

Лабораторная работа №1 СТРweb-пр
на тему «Системы контроля версий (Git)»

Цель: приобрести навыки работы с системой контроля версий Git, ознакомиться с особенностями платформы GitHub.

Задачи:

- 1 Создать репозиторий на платформе GitHub.
- 2 Установить приложение Git Bash.
- 3 Воспользоваться основными Git-командами и некоторыми возможностями платформы GitHub.

СОДЕРЖАНИЕ

1 Установка приложения Git Bash	3
2 Теория и задания.....	5
Открытие консоли Git Bash	5
Часть №1	6
git init (инициализация локального репозитория)	6
git status (статус файлов в репозитории)	6
git add (добавление файла в индексирование)	7
git commit (сохранение изменений)	8
git log (просмотр истории коммитов)	10
git restore (откат изменений).....	10
Заключение №1	10
Задание №1	11
Часть №2.....	12
git branch, git checkout (работа с ветками).....	12
Конфликты при слиянии.....	16
git commit –amend (изменение коммитов)	16
git reset, git revert (отмена коммитов)	17
Заключение №2.....	19
Задание №2.....	20
Часть №3	21
git remote (работа с удаленным репозиторием).....	21
Создание репозитория на платформе GitHub	21
git clone, git fetch, git pull (извлечение данных из удаленных репозиториях).....	22
git push (отправка изменений)	24
Fork.....	24
Заключение №3	25
Задание №3	26

Предисловие

GitHub – это веб-сервис, который использует Git для хостинга репозитория. Он предоставляет интерфейс для управления проектами и совместной разработки.

Git – это система контроля версий, которая позволяет разработчикам отслеживать изменения в коде, управлять версиями и работать над проектами в команде. Он установлен локально на компьютере разработчика и предоставляет команды для работы с репозиториями.

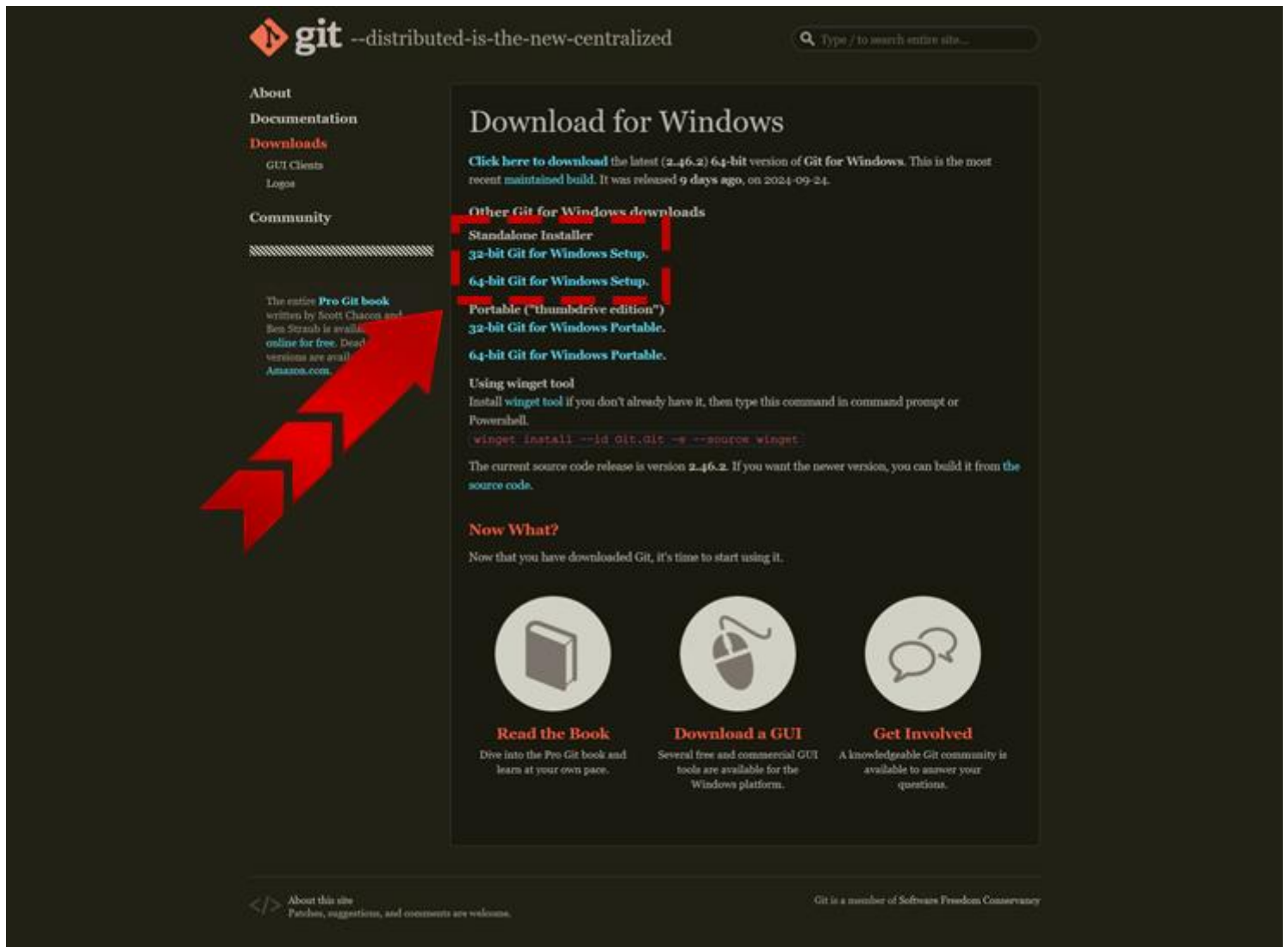
Вкратце: Git – это инструмент для контроля версий, а GitHub – платформа для совместной работы с использованием Git.

1 УСТАНОВКА ПРИЛОЖЕНИЯ GIT BASH

Задача:

Перейдите по ссылке <https://git-scm.com/downloads/win>. В зависимости от разрядности Вашей системы, скачайте установщик и установите это приложение. Перезагрузите компьютер. Это приложение установит Git и Git Bash на ваш компьютер.

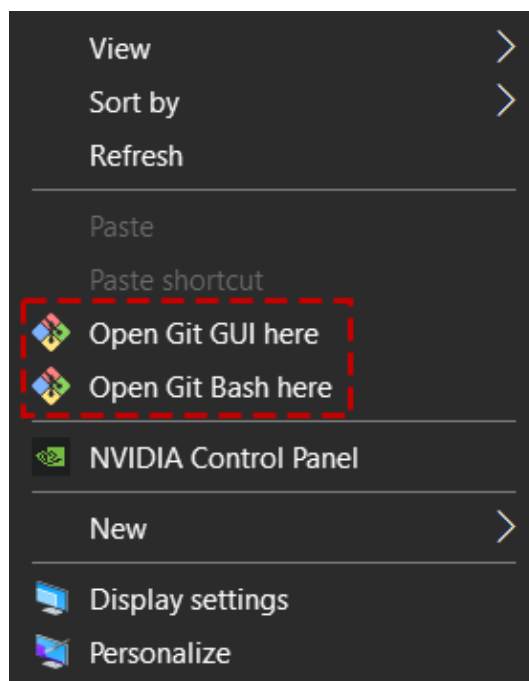
Перезагрузка нужна, чтобы установилась переменная среды. Это позволит управлять git через обычную командную строку (Win+R -> cmd). Без перезагрузки с гитом можно будет работать только через Git Bash.



После перезагрузки компьютера нажмите ПКМ по свободному месту рабочего стола. Выберите пункт Open Git Bash here для открытия окна-эмулятора командной строки. Это окно предоставляет интерфейс для работы с Git и возможность выполнения Unix-команд. Стандартные Unix-команды, которые пригодятся при выполнении лабораторной работы:

- clear – очистка консоли;
- cd <path> – перемещение по каталогам;

- `cd ..` – перемещение назад, на уровень выше;
- `ls` – список файлов и каталогов в текущем каталоге;
- `pwd` – показать текущий рабочий каталог.



Параметры `user.name` и `user.email` в Git используются для идентификации авторов вносимых изменений. `user.name` представляет имя пользователя, а `user.email` – адрес электронной почты соответственно.

Задача:

В соответствии с приведённым примером, изменить параметры для идентификации автора изменений.

```
Usevalad@DESKTOP-5EKA060 MINGW64 ~/Desktop
$ git config --global user.name "CTPweb-pr"

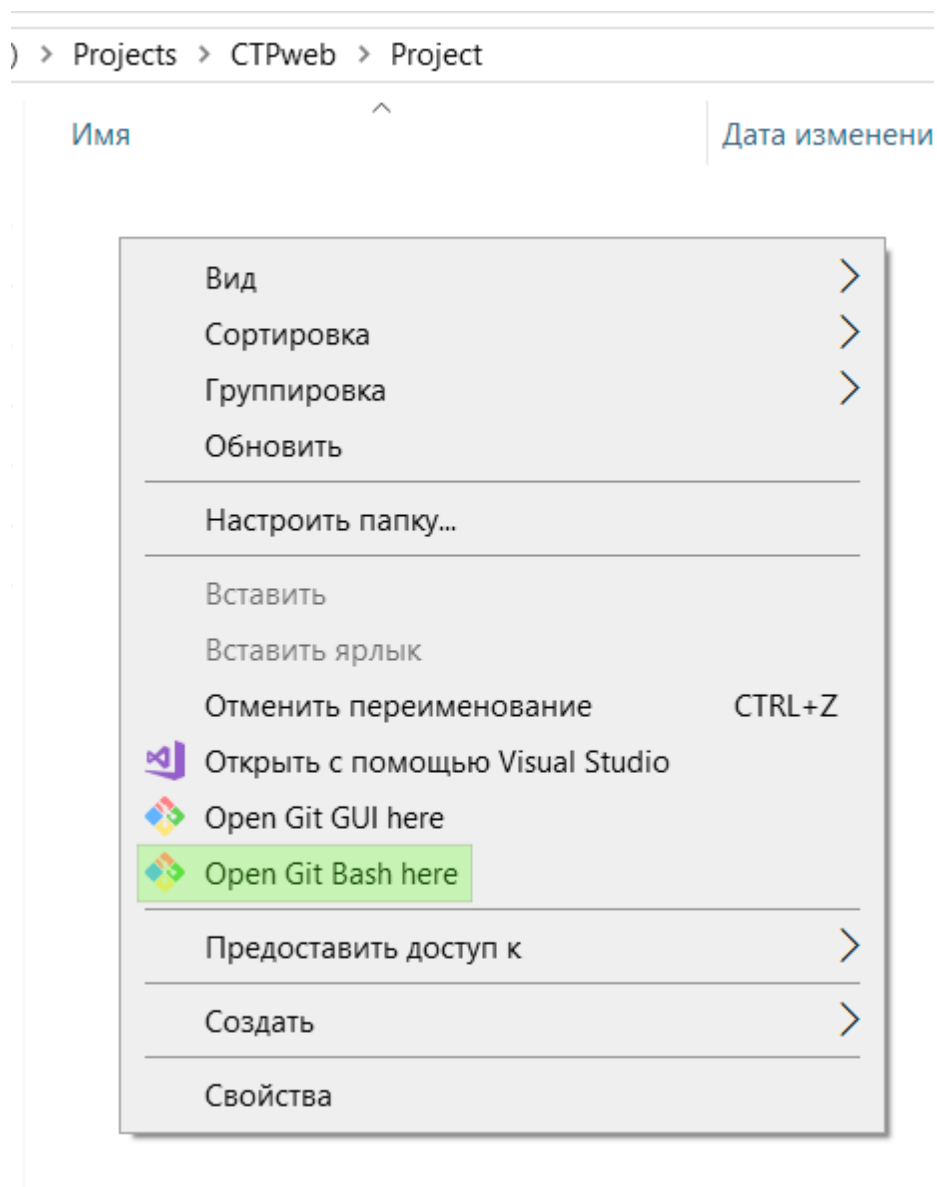
Usevalad@DESKTOP-5EKA060 MINGW64 ~/Desktop
$ git config --global user.email example_mail@gmail.com

Usevalad@DESKTOP-5EKA060 MINGW64 ~/Desktop
```

2 ТЕОРИЯ И ЗАДАНИЯ

ОТКРЫТИЕ КОНСОЛИ GIT BASH

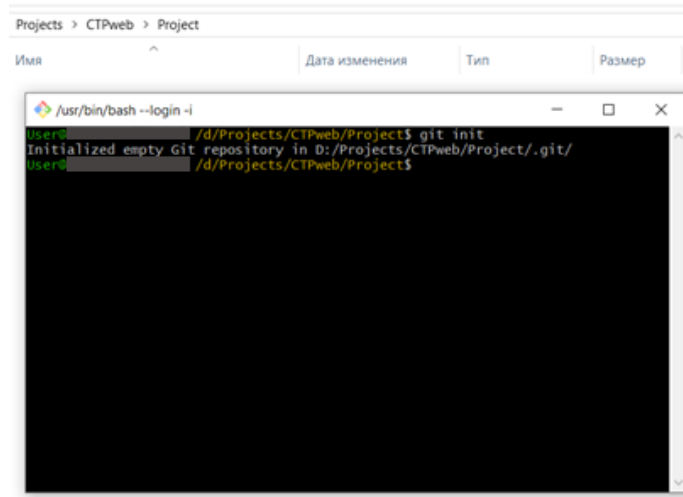
Для начала слежения за проектом необходимо перейти в папку проекта и открыть Git Bash с помощью контекстного меню.



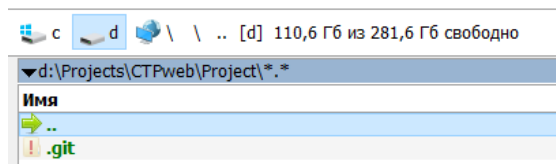
ЧАСТЬ №1

git init (инициализация локального репозитория)

В открывшейся консоли используем команду
\$ git init

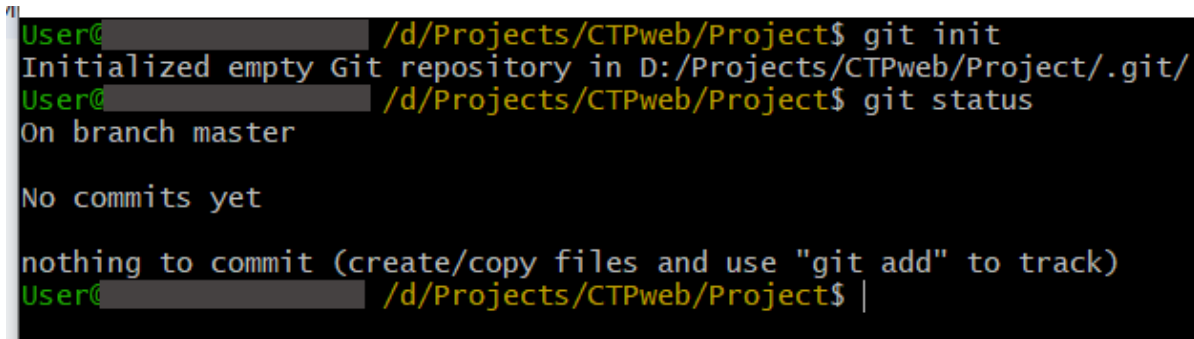


Эта команда инициализирует новый репозиторий GIT и начинает отслеживание существующего каталога. В текущей директории создаётся новая скрытая поддиректория с именем .git, содержащая все необходимые файлы репозитория – основу Git-репозитория.



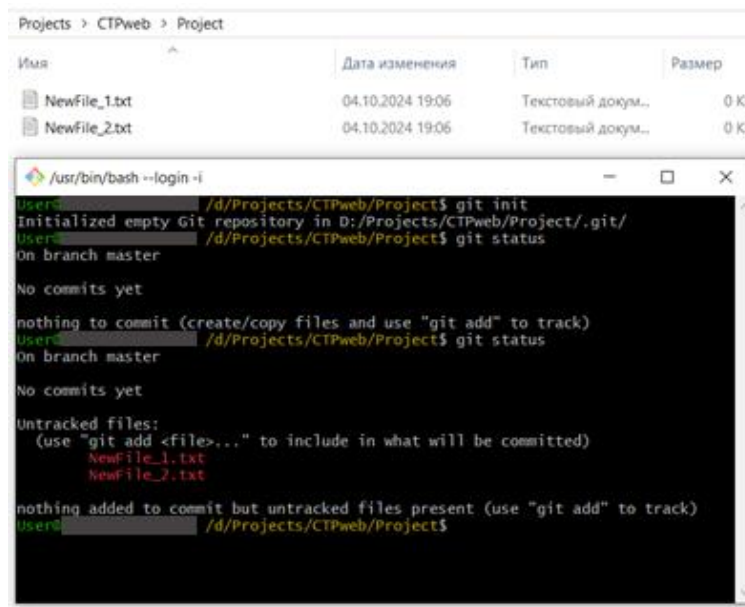
git status (статус файлов в репозитории)

Далее выполните
\$ git status



В данном случае Git не обнаружил измененный отслеживаемых файлов. Также Git не обнаружил не отслеживаемых файлов.

Добавим пару файлов и выполним команду ещё раз.



```
Projects > CTPweb > Project

Имя          Дата изменения  Тип          Размер
NewFile_1.txt 04.10.2024 19:06 Текстовый докум.. 0 КБ
NewFile_2.txt 04.10.2024 19:06 Текстовый докум.. 0 КБ

/usr/bin/bash --login -i
User@ /d/Projects/CTPweb/Project$ git init
Initialized empty Git repository in D:/Projects/CTPweb/Project/.git/
User@ /d/Projects/CTPweb/Project$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
User@ /d/Projects/CTPweb/Project$ git status
On branch master

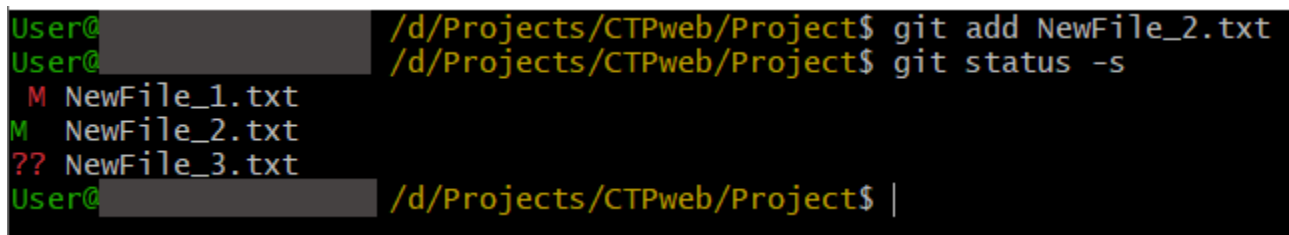
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    NewFile_1.txt
    NewFile_2.txt

nothing added to commit but untracked files present (use "git add" to track)
User@ /d/Projects/CTPweb/Project$
```

Как мы можем заметить появился блок «Untracked files». Здесь красным выделены файлы одноименного статуса. «Untracked files», по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (коммите). Git не станет добавлять такой файл в ваши коммиты.

В Git существует флаг, позволяющий получить сведения в более компактной форме. Запустив команду `git status -s` или `git status --short`, вы получите упрощенный вариант вывода.



```
User@ /d/Projects/CTPweb/Project$ git add NewFile_2.txt
User@ /d/Projects/CTPweb/Project$ git status -s
 M NewFile_1.txt
 M NewFile_2.txt
 ?? NewFile_3.txt
User@ /d/Projects/CTPweb/Project$ |
```

В этой форме рядом с именами стоят специальные знаки.

- Знак ?? соответствует именам новых не отслеживаемых файлов, новые файлы;
- Знак A указывает на файлы, добавленные в область предварительной подготовки;
- Знак M указывает на модифицированные файлы.

git add (добавление файла в индексирование)

Для фиксации изменений необходимо начать слежение за файлами. Для этого воспользуемся командой

\$ git add <Название файла>

```
User@ /d/Projects/CTPweb/Project$ git add NewFile_1.txt
User@ /d/Projects/CTPweb/Project$ git add NewFile_2.txt
User@ /d/Projects/CTPweb/Project$ git status

On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   NewFile_1.txt
        new file:   NewFile_2.txt

User@ /d/Projects/CTPweb/Project$ |
```

Используя git add, мы добавляем файлы в индекс. Мы можем как указывать файлы по одному, так и добавлять множество файлов используя различные формы команды:

\$ git add . – добавляет в индекс все файлы текущей директории

\$ git add *.txt – добавляет в индекс все файлы с расширением .txt в текущей директории

\$ git add **/*.txt – добавляет в индекс все файлы с расширением .txt во всех подкаталогах

Файлы теперь имеют статус «Tracked» и выделены зелёным. Такие файлы являются отслеживаемыми и будут добавлены в коммиты.

git commit (сохранение изменений)

Используем

\$ git commit

Добавим параметр -m, чтобы написать комментарий прямо в командной строке. -m – это параметр, модифицирующий команду. Список подобных параметров обширен, с их списком можно ознакомиться с помощью параметра -h.


```

User@ /d/Projects/CTPweb/Project$ git commit -h
usage: git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
                  [--dry-run] [(-c | -C | --squash) <commit> | --fixup [(amend|reword):]<
commit>]]
                  [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
                  [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
                  [--date=<date>] [--cleanup=<mode>] [--[no-]status]
                  [-i | -o] [--pathspec-from-file=<file>] [--pathspec-file-nul]
                  [--trailer <token>[(=|:)<value>]]... [-S[<keyid>]]
                  [--] [<paths>...]

-q, --[no-]quiet      suppress summary after successful commit
-v, --[no-]verbose    show diff in commit message template

Commit message options
-F, --[no-]file <file>
                        read message from file
--[no-]author <author>
                        override author for commit
--[no-]date <date>
                        override date for commit
-m, --[no-]message <message>
                        commit message
-c, --[no-]reedit-message <commit>
                        reuse and edit message from specified commit
-C, --[no-]reuse-message <commit>
                        reuse message from specified commit
--[no-]fixup [(amend|reword):]<commit>
                        use autosquash formatted message to fixup or amend/reword speci
fied commit
--[no-]squash <commit>
                        use autosquash formatted message to squash specified commit
--[no-]reset-author
                        the commit is authored by me now (used with -C/-c/--amend)
--trailer <trailer>
                        add custom trailer(s)
-s, --[no-]signoff    add a Signed-off-by trailer
-t, --[no-]template <file>
                        use specified template file
-e, --[no-]edit        force edit of commit
--[no-]cleanup <mode>
                        how to strip spaces and #comments from message
--[no-]status          include status in commit message template
-S, --[no-]pgp-sign[=<key-id>]
                        GPG sign commit

Commit contents options
-a, --[no-]all        commit all changed files
-i, --[no-]include    add specified files to index for commit
--[no-]interactive    interactively add files
-p, --[no-]patch      interactively add changes
-o, --[no-]only       commit only specified files
-n, --no-verify       bypass pre-commit and commit-msg hooks

```

Изменим отслеживаемый файл.

Имя	Дата изменения	Тип	Размер
NewFile_1.txt	04.10.2024 19:25	Текстовый докум...	1 КБ
NewFile_2.txt	04.10.2024 19:06	Текстовый докум...	0 КБ

NewFile_1.txt - Блокнот

Файл Правка Формат Вид Справка

Changes

Используем вновь
\$ git status

```

Projects > CTPweb > Project

Имя          Дата изменения  Тип          Размер
NewFile_1.txt 04.10.2024 19:25 Текстовый докум... 1 КБ
NewFile_2.txt 04.10.2024 19:06 Текстовый докум... 0 КБ

/usr/bin/bash --login -i

No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file:   NewFile_1.txt
new file:   NewFile_2.txt

User@ /d/Projects/CTPweb/Project$ git status
On branch master

No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file:   NewFile_1.txt
new file:   NewFile_2.txt

Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified:   NewFile_1.txt

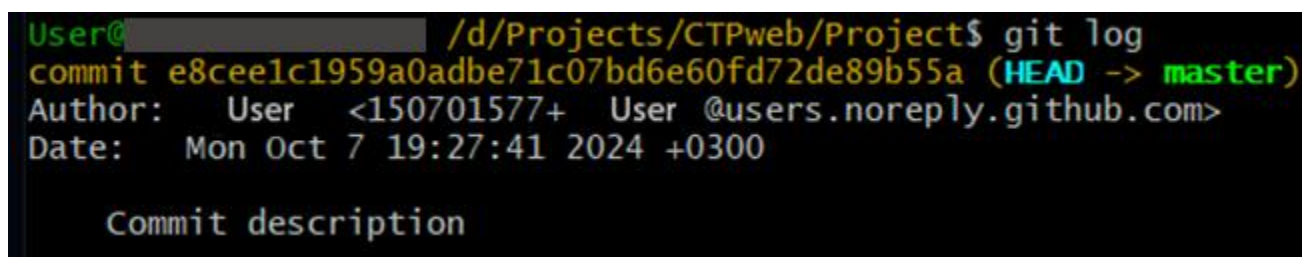
User@ /d/Projects/CTPweb/Project$

```

Git индексирует файл в том состоянии, в котором он пребывал на момент выполнения команды `git add`. Если сейчас зафиксировать изменения, в коммит войдет версия, появившаяся после последнего запуска команды `git add`, а не версия, находившаяся в рабочей папке при запуске команды `git commit`. Редактирование файла после выполнения команды `git add` требует повторного запуска этой команды для индексирования самой последней версии файла.

git log (просмотр истории коммитов)

После сохранения нескольких версий файлов вы, скорее всего, захотите взглянуть на то, что было сделано ранее. Базовым инструментом в данном случае является команда `$ git log`



```
User@ /d/Projects/CTPweb/Project$ git log
commit e8cee1c1959a0adbe71c07bd6e60fd72de89b55a (HEAD -> master)
Author: User <150701577+ User @users.noreply.github.com>
Date: Mon Oct 7 19:27:41 2024 +0300

Commit description
```

Команда `git log` выводит в обратном хронологическом порядке список сохраненных в данный репозиторий версий. То есть первыми показываются самые свежие коммиты. Рядом с каждым коммитом указывается его контрольная сумма SHA-1, имя и электронная почта автора, дата создания и сообщение о фиксации.

git restore (откат изменений)

В процессе работы с Git может возникнуть необходимость откатить изменения, сделанные в файлах. Файл можно вернуть в состояние, в котором он находился до последней фиксации. Для этого используется команда `$ git restore <имя_файла>`

После использования этой команды любые изменения соответствующего файла исчезнут, поверх него копируется содержимое другого файла, поэтому при использовании этой команды следует быть осторожнее.

Заключение №1

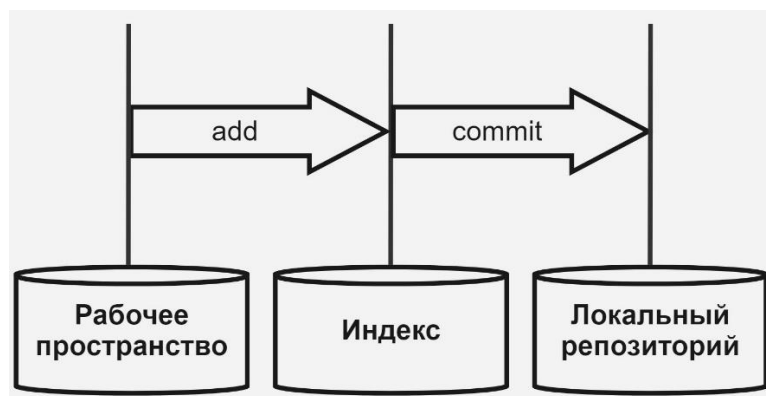
Таким образом, обобщая вышесказанное, получаем:

- 1 Для начала работы с git в Git Bash необходимо открыть необходимую директорию и создать репозиторий с помощью команды `$ git init`

- 2 Для отслеживания изменений файлов, их необходимо поместить в индекс с помощью команды
\$ git add
- 3 Для получения информации о текущем состоянии файлов используется команда
\$ git status
- 4 Для фиксации изменений файлов, добавленных в индекс, создается коммит – «снимок» состояния файлов в репозитории на определённый момент времени. Создается коммит с помощью команды
\$ git commit
- 5 Для просмотра истории версий, используется команда
\$ git log
- 6 Для отката изменений файла до состояния последней фиксации используется
\$ git restore <имя_файла>

Как и другие команды, команда `git commit` имеет параметры. Параметр `-m` позволяет добавить комментарий к коммиту.

Рисунок отображает процесс работы с Git: изменения из рабочего пространства добавляются в индекс (`add`), затем фиксируются в локальном репозитории (`commit`).



Задание №1

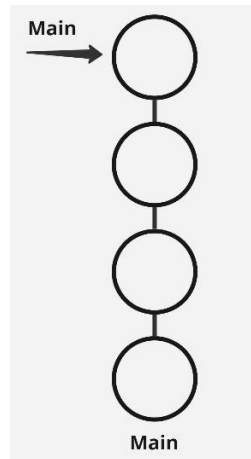
Для успешного выполнения задания №1 требуется:

- 1 Создайте новую директорию.
- 2 Создайте в ней новый репозиторий.
- 3 Добавьте в директорию несколько файлов.
- 4 Проиндексируйте файлы.
- 5 Создайте коммит с комментарием.
- 6 Измените файлы.
- 7 Проиндексируйте измененные файлы.
- 8 Создайте второй коммит.
- 9 Просмотрите историю коммитов

ЧАСТЬ №2

git branch, git checkout (работа с ветками)

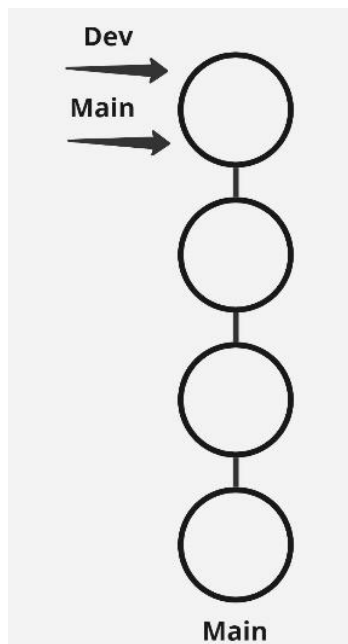
Что такое ветка или бранч (branch)? Ветка (branch) в Git – это отдельная линия разработки, которая позволяет изолировать изменения в проекте. Она представляет собой независимую копию файлов, где разработчики могут вносить изменения, добавлять новые функции или исправлять ошибки, не затрагивая основной код (или другие файлы) в главной ветке. Строго говоря это указатель на определённый коммит в истории репозитория. Для более простого восприятия используем визуальную модель.



Кругами на картинке изображены коммиты, стрелкой отображается текущее положение указателя. На данный момент на картинке только одна ветка – Main.

Добавим новую ветку. Для этого воспользуемся командой

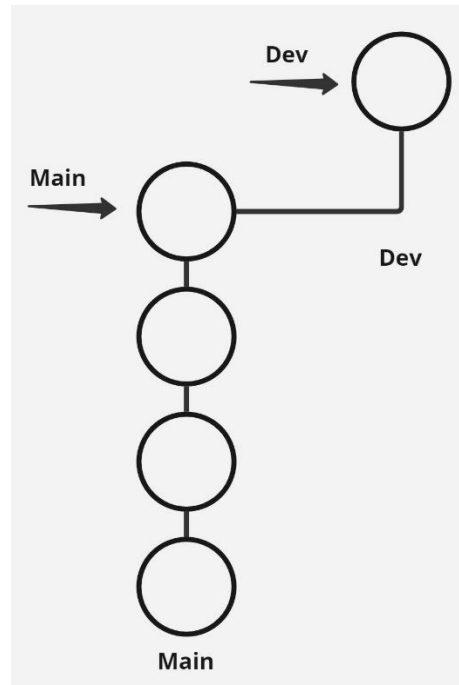
`$ git branch <имя_ветки>`



С помощью `git branch` мы создали новую ветку `Dev`, однако работать мы продолжаем с `Main`. Чтобы начать работу в новой ветке, необходимо переключиться на неё. Это можно сделать с помощью команды

```
$ git checkout <имя_ветки>
```

Теперь, когда мы выполним коммит, схема будет выглядеть следующим образом:



С помощью `git checkout` мы можем свободно переключаться с одной ветки на другую, причём изменения в файлах одной ветки не повлияют на файлы в другой. Таким образом мы можем вернуться на ветку `Main`, создать несколько коммитов, после чего вернуться к ветке `Dev`. Такой подход позволяет вести параллельную разработку. Одни разработчики работают с одной веткой, другие – с другой, и их изменения не мешают друг другу.

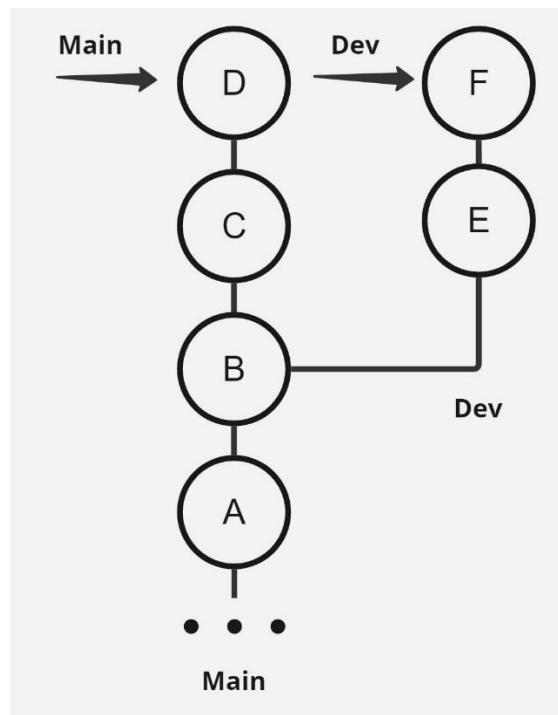
Для создания ветки мы также можем использовать команду

```
$ git checkout -b <имя_ветки>
```

Если мы используем эту команду, то мы сразу переключаемся на новую ветку.

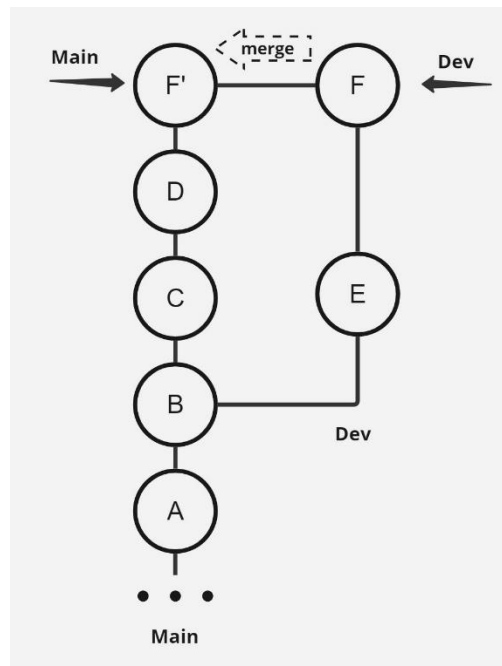
`git merge`, `git rebase`, `git cherry pick` (интеграция изменений)

Допустим, мы создали несколько коммитов на ветке `Dev` и на ветке `Main`, что дальше? Пока что файлы в ветках независимы, однако после того, как мы закончили работу в одной ветке, мы хотим, чтобы её изменения отразились на основной.

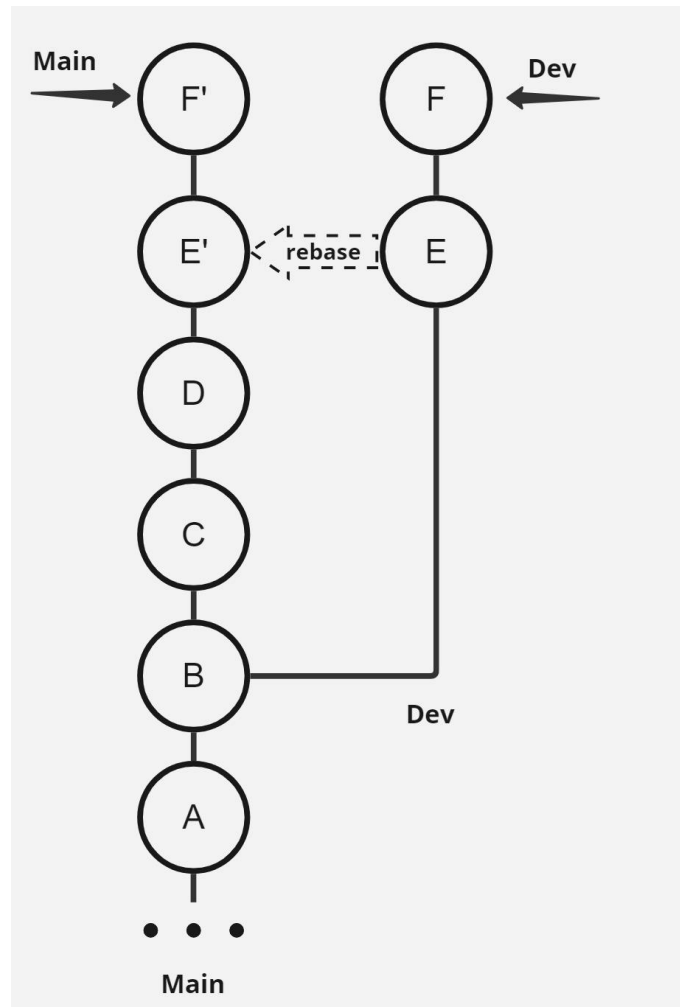


В Git существует два основных способа интеграции изменений из одной ветки в другую: **merge** и **rebase**. Также существует метод частичной интеграции изменений – **cherry pick**.

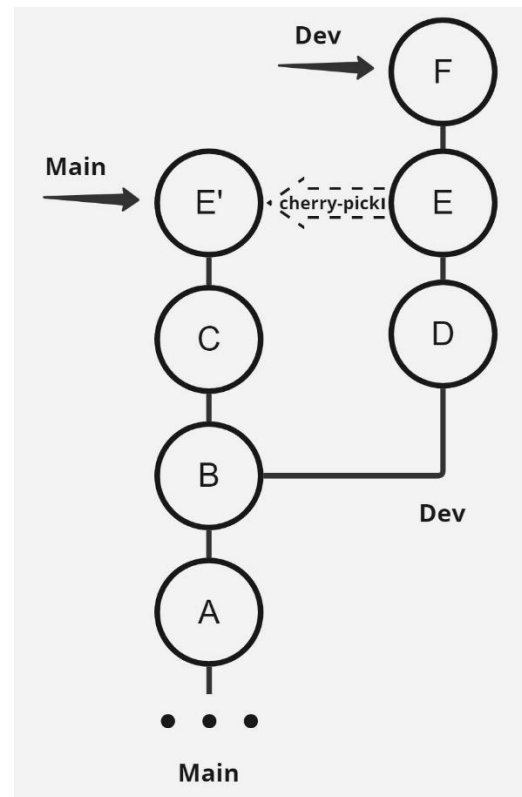
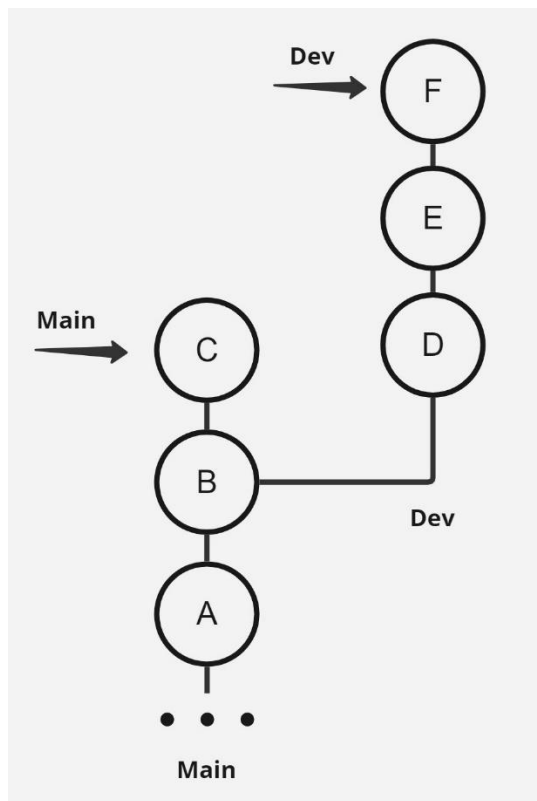
Merge – это процесс, при котором изменения из одной ветки (например, **Dev**) объединяются с другой веткой (например, **Main**). При слиянии создаётся новый коммит, который объединяет изменения из обеих веток. Выполняется командой
\$ git merge



Rebase – это процесс, при котором изменения из одной ветки "переносится" на другую. Это означает, что коммиты из ветки Dev будут добавлены поверх последнего коммита в ветке Main, тем самым создавая более линейную историю. Выполняется командой
`$ git rebase`



Cherry Pick – это команда в Git, которая позволяет извлекать и применять изменения из одного или нескольких конкретных коммитов к текущей ветке. Это удобно, когда нужно перенести определённые исправления или функции, не сливая всю ветку.



Чтобы применить черри-пик используем команду

```
$ git cherry-pick <индекс_коммита>
```

Таким образом мы можем выцепить один или несколько необходимых нам коммитов из другой ветки.

Конфликты при слиянии

Конфликты при слиянии (**merge conflicts**) в Git возникают, когда изменения в двух ветках не могут быть автоматически объединены. Это происходит, когда разные изменения сделаны в одной и той же части файла или, когда один из разработчиков удалил файл, который другой изменил. О том, как решать конфликты при слиянии можно найти [здесь](#).

git commit --amend (изменение коммитов)

Теперь подробнее поговорим об изменении коммитов. Допустим, мы создали коммит и обнаружили, что мы что-то в него не добавили. Конечно, мы можем внести изменения создать новый коммит, однако иногда полезно изменить последний созданный коммит.

Сделать это можно с помощью команды

```
$ git commit --amend
```


Перед использованием команды индексируем измененную версию файла, после чего эта команда добавит файл в коммит. С помощью этой команды также можно изменять комментарии крайнего коммита.

```
$ git commit --amend -m "Исправленное сообщение"
```

Использовать данную команду рекомендуется для не опубликованных локальных изменений.

git reset, git revert (отмена коммитов)

Иногда возникает необходимость в удалении коммитов или откате изменений. Для этих целей можно использовать команду

```
$ git reset
```

Для того, чтобы удалить последний коммит и вернуть все файлы к состоянию предшествующего коммита используется

```
$ git reset --hard HEAD~1
```

--hard означает, что команда удалит все изменения как из индекса, так и из рабочего каталога. Это приведёт к полной потере всех незакоммиченных изменений, и файлы будут возвращены к состоянию последнего коммита. Для сохранения изменений в индексе мы можем использовать --soft.

HEAD~1 означает, что указатель текущей ветки будет перемещён на один коммит назад от HEAD, то есть на родительский коммит последнего коммита. Это позволяет вернуться к предыдущему состоянию проекта. Вместо 1 мы можем поставить любое положительное целое число, чтобы указать, насколько назад мы хотим переместить указатель ветки (например, HEAD~2 вернёт на два коммита назад).

HEAD – это указатель на последний коммит в текущей ветке. Он служит для обозначения текущего положения в истории коммитов и показывает, где вы в данный момент находитесь в вашей репозитории.

Помимо этого, команду git reset можно использовать для того, чтобы убрать файл из индекса:

```
$ git reset <имя_файла>
```

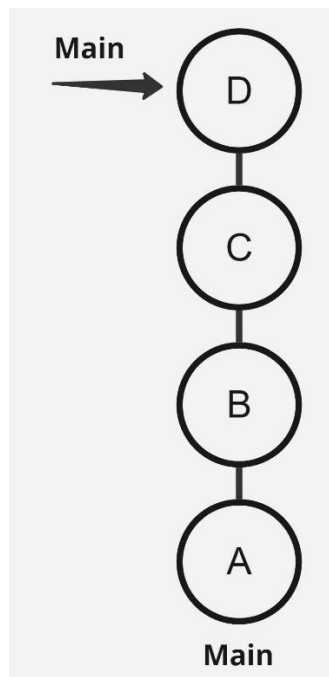
Или всех файлов из индекса:

```
$ git reset
```

Другим способом отмены коммитов является команда

```
$ git revert <идентификатор_коммита>
```

Допустим у нас есть следующая ветка:



Допустим, мы хотим откатить изменения до коммита C. Вместо `git reset HEAD~1`, который удалил бы последние коммиты мы используем `git revert`. Для этого нам необходимо получить идентификатор отменяемого коммита (SHA). Его можно получить с помощью `$ git log`

```
User@ /d/Projects/CTPweb/Project$ git log
commit e8cee1c1959a0adbe71c07bd6e60fd72de89b55a (HEAD -> master)
Author: User <150701577+ User @users.noreply.github.com>
Date: Mon Oct 7 19:27:41 2024 +0300

    Commit description
```

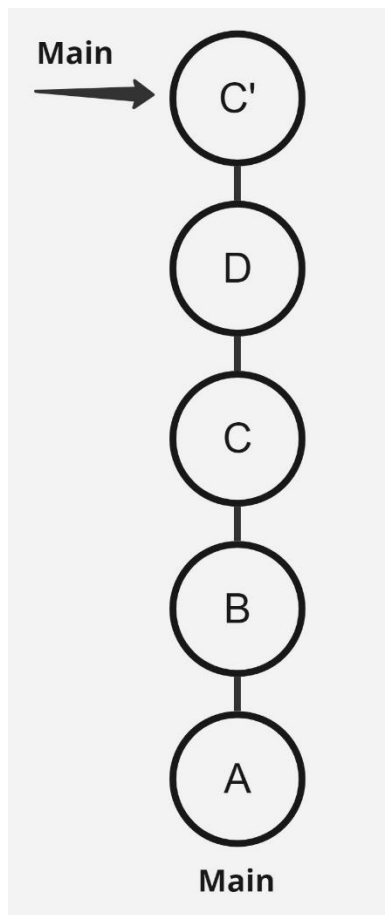
В данном случае команда выглядела бы так:

```
$ git revert e8cee1c1959a0adbe71c07bd6e60fd72de89b55a
```

На самом деле не обязательно писать этот код целиком, хватит первых символов, если они уникальны:

```
$ git revert e8cee1
```

После использования этой команды в нашей ветке появился бы новый коммит, откатывающий изменения к состоянию C.



Заключение №2

Таким образом, обобщая вышесказанное, получаем:

- 1 Для параллельной работы над одним проектом используются **ветки**, изолирующие изменения друг от друга.
- 2 Для создания ветки используется
`$ git branch`
- 3 Для переключения между ветками используется
`$ git checkout`
- 4 Для интеграции изменений используются **merge**, **rebase** и **cherry pick**.
`$ git merge` – создаёт новый коммит, который объединяет изменения из обеих веток.
`$ git rebase` – переносит копию коммитов с одной ветки на другую.
`$ git cherry-pick` – позволяет выборочно выцепить один или несколько коммитов.
- 5 При слиянии веток могут возникать конфликты **merge conflicts**, для решения которых Git предлагает набор инструментов.
- 6 При необходимости удаления коммитов используются
`$ git reset --soft` – для удаления коммита с сохранением изменений в индексе
`$ git reset --hard` – для удаления коммита с потерей всех изменений
- 7 Для удаления файлов из индекса также используется
`$ git reset`

- 8 При необходимости отката изменений используется команда, создающая новый коммит с обратными изменениями.

\$ git revert

Задание №2

Для успешного выполнения задания №2 требуется:

- 1 Откройте ранее созданную директорию.
- 2 Добавьте новую ветку Dev.
- 3 Изменяйте файлы в директории, создайте несколько коммитов в ветке Dev и сделайте merge, rebase или cherry pick на выбор.
- 4 Восстановите состояние файлов любым способом, предоставляемым технологией Git.

ЧАСТЬ №3

git remote (работа с удаленным репозиторием)

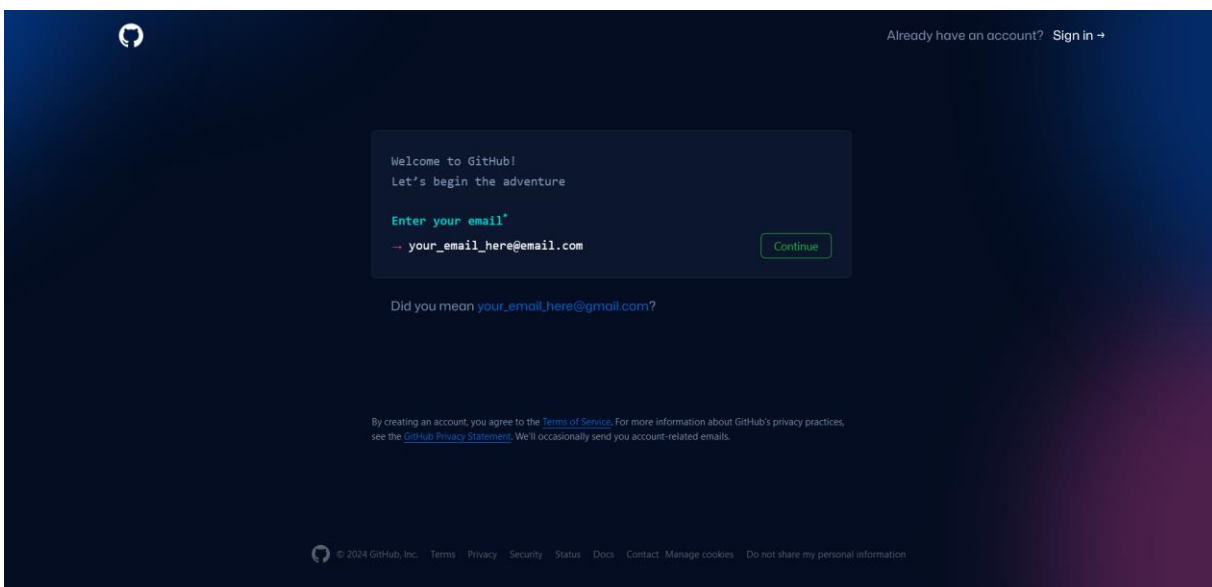
Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториях, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории.

Просмотр уже настроенных удаленных серверов осуществляется командой `git remote`. Если репозиторий был клонирован, вы должны увидеть по крайней мере источник, то есть имя, которое Git по умолчанию присваивает копирующему серверу. Параметр `-v` позволяет увидеть URL-адреса, которые Git хранит для сокращенного имени, используемого при чтении из данного удаленного репозитория и при записи в него:

```
/usr/bin/bash --login -i
User@ /d/Projects/Manuals_CTPweb/CTPweb_manuals$ git remote -v
origin https://github.com:/CTPweb_manuals.git (fetch)
origin https://github.com:/CTPweb_manuals.git (push)
User@ /d/Projects/Manuals_CTPweb/CTPweb_manuals$
```

Создание репозитория на платформе GitHub

Если у Вас ещё нет аккаунта на GitHub, перейдите по ссылке <https://github.com/signup> и поэтапно пройдите процесс регистрации.



После регистрации перейдите по ссылке <https://github.com/new>. Откроется окно создания нового репозитория. В этом окне имеется возможность настраивать имя репозитория, его описание, настраивать видимость (public / private) репозитория для других пользователей, создать README и .gitignore-файлы и выбрать лицензию.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner * / Repository name *

CTPweb-pr / first_lab

first_lab is available.

Great repository names are short and memorable. Need inspiration? How about [fictional-memory](#) ?

Description (optional)

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

☒ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: Java

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set `main` as the default branch. Change the default name in your [settings](#).

☐ You are creating a public repository in your personal account.

[Create repository](#)

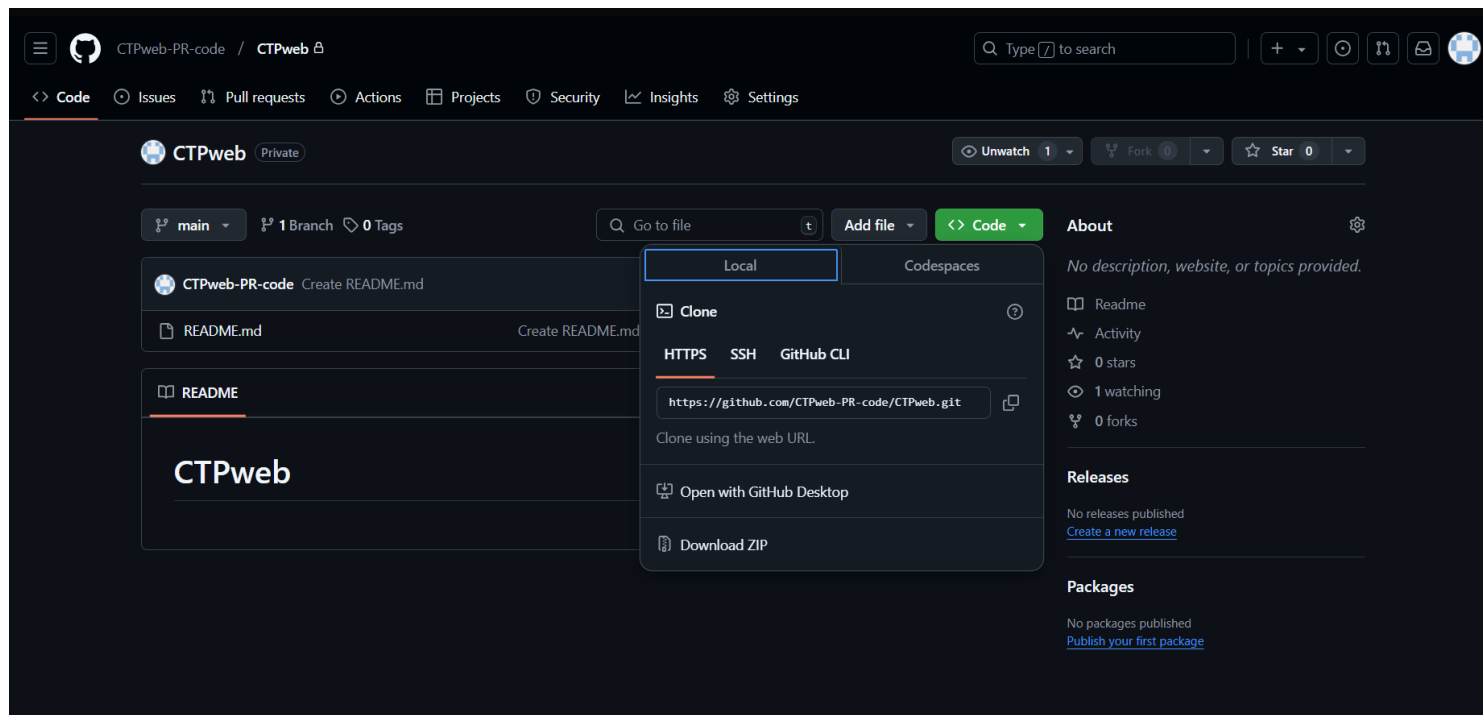
.gitignore – файл, который содержит объекты (расширения, каталоги и отдельные файлы), которые не должны быть подвержены индексации. Изменения в таких объектах фиксируются в истории проекта.

git clone, git fetch, git pull (извлечение данных из удаленных репозиториях)

Извлечение данных из удаленных репозиториях в Git осуществляется с помощью нескольких ключевых методов.

\$ git clone <repository>

Эта команда используется для клонирования репозитория. Она копирует весь репозиторий (включая историю) в новую директорию.



Для использования `git clone` копируем URL репозитория, пример:
`$ git clone https://github.com/CTPweb-PR-code/CTPweb.git`

Команда

`$ git fetch`

позволяет синхронизировать локальный репозиторий с удалённым, загружая обновления (ветки, коммиты и теги), но не изменяя текущую рабочую копию или активную ветку. Эта команда полезна для анализа изменений перед их интеграцией. Полученные изменения доступны через ссылки `origin/<имя_ветки>`, которые можно просмотреть с помощью
`$ git log origin/<имя_ветки>`

Для слияния локального репозитория с текущей рабочей копией используются уже знакомые `git merge`, `git rebase`.

`$ git merge origin/branch_name`

Для объединения изменений из удаленного репозитория с текущей веткой используется команда

`$ git pull`

Она автоматически выполняет `git fetch`, а затем сливает изменения. По умолчанию используется `merge`, но для `rebase` можно указать флаг:

`$ git pull --rebase`

git push (отправка изменений)

Для отправки локальных изменений в удалённый репозиторий используется команда

```
$ git push <репозиторий> <ветка>
```

Она синхронизирует локальную ветку с её связанной удалённой веткой. Репозиторий по умолчанию – origin.

origin – это стандартное имя удалённого репозитория, автоматически присваиваемое при клонировании репозитория с помощью команды git clone. Оно используется как ссылка на URL-адрес удалённого репозитория, упрощая взаимодействие с ним.

Пример отправки главной ветки в удалённый репозиторий:

```
$ git push origin main
```

Если ветка отсутствует в удалённом репозитории, используйте:

```
$ git push -u origin new_branch
```

Вместо new_branch подставьте название своей ветки.

Для отправки всех веток можно использовать:

```
$ git push --all origin
```

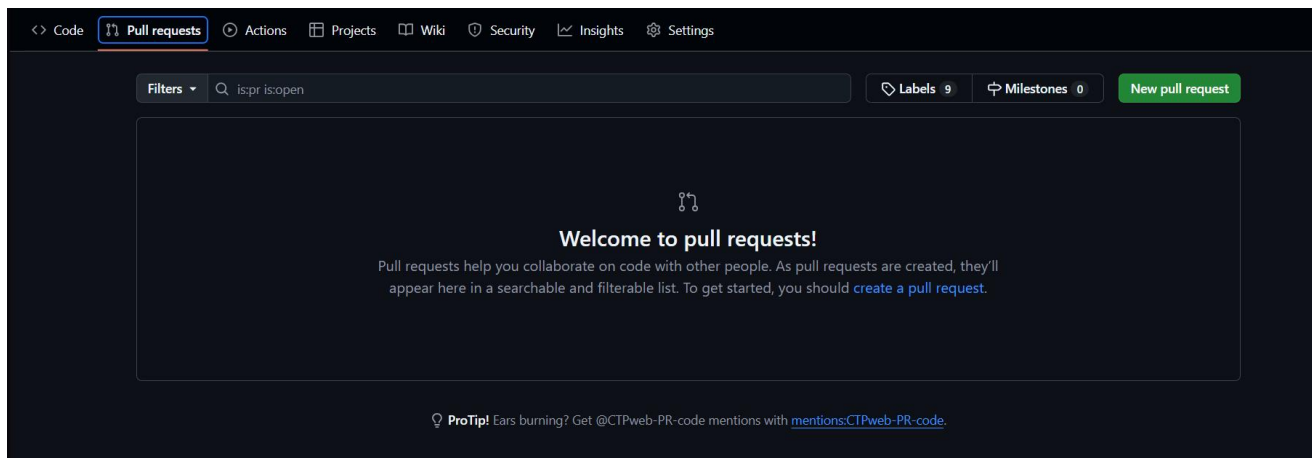
Fork

Fork – это копия репозитория, которую вы можете создать на основе существующего репозитория. Они создаются для независимой работы с проектом. Их применяют для разработки новых функций, исправления ошибок или предложений изменений в чужие проекты через Pull Request.

Pull Request – это запрос на внесение ваших изменений из форка или ветки в основной репозиторий. Его используют, чтобы показать свои доработки, обсудить их с авторами проекта и, при одобрении, объединить с основным кодом.

Для создания Fork находим нужный репозиторий и нажимаем на кнопку Fork. После этого создается копия этого репозитория, но уже в вашем аккаунте.

Для создания Pull request используем соответствующую кнопку.



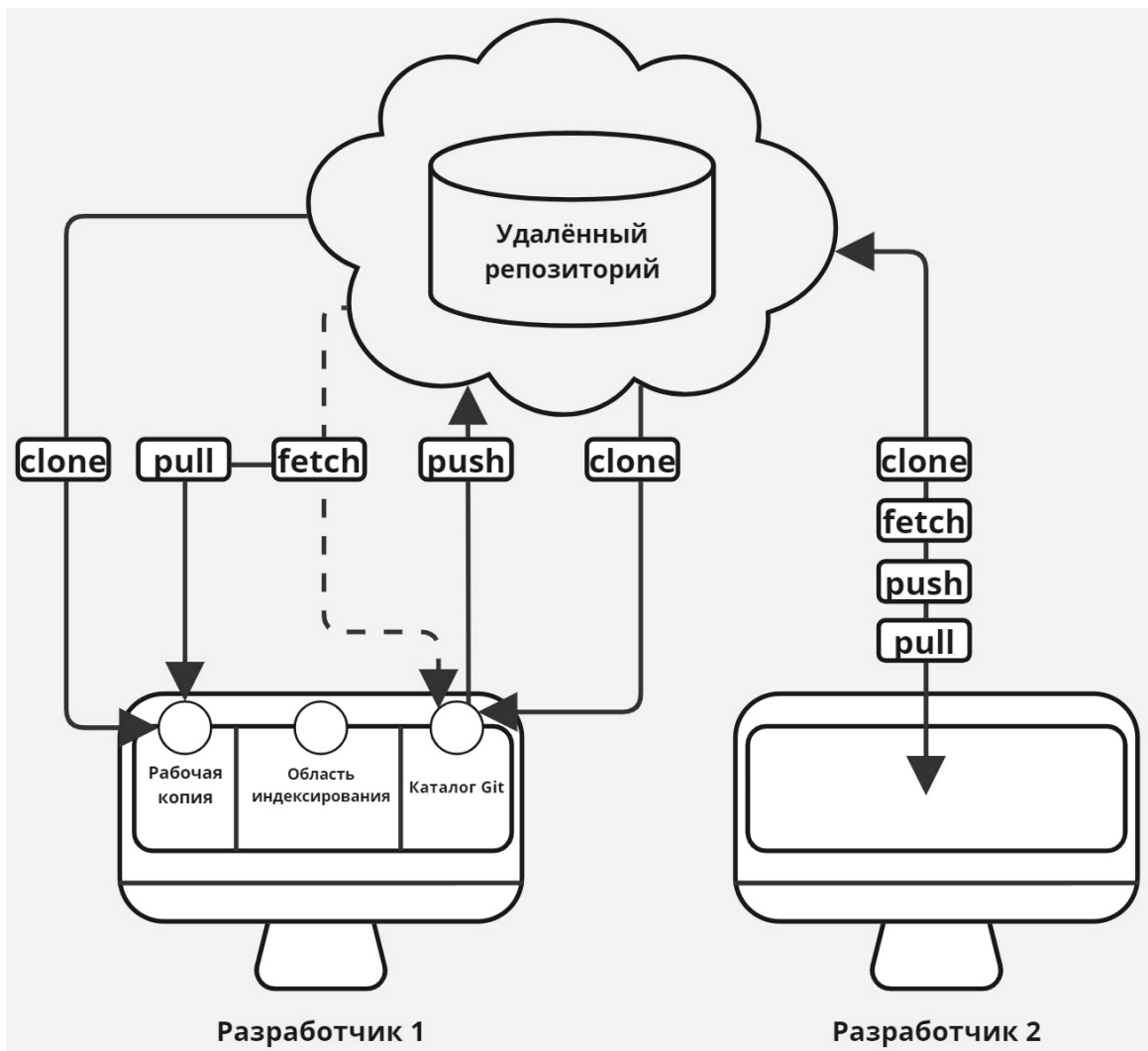
После создания Pull Request изменения будут рассмотрены другими программистами, и после их подтверждения они будут внесены в основной репозиторий.

Заключение №3

Таким образом, обобщая вышесказанное, получаем:

- 1 Для использования удалённых репозиторий существуют различные платформы (GitHub, GitLab и др.)
- 2 Просмотр уже настроенных удаленных серверов осуществляется командой
`$ git remote`
- 3 `.gitignore` – файл, который содержит объекты, которые не должны быть подвержены индексации.
- 4 Для клонирования репозитория используется
`$ git clone <repository>`
`$ git fetch` – синхронизирует локальный репозиторий с удалённым, загружая обновления без изменения текущей рабочей ветки. Изменения доступны через ссылки `origin/<имя_ветки>` и просматриваются с помощью
`$ git log origin/<имя_ветки>`.
- 5 Для слияния используются команды
`$ git merge`
`$ git rebase`
- 6 `$ git pull` – выполняет `git fetch`, а затем автоматически сливает изменения.
- 7 Для отправки локальных изменений в удалённый репозиторий используется команда
`$ git push <репозиторий> <ветка>`
- 8 Fork – это удалённый репозиторий, созданный на основе другого существующего репозитория.
- 9 Pull Request – это запрос на внесение ваших изменений из форка или ветки в основной репозиторий.

Визуализация работы команд clone, fetch, pull, push:



Задание №3

Для успешного выполнения задания №3 требуется:

- 1 Создайте удалённый репозиторий.
- 2 Отправьте изменения из ранее созданного локального репозитория в удалённый.
- 3 Передайте напарнику ссылку на свой репозиторий.
- 4 Предоставьте ему доступ к своему репозиторию (Settings > Collaborators)
- 5 Напарник создаёт Fork.
- 6 Напарник клонирует репозиторий.
- 7 Напарник вносит изменения в файлы и загружает изменения в удалённый репозиторий.
- 8 Напарник создаёт Pull Request.
- 9 Просмотрите Pull Request и сделайте Merge.
- 10 Загрузите актуальную версию файлов из удалённого репозитория в свой локальный, используя Rebase для слияния.