b8262d51f3434712e902027cd5c2107accb528e`**), и нам надо перевесить на него ссылку **`HEAD`**, чтобы не потеряться (такая ситуация, когда мы вручную создали коммит, создаёт detached **`HEAD`** state – подумайте, почему).

   Посмотрите выдачу **`git cat-file -p <commit_hash>`**.

9. С первым коммитом в репозитории и в .git многое изменилось. Во-первых, теперь файл **`.git/refs/heads/main`** содержит хеш этого коммита (мы перевесили ссылку ранее). В папке **`.git/logs/refs/heads`** есть файл **`HEAD`** и папка **`refs/heads/main`** – обе они нужны для сохранения логов и просмотра истории коммитов. Просмотрите историю с помощью **`git log`**. Что в ней находится? Что изменилось в **`.git/objects`**?
10. Посмотрите внимательно на выдачу git status, она должна была измениться. Теперь там появилась следующая строка:

    ```
    Your branch is based on 'origin/main', but the upstream is gone.
      (use "git branch --unset-upstream" to fixup)
    ```

Так, конечно, быть не должно. **Git** ругается не потому, что мы вручную собрали коммит. Проблема заключается в другом – наша локальная ветка **.git/refs/heads/main** существует (в ней 1 коммит), в то время как на сервере Github её нет. Для того, чтобы в дальнейшем не волноваться об этом, выполним пуш локальной ветки в удалённый репозиторий и сделаем **fetch**:

```
git push -u origin main
git fetch
```

Верхнеуровневая команда **git fetch** ходит в удалённый репозиторий, считывает оттуда информацию о ветках и кладёт её локально. Теперь в **git status** нет проблем отсутствия веток. Посмотрите, появилось ли ещё что-то в файлах **.git**. Обратите внимание, что есть файл **.git/info/exclude**, работающий как **.gitignore**, но в репо не попадающий. Локальные ignore-файлы, которые не коммитятся, лежат в **.git/info/exclude**.

11. Проведите те же манипуляции с ещё одним файлом. Напишите питоновский скрипт **greet_git.py**, который печатает строку **“Hello, USERNAME”**, где **USERNAME** – имя пользователя в вашей системе (поможет модуль **getpass** и **getpass.getuser()**). Вручную посмотрите на файл, добавьте его в индекс, соберите дерево и отправьте коммит, обновив ссылку **HEAD**. **ВАЖНО! Отправьте коммит другой командой:**

    ```
    commit=$(echo "Commit Message" | git commit-tree $tree -p $(git rev-parse HEAD))
    ```

    Как вы думаете, почему команда коммита стала визуально “сложнее”? Что будет, если не указывать **-p**?
12. Повторите то же, заменив в файле **“Hello”** на **“Greetings”** и сделав коммит через **git commit**. Следите за тем, что меняется в папке **.git/objects** (пользуйтесь **cat-file** для blob и tree, **ls-files** и др.). Файлы blob неизменяемы, поэтому в репозитории будет храниться как новый, так и старый blob.
13. Переименуйте **first_file.txt** в **new_file.txt**. Какими командами вы это сделали? Опции: **mv** + **git rm** + **git add** ИЛИ **git mv**. Сделайте **HEAD** коммиту тег **“v0.1”** (без кавычек)ю
14. Рассмотрите выдачу **git log --graph --all**. Соответствует ли она ожиданиям? Если да, перейдём в коммит на шаг назад, отсоединив **HEAD**. Сделайте это с помощью команды **git checkout** и относительной по **HEAD** адресации коммитов (**“^”, “@{1}”**). Внимательно рассмотрите выдачу лога в **git**. Когда вы перешли на коммит пораньше, **git** проходит по ссылке коммита на дерево, а затем для каждого файла в дереве делает распаковку BLOB и запись в файл. Создайте ветку **test_branch** с помощью **git branch** и переключитесь на неё с помощью **git checkout**. Сделайте в этой ветке новый файл secret.json, в который запишите случайную пару ключ-значение. Сделайте коммит и посмотрите на **git log --graph --all** ещё раз.
15. Отправьте ветку на сервер с помощью **git push -u origin test_branch**. Здесь **origin** – псевдоним-ссылка на удалённый репозиторий, которая хранится в **.git/config**.

16. Переключитесь обратно на **main** и проведите **git merge test_branch**. Получилось ли? Почему? Посмотрите на **git log --oneline --graph --all**. Что происходит в дереве, когда запускается алгоритм слияния?
17. Переключитесь на **test_branch** и внесите изменения в txt-файл (как он называется до расширения и почему?). Попробуйте снова сделать **merge** из **main**. Всё ли **git** понимает? Вот как примерно должен выглядеть граф:

```
*   224e262 (HEAD -> main) Merge branch 'test_branch'
|\
| * 785e0eb (origin/test_branch, test_branch) modify first_file
* | 3e29180 Merge test_branch to main
|\|
| * e41576c Add branch-specific file
* | ee4981b (tag: v0.1) moved first to new
|/
* d26dfe6 Fix greeter
* 7e82524 (origin/main) add greeter
```

18. Запушьте изменения в удалённый репозиторий. Внутри Github UI после измените какой-нибудь файл (**new_file.txt**, например) и сделайте коммит через Github UI. Измените его же локально. Попробуйте закоммитить и запушить. Что происходит и почему? Как это решается?
19. Создайте Github-Actions workflow для автоматического тестирования кода в **greet_git.py**, активируемый во время **pull-request** и **push** в репозиторий в ветку **main**. Вам понадобится шаблон:

```yaml
name: Python Greet Test

on:
  ???

jobs:
  test-and-lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.x'
      - name: Install pylint
        run: |
          python -m pip install --upgrade pip
          ???
      - name: Run pylint
        run: |
          pylint greet_git.py --disable=???

      - name: Run greet script
        run: |
          ???
          ???
```