

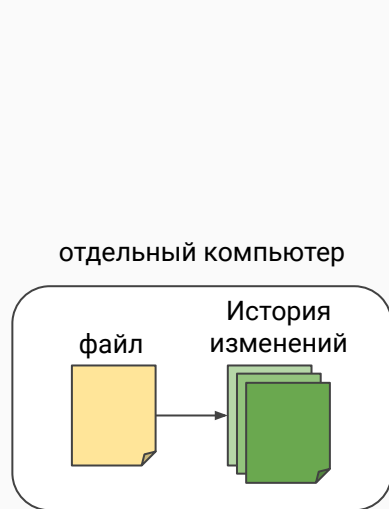
Основы DevOps

Системы контроля версий (на примере Git)

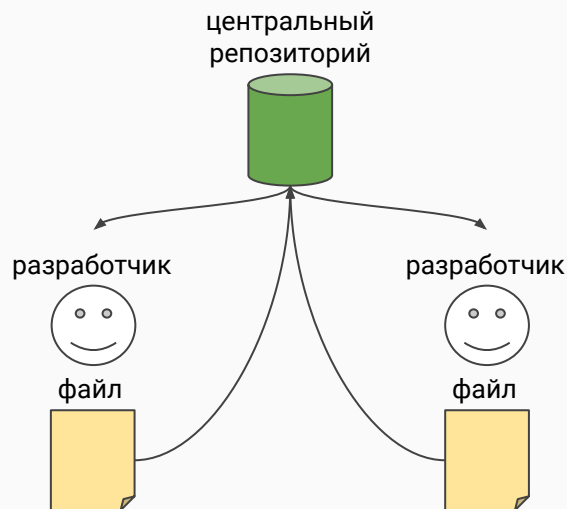
Введение

- **Система контроля версий** - программа, которая хранит историю изменений одного или нескольких файлов, фиксирует время и авторов всех изменений, а также позволяет вернуться к ранним версиям отслеживаемых файлов.
- Системы контроля версий бывают трех видов - **локальные, централизованные и распределенные**. Очевидно, что **локальные системы** предназначены для хранения изменений файлов на одном компьютере. В свою очередь **централизованные системы** позволяют работать над одним проектом сразу нескольким разработчикам. Данные хранятся на одном общем сервере, куда и вносятся необходимые изменения, которые, в свою очередь, и отслеживает система контроля версий. Наконец, **распределенные системы** также могут использовать централизованное хранилище, но в этом случае у каждого разработчика есть полная копия проекта - сначала изменения вносятся в копию, а затем копия вливается в центральный репозиторий. Это снижает риск потерять все данные проекта в том случае, если центральный репозиторий выйдет из строя.

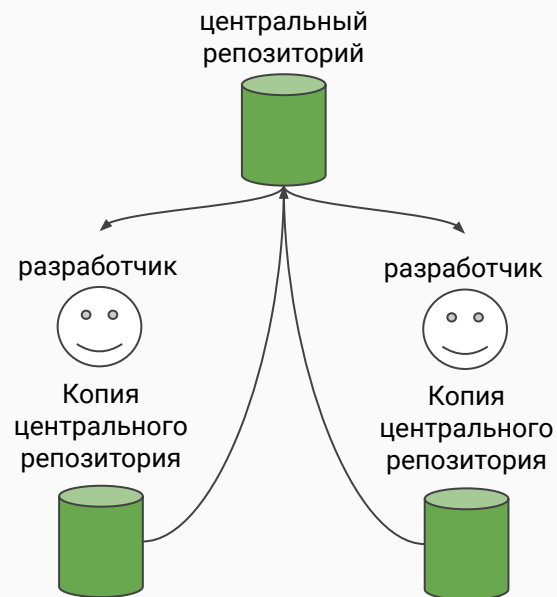
Визуальное представление типов систем контроля версий



**Локальная
система**



**Централизованная
система**



**Распределенная
система**

- Самой популярной распределенной системой контроля версий на данный момент (конец 2023 года) следует признать **Git**, автором которой является Линус Торвальдс - создатель ядра Linux. Данная система была представлена публике в 2006 году, она абсолютно бесплатна, обладает хорошим быстродействием и надежностью, прекрасно подходит как для разработки небольших программ, над которыми работает один человек, так и для широкомасштабных проектов, в которых задействованы тысячи разработчиков.
- Основной принцип работы **Git** опирается на хранение состояний отслеживаемых файлов. Каждый раз, когда происходит фиксация изменений (т.н. коммит), **Git** сохраняет слепок всех файлов - если измененные файлы сохраняются целиком, но вместо файлов без изменений хранится ссылка на предыдущее изменение или на первоначальную версию файла (если изменений еще не было).
- Внутри системы файлы могут находиться в трех состояниях. Во-первых, это измененные файлы, которые пока не зафиксированы. Во-вторых, это проиндексированные файлы - они были изменены и также пока не зафиксированы, но подготовлены для фиксации (коммита). В-третьих, зафиксированные (сохраненные) версии файлов.

Установка и настройка Git

Установка Git

- Если мы используем операционную систему на ядре Linux, то для установки Git мы должны использовать менеджер пакетов нашего дистрибутива. Например, если у нас Ubuntu, то Git можно проинсталлировать при помощи команды **apt**:

```
sudo apt install -y git
```

- Пользователи операционной системы Windows могут загрузить установочный файл непосредственно с официального сайта Git: <https://git-scm.com/downloads>. После этого надо запустить этот файл и пройти несложный процесс установки.
- После установки программы мы должны проверить, что все прошло без ошибок. Для этого в обеих операционных системах необходимо перейти в терминал и попытаться узнать версию Git при помощи следующей команды:

```
git --version
```

Настройка пользователя Git

- Перед тем, как начать использовать Git, необходимо указать (команда **git config**) имя (**user.name**) и эл. почту (**user.email**) пользователя. Важно понимать, что данные настройки (а также все остальные) могут быть установлены в трех контекстах. Во-первых, в системном (**--system**) - для всех пользователей системы (требует права суперпользователя). Во-вторых, в глобальном (**--global**) - будет распространяться только на все репозитории конкретного пользователя. В-третьих, в локальном (**--local** либо вообще без параметра) - распространяется только на тот репозиторий, где эти настройки установлены (в данном случае у вас уже должен быть репозиторий). В плане приоритета сначала идут локальные настройки, затем глобальные, а в самом конце - системные.
- Установим (в командной строке) имя и почту нашего пользователя в глобальном контексте - чтобы не заниматься этим внутри каждого нового репозитория:

```
# установка имени пользователя
git config --global user.name "Alexander Kovalenko"

# установка электронной почты пользователя
git config --global user.email user@gmail.com
```


Настройка преобразований символов концов строк и редактора

- При работе на операционных системах Windows и Linux может возникнуть проблема с различными символами концов строк (на Linux - `\n`, а на Windows `\r\n`). Чтобы не разработчики не перезаписывали друг друга в Git была введена настройка **core.autocrlf**, которая как раз отвечает за контроль над концами строк. При значении параметра **false** контроль вообще выключен (поведение по умолчанию), если значение установлено в **true**, то при сохранении в репозиторий окончания автоматически конвертируются в Linux-подобные `\n`, а при получении данных из репозитория - в Windows-подобные `\r\n`. Если установлено значение **input**, то при получении данных из репозитория конвертаций не происходит, а при сохранении новые и измененные файлы будут автоматически получать Linux-подобные окончания `\n`. Для примера установим значение **input**:

```
git config --global core.autocrlf input
```

- В некоторых редких случаях Git может вызвать текстовый редактор для заполнения некоторых данных. Чтобы не полагаться на Git, а указать его явно заранее, используется настройка **core.editor**. Для примера установим **nano**:

```
git config --global core.editor nano
```

Инициализация репозитория Git

- Для начала работы нам необходимо выбрать некую директорию (либо создать новую) - назовем ее **project**, затем перейдем в нее в командной строке и выполним следующую команду (существуют ли другие файлы и директории внутри - не имеет значения):

```
git init
```

В результате в директории будет создана новая директория **.git**, в которой и будут храниться состояния отслеживаемых файлов, настройки и дополнительные объекты репозитория. Отслеживаться будут все изменения внутри директории **project**, а также внутри всех ее вложенных директорий.

- Если мы хотим создать репозиторий, который будет центральным хранилищем для других репозиториях (подробнее об этом мы поговорим несколько позже), то к изначальной команде следует добавить параметр **--bare**:

```
git init --bare
```

По сути после этого в директории **project** появятся те файлы, которые находились в директории **.git** после выполнения простой команды **git init**.

Основы работы с локальными изменениями в Git репозитории

Команда **git add** - индексирование изменений

- Представим, что в нашей директории `project` появились два новых файла - **index.py** и **lib.py**. Для того, чтобы проиндексировать эти изменения (внести их в кандидаты для окончательной фиксации), мы должны использовать команду **add**. Если мы хотим проиндексировать вообще все новые или измененные файлы, то мы должны поставить после команды точку:

```
git add .
```

Также мы можем через пробел явно указать те файлы, которые хотим подготовить для фиксации:

```
git add index.py lib.py
```

Кроме того, надо понимать, что необязательно готовить для фиксации все новые или измененные файлы - допустим, мы закончили работу над **lib.py**, а **index.py** еще не готов. Поэтому следует проиндексировать только **lib.py**:

```
git add lib.py
```

Команда **git commit** - фиксация изменений

- Представим, что мы все же добавили все новые файлы (***index.py*** и ***lib.py***) в индекс и хотим сохранить (зафиксировать) в репозитории состояние (и само их существование) этих файлов на данный момент. Для этого используется команда **git commit**, после которой следует применить параметр **-m**, а затем в кавычках вставить текст описания измененных файлов (например, "first commit"). Данное действие называется "сделать коммит":

```
git commit -m "first commit"
```

- Может так случиться, что мы забудем применить параметр **-m** и описание коммита. Удивительно, но ошибки не произойдет - в этом случае у нас откроется программа, которая установлена в настройке **core.editor**. При помощи этого текстового редактора нам и будет позволено ввести текстовое описание коммита. Именно поэтому установку редактора надо производить заранее, чтобы не столкнуться с тем редактором, работать с которым мы не умеем:

```
git commit
```

Команда **git status** - просмотр состояния репозитория после изменения файлов

- Время от времени нам захочется посмотреть на список тех файлов, которые были добавлены или изменены, но еще не были зафиксированы. Для этого следует применять команду **git status**. Если мы применим ее после предыдущего коммита, то увидим надпись *"nothing to commit, working tree clean"*:

```
git status
```

```
C:\Users\User\Desktop\project>git status
On branch main
nothing to commit, working tree clean

C:\Users\User\Desktop\project>|
```

- Теперь представим, что мы добавили функцию (например, записи из одного в файл в другой) в **lib.py** и добавили новый файл **test.py**. После этого запустим **git status** опять и убедимся, что Git увидел нововведения:

```
C:\Users\User\Desktop\project>git status
On branch main
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:   lib.py

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

no changes added to commit (use "git add" and/or "git commit -a")

C:\Users\User\Desktop\project>
```

Команда **git status** - просмотр состояния репозитория после индексации

- Теперь проиндексируем все изменения, чтобы посмотреть, как это видит Git:

```
git add .
```

```
git status
```

Мы можем убедиться, что все отслеживается безо всяких проблем:

```
C:\Users\User\Desktop\project>git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   lib.py
        new file:   test.py

C:\Users\User\Desktop\project>
```

- Добавим еще одну любую функцию в **lib.py** и увидим, что Git разделит ранее проиндексированные изменения из **lib.py** (при коммите сохранятся только они) и изменения в **lib.py** без индексации (они не будут сохранены при коммите):

```
C:\Users\User\Desktop\project>git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   lib.py
        new file:   test.py

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:   lib.py

C:\Users\User\Desktop\project>
```

Команда **git restore --staged** - удаление файлов из индекса

- Если мы взглянем на проиндексированный файл **test.py**, то обнаружим, что он пуст. В этом случае не будет разумным решением добавлять его в коммит, однако, как уже было сказано, он уже проиндексирован. Для того, чтобы убрать файл из индекса используют следующую команду:

```
git restore --staged test.py
```

- Нам может понадобиться удалить не конкретный файл из индекса, а полностью все файлы. Для этого надо применить ту же самую команду `git restore --staged`, но после нее следует указывать не названия файла, а точку:

```
git restore --staged .
```


Команда **git restore** - отмена изменений в файле

- На данный момент у нас нет индексированных файлов - есть измененный **lib.py** и добавленный **test.py**. У нас может возникнуть необходимость убрать все изменения из **lib.py** (или даже изменения из всех файлов) - т.е. вернуть состояние файла на момент последнего коммита. Git содержит такой функционал - это команда **git restore**, после которой следует написать название файла. Все изменения файла будут безвозвратно удалены:

```
git restore lib.py
```

Если есть необходимость отменить вообще все изменения со времени последнего коммита, то вместо названия файла надо поставить точку. Важный момент - новые файлы (которые еще не были закомичены) останутся неизменными:

```
git restore .
```

- Убедимся, что остался **test.py** - проиндексируем его и зафиксируем изменения:

```
git add test.py  
git commit -m "added test.py"
```

Команда **git log** - просмотр истории коммитов

- На данный момент у нас уже было две фиксации изменений (2 коммита). Если мы хотим увидеть всю историю, нам необходимо выполнить команду `git log` (будет показан список коммитов, с их авторами, хэшами и датами создания):

```
git log
```

- Часто команда `git log` занимает довольно много места на экране, что затрудняет понимание происходящего. Чтобы решить эту проблему, можно отобразить историю в более компактном виде, с помощью параметра **--oneline** (будут отображены только названия коммитов и их хэши):

```
git log --oneline
```

- Когда у нас будет много коммитов, притом будет создано много веток (о них мы поговорим позже), то нам будет удобно посмотреть историю в виде древовидной структуры. Это можно осуществить с помощью параметра **--graph**:

```
git log --graph
```

Команда **git checkout** - обратимый возврат к предыдущим коммитам

- Иногда нам надо будет посмотреть состояние нашего проекта на момент некоего прошлого коммита - т.е. по сути вернуться в прошлое, переключиться на версии файлов в этот момент. Чтобы это сделать следует применить команду **git checkout**, после которой надо указать хэш того коммита, к которому мы хотим вернуться (хэш можно увидеть в истории при помощи команды **git log**):

```
git checkout dbfb3c9468b946bb0733290af69260e8df4704ed
```

- Проанализировав состояние прошлого коммита, мы рано или поздно захотим вернуться к актуальному состоянию репозитория. Чтобы это сделать, нам надо применить команду **git switch**, после которой надо указать тире (мы еще поговорим подробнее **git switch** в дальнейших разделах):

```
git switch -
```

Команда **git reset** - необратимый возврат к предыдущим коммитам

- До этого мы использовали временный возврат к предыдущим версиям проекта. Однако может случиться, что наш проект зашел куда-то не туда, и мы хотим необратимо сбросить все изменения до предыдущего коммита. Для этого используется команда **git reset**, после которой надо указать хэш коммита. Если применить параметр **--hard**, то возврат будет максимально необратимым:

```
git reset --hard dbfb3c9468b946bb0733290af69260e8df4704ed
```

- Если мы применим параметр **--soft**, то возврат тоже будет необратимым (т.е. пропадут все коммиты), однако репозиторий не изменится, более того - все различия будут проиндексированы, т.е. подготовлены для коммита:

```
git reset --soft dbfb3c9468b946bb0733290af69260e8df4704ed
```

- Еще один параметр **--mixed** (он применяется, если мы не будем указывать параметры) похож на **--soft**, но есть одно отличие - при его применении репозиторий не изменяется, но различия не будут проиндексированы:

```
git reset --mixed dbfb3c9468b946bb0733290af69260e8df4704ed
```

Команда **git revert** - отмена изменений какого-либо прошлого коммита

- Время от времени нам может прийти осознание, что какой-либо коммит из прошлого был ошибочным, и мы хотим его отменить. Для этого у нас есть команда **git revert**, после которой надо указать хэш того коммита, который мы хотим отменить. Важно понимать, что мы не потеряем предыдущие коммиты - просто будет добавлен новый, который отменит ненужные нам изменения. Отменим самый первый коммит, который добавил файлы **index.py** и **lib.py**:

```
git revert dbfb3c9468b946bb0733290af69260e8df4704ed
```

После того, как мы выполним приведенную выше команду, откроется браузер по умолчанию, и мы сможем вписать название нового коммита. Если же мы хотим, чтобы описание добавилось автоматически, то следует применить параметр **--no-edit**:

```
git revert --no-edit dbfb3c9468b946bb0733290af69260e8df4704ed
```

Команда **git show** - просмотр изменений прошлых коммитов

- Время от времени нам может понадобиться не просто посмотреть описание какого-либо коммита, а принесенные им конкретные изменения (какие файлы были добавлены, какие удалены, как построчно поменялось содержание и т.д.). Для решения этой задачи используется команда **git show**, после которой следует хэш коммита:

```
git show dbfb3c9468b946bb0733290af69260e8df4704ed
```

Файл **.gitignore** - игнорирование файлов при отслеживании

- Как бы это парадоксально не звучало, в нашем проекте могут быть файлы, которые не должны отслеживаться - всяческие конфигурации, секретные пароли, результаты компиляции каких-то программ и т.д. Для игнорирования таких файлов и даже целых директорий используется файл **.gitignore**. Ниже представлен пример такого файла, где приведены наиболее типичные случаи:

```
# игнорирование файла или директории (в любой директории или вложенной директории)
somename

# игнорирование директории (в любой директории или вложенной директории)
dirname/

# игнорирование файла или директории по их точному пути от корня проекта
/some/concrete/path/somename

# игнорирование директории по ее точному пути от корня проекта
/some/concrete/path/dirname/

# игнорирование всех файлов с конкретным расширением (данном случае .log)
*.log

# файлы или директории, содержащие часть пути, автоматически считаются расположенными в корне проекта
some/dir/somename

# две точки означают, что перед файлом или директорией может находиться сколько угодно (хоть несколько) директорий
**/some/dir/somename
```

- Чтобы файл **.gitignore** начал действовать, он сначала должен быть включен в какой-либо коммит!

Команда **git rm --cached** - прекращение отслеживания файла

- Представим, что некоторый файл **example.py**, который раньше отслеживался, теперь должен быть отключен от отслеживания. Очевидно, что он должен быть занесен в файл **.gitignore** (который опять должен быть включен в коммит), но даже после этого мы увидим, файл продолжает отслеживаться. Чтобы ранее зафиксированный файл перестал быть видимым для репозитория, необходимо применить команду **git rm --cached**, потом добавить полученные изменения в индекс и зафиксировать это состояние с помощью коммита:

```
git rm --cached example.py

# added example.py to the .gitignore

git add .gitignore

git commit -m "removed example.py from version control"
```


Ветвление в Git

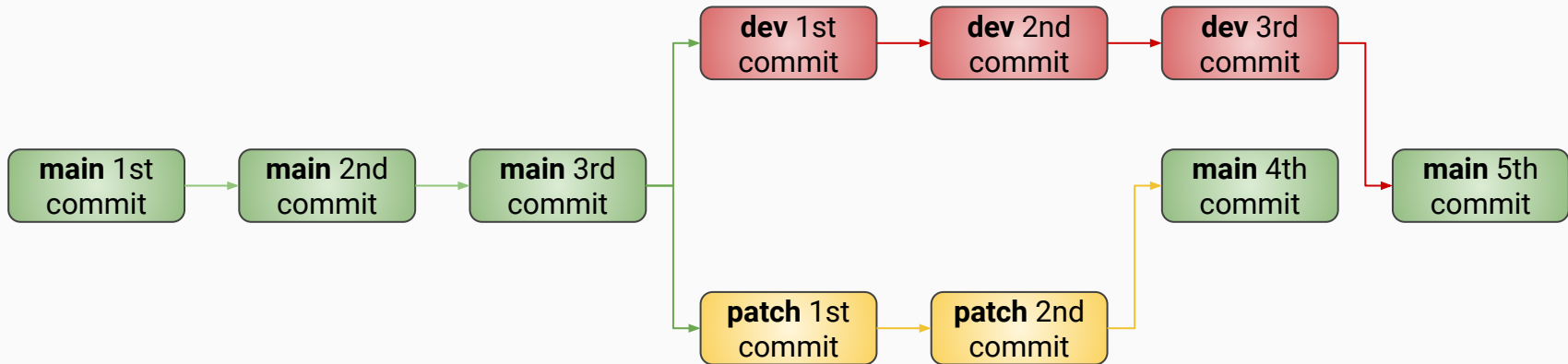
Введение

- Нас может полностью устраивать состояние нашего репозитория, однако рано или поздно наступит момент, когда потребуется дальнейшая разработка. Если мы будем разрабатывать как раньше (через последовательные коммиты), то код в репозитории может перестать быть работоспособным (на момент разработки).
- Чтобы решить обозначенную проблему, следует применить механизм ветвления. Мы создаем отдельную ветку - своего рода копию репозитория, переключаемся на нее, ведем разработку в этой копии, а основная версия остается без изменения. Вся прелесть ветвления заключается в том, что копия является виртуальной - мы получим возможность переключаться между стабильной версией (для использования репозитория) и копией (для разработки) находясь в одной и той же директории. После окончания и тестирования отдельной ветки ее содержимое вливается в основную версию репозитория.
- Основная версия репозитория также является веткой, но она создается автоматически при инициализации репозитория. Обычно эта основная ветка носит название `master` или `main` (современная версия). Изначальное название можно установить при помощи параметра **`init.defaultBranch`**:

```
git config --global init.defaultBranch main
```

Визуальное представление ветвления в Git

- Ниже представлена разработка внутри репозитория, который содержит 3 ветки. После 3 коммита в контексте основной ветки **main** были созданы еще две ветки - **dev** и **patch**. Ветка **patch** разрабатывалась на протяжении двух коммитов, затем ее изменения были объединены с **main**. В свою очередь разработка ветки **dev** заняло целых три коммита - результат также был объединен с **main** (который уже содержал слияние с **patch**). Из этой схемы можно понять, что в числе прочего механизм ветвление идеально подходит для тех случаев, когда происходит командная разработка с участием нескольких программистов.



Создание веток

- Начнем новый репозиторий (главная ветка - **main**) для разработки на языке программирования Python и сделаем как минимум два коммита - в первом добавим файлы **app.py** и **requirements.txt** ("**first commit**"), во втором - директории **static** и **templates** со стилями и шаблонами нашего приложения ("**integrated templates and styles**").
- Теперь пришла пора подключать базу данных и создавать файл для работы с данными (**dao.py**). Не будем ломать наш сайт, а будет делать все обновления в новой ветке под названием **development**. Для этого надо применить команду **git branch**, после которой указывается имя новой ветки (это не единственный способ создания ветки, но применим пока именно его). Новая ветка будет создана на основании файлов ветки, в которой мы в данный момент находимся (**main**):

```
git branch development
```

Просмотр веток

- Теперь у нас появилась возможность просмотреть все ветки, какие у нас есть в наличии - это осуществляется при помощи команды **git branch** без всяких дополнительных команд и аргументов:

```
git branch
```

В результате мы увидим список с репозиториями, где активный репозиторий (**main**) отмечен звездочкой:

```
development  
* main
```

Переключение между ветками

- Чтобы начать работу в новой ветке, надо сначала на нее переключиться. Для этого существует несколько способов, но самым современным на данный момент считается применение команды **git switch**, после которой следует вписать имя ветки, на которую мы хотим переключиться (**development**):

```
git switch development
```

Устаревшим, но тем не менее рабочим способом переключения веток является применение команды **git checkout**, после которой идет название ветки:

```
git checkout development
```

- Существует объединенный способ создания ветки и переключения на нее. Например нам надо создать ветку **test** и сразу попасть в нее - для этого будем использовать команду **git switch** совместно с параметром **-c**:

```
git switch -c test
```

Это же умеет делать команда **git checkout**, но уже с параметром **-b**:

```
git checkout -b test2
```

Удаление веток

- В процессе знакомства с ветками мы сделали две не очень нужные нам ветки - **test** и **test2**. Сейчас мы научимся уничтожать ветки на их примере. В общем случае для удаления используется команда **git branch** с параметром **-d**, после которого идет название удаляемой ветки (удалим сначала **test**). Важный момент - при удалении ветки мы не должны быть на нее переключены:

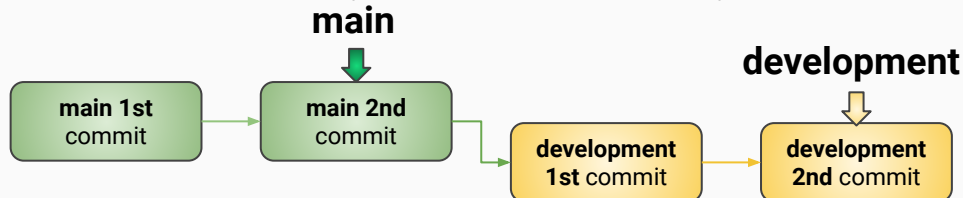
```
git branch -d test
```

- Если внутри ветки были сделаны коммиты, сама ветка еще не объединена с родительской (мы поговорим об этом несколько позднее), и мы пытаемся удалить ее, Git вернет нам ошибку и ничего удалять не будет. Чтобы заставить его это сделать, надо применить параметр **-D** - в этом случае удаление произойдет успешно:

```
git branch -D test2
```

Команда **merge** - слияние веток (вариант **fast-forward**)

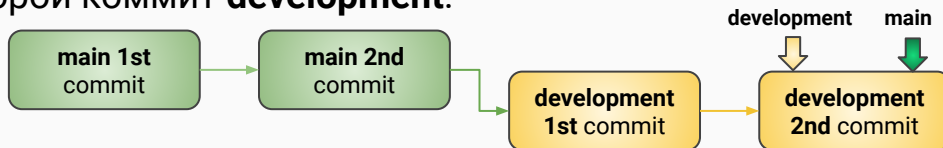
- Представим, что мы продолжаем успешную разработку в ветке **development** и добавили в ней два коммита - один с файлом **dao.py**, а второй с файлом **utils.py**. В этом случае наша история будет выглядеть следующим образом:



Теперь пришло время вставить данные ветки **development** в основную ветку - **main**. Для этого надо переключиться на ветку **main**, а затем выполнить команду **git merge**, после которой следует указать ветку, коммиты которой нам нужны:

```
git switch main  
git merge development
```

Т.к. после создания ветки **development** в ветке **main** не было изменений, при слиянии Git применит т.н. **fast-forward** стратегию - просто перенесет указатель **main** на второй коммит **development**:



Команда **merge** - слияние веток (вариант **ort**)

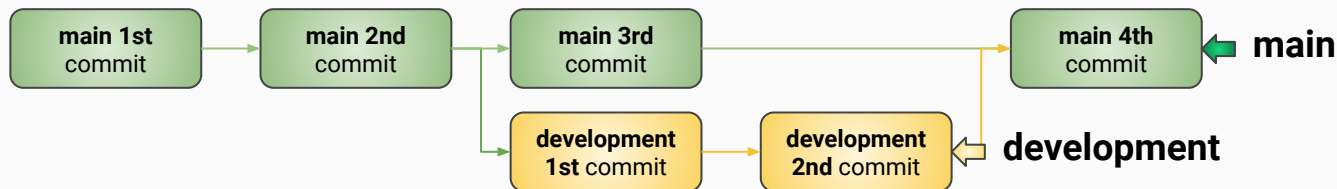
- Может оказаться, что после создания ветки **development** в ветке **main** были добавлены дополнительные коммиты - в этом случае схема веток выглядит так:



В данной ситуации для слияния веток нам делать то же самое, что мы делали в предыдущем случае - переключиться на ветку **main** и выполнить команду **merge**:

```
git switch main  
git merge development
```

После все этих действий у нас не будет переноса курсора в **main** - в этой ветке с помощью стратегии **ort** (**ostensibly recursive three-way**) будет автоматически создан коммит, который вберет в себя данные ветки **development**:



Команда **rebase** - перебазирование веток (теория)

- Как мы могли заметить, **git** умеет объединять изменения не последовательно, а рекурсивно, используя стратегию **ort**. Однако это достается нам дорогой ценой - с одной стороны создается дополнительный коммит, а с другой стороны - история коммитов может начать выглядеть крайне неприятно, с большим количеством линий и пересечений, что может затруднить восприятие. Для решения этой проблемы у нас есть еще одна команда под названием **rebase**.
- Обычно для **rebase** применяется следующая тактика. Сначала мы узнаем, что в главной ветке **main** появились те коммиты, которых нет в ветке **development**. Затем, находясь в ветке **development**, выполняем команду **git rebase main**. После этого произойдет следующее - те коммиты из **main**, которых нет в **development**, в **development** появятся, причем появятся в качестве первых. В свою очередь те коммиты, которые уже были в **development**, сместятся после коммитов из **main** и получат новые хэши. Затем вернемся в **main** и выполним команду **git merge development**, которая осуществится с помощью стратегии fast forward.
- Команду **git rebase** не следует делать в ветках, на которые ссылаются другие ветки, т.к. происходит изменение хэшей коммитов, что сильно затруднит слияние побочных веток в ту ветку, которая использовала **git rebase** (ветки не смогут адекватно найти общих предков из-за изменившихся хэшей).

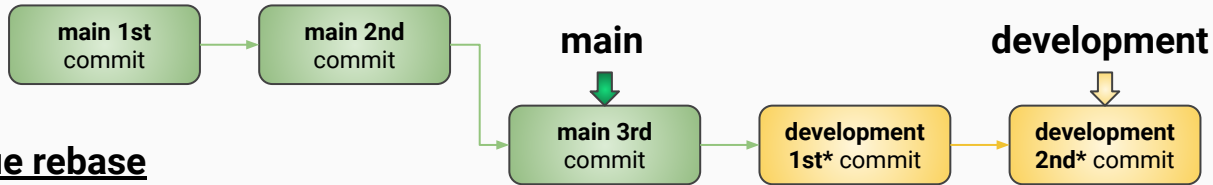
Команда **rebase** - перебазирование веток (практика)

ИЗНАЧАЛЬНОЕ СОСТОЯНИЕ



```
git switch development  
git rebase main
```

СОСТОЯНИЕ ПОСЛЕ REBASE



```
git switch main  
git merge development
```

ФИНАЛЬНОЕ СОСТОЯНИЕ



Команда **rebase -i** - перебазирование для переименования коммита (reword)

- Может получиться, что в описании какого-то давнего коммита мы допустили ошибку и хотим ее исправить. В этом случае нам на помощь опять команда **rebase** с параметром **-i**. После команды и параметра надо указать хэш коммита, который предшествует коммиту, требующему исправления:

```
git rebase -i ee7d9fc183f111edc51b30198eda11c33a7d08eb
```

- После этого в окне текстового редактора откроется файл со списком коммитов для переименования. Перед нужным коммитом вместо слова **pick** впишем слово **reword**, а затем сохраним файл и выйдем из него. Откроется новый файл, где будет предложено переименовать коммит - мы сможем вписать новое описание, затем сохраним файл и закроем его. В результате описание коммита обновится.
- Важно помнить, что все хэш обновленного коммита и все хеши коммитов, расположенных после него, будут обновлены - это значит, что этот вид и все другие виды интерактивного перебазирования нельзя применять внутри веток, на которые ссылаются другие ветки.

Команда **rebase -i** - перебазирование для объединения коммитов (squash)

- Часто можно столкнуться со следующей ситуацией - мы создали новую ветку, создали одним большим коммитом некое обновление, а затем совершили много мелких правок для этого изменения. История не будет выглядеть красиво, если мы вставим в основную ветку все эти хаотичные манипуляции с кодом. Здесь опять можно опять применить **git rebase** с параметром **-i**, после которого нам надо указать коммит, который следует перед первым коммитов в нашей ветке:

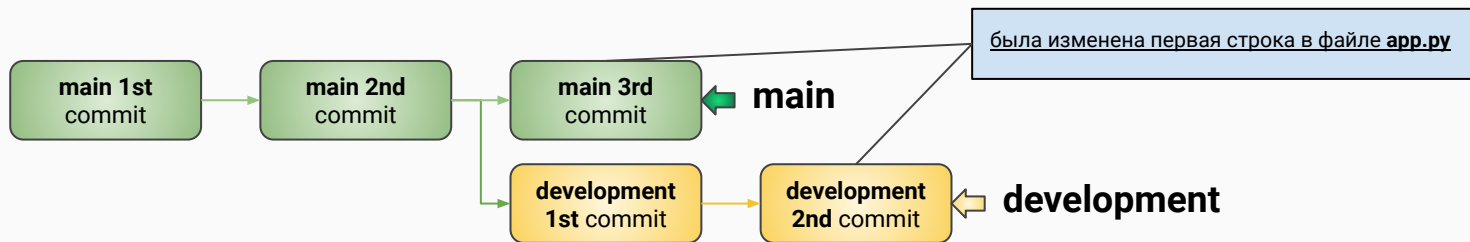
```
git rebase -i 551425990123d4cf165fb6a285a791124cbb0c5f
```

- После этого в окне текстового редактора закономерно откроется файл со списком коммитов. Перед всеми коммитами кроме первого вместо слова **pick** впишем слово **squash**, а затем сохраним файл и выйдем из него. Откроется новый файл, где будет предложено назвать наш новый объединенный коммит. Подставим необходимое название, сохраним файл и выйдем из него. В качестве результата получим один большой новый коммит вместо множества маленьких.

Разрешение конфликтов при слиянии веток

Конфликт при выполнении команды **merge** (введение)

- Представим себе, что при объединении (**merge**) двух веток посредством команды **merge** (и стратегии **ort**) выяснилось, что в новых коммитах **main** и **development** были изменены одни и те же строки в одних и тех же файлах. В этом случае Git не сможет понять, что именно оставить в результате и вернет ошибку.



- Помимо возвращения ошибки, Git изменит проблемные файлы так, что в них будут представлены оба варианта конфликтующих строк:

```
<<<<<< HEAD
вариант конфликтной строки из ветки main
=====
вариант конфликтной строки из ветки development
>>>>>> development
```

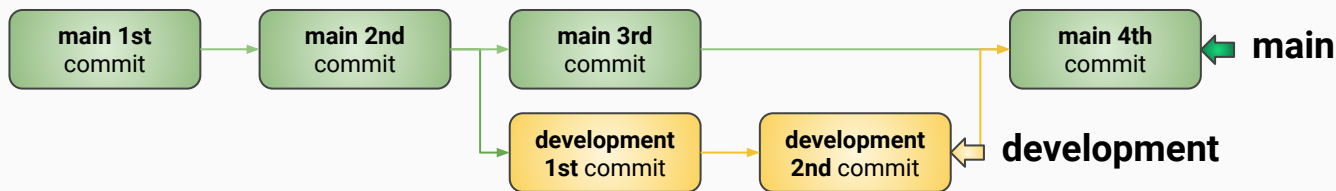
Конфликт при выполнении команды **merge** (варианты разрешения)

- Первый вариант, который можно использовать при разрешении конфликта - использование команды **git merge --abort**, после которой состояние репозитория вернется в состояние до объединения. После этого можно сбросить (**reset --hard**) определенные коммиты из какой-либо ветки, восстановить неконфликтующее содержимое, а затем попытаться объединить ветки вновь:

```
git merge --abort
```

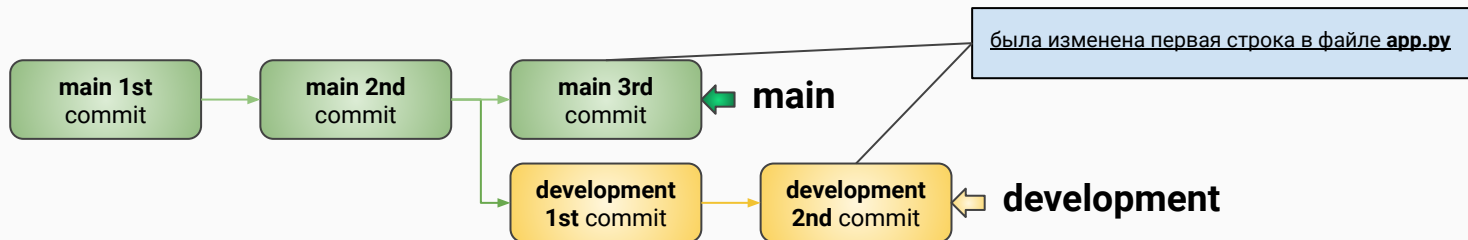
- Второй вариант - после того, как Git сообщил об ошибке и изменил файлы (измененные файлы можно просмотреть командой **git status**), следует обойти каждый конфликтный файл, и внутри исправить код так, чтобы он нас устраивал в конечном результате. Затем добавить измененные файлы в индекс (**git add .**) и сделать коммит, который зафиксирует финальное состояние ветки:

```
git add .  
git commit -m "resolved conflicts"
```

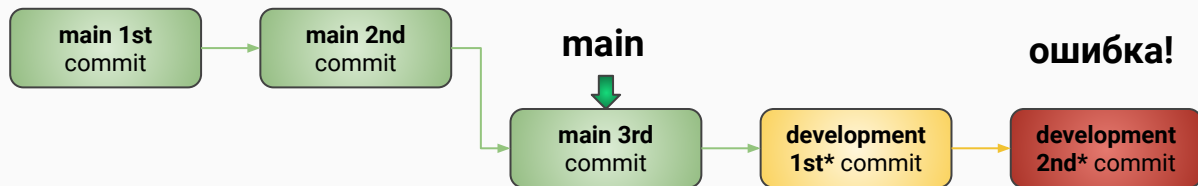


Конфликт при выполнении команды **rebase** (введение)

- При выполнении команды **rebase** у нас также могут возникнуть проблемы, если в разных ветках были произведены изменения в одних и тех же файлах. Представим такую же ситуацию, какая была в предыдущем случае:



- При выполнении **git rebase main**, коммит **main 3rd** перенесется в **development**, коммит **development 1st** получит измененный хэш, но будет обработан успешно, однако при обработке **development 2nd** произойдет ошибка (и изменятся проблемные файлы - будут предложены 2 варианта) - Git поймет, что он содержит изменения, которые касаются строк, уже измененных в **main 3rd**:



Конфликт при выполнении команды **rebase** (варианты разрешения)

- При каждом конфликтующем коммите **development** Git будет останавливаться и предлагать варианты решения. Первый вариант - вернуться в первоначальное состояние до перебазирования. Это осуществляется следующей командой:

```
git rebase --abort
```

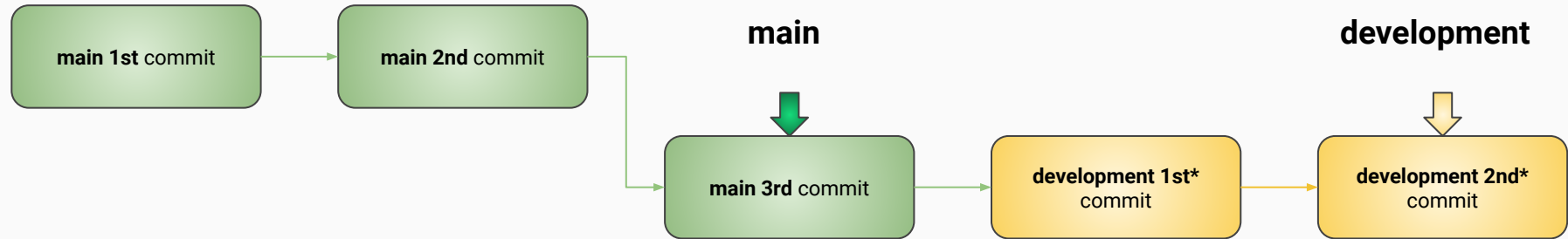
- Следующий вариант - полностью проигнорировать конфликтующий коммит из ветки **development** - т.е. останутся изменения из ветки **main**. Для это надо запустить следующую конструкцию:

```
git rebase --skip
```

- Последний вариант - при отображении ошибки, мы с помощью **git status** смотрим конфликтные файлы, вручную их исправляем, индексируем (**git add .**) а затем запускаем команду **git rebase --continue**. Нам в текстовом редакторе будет предложено написать новое описание коммита - мы сделаем это, потом сохраним его и выйдем из редактора. В результате конфликт будет разрешен:

```
git add .  
git rebase --continue
```

Конфликт при выполнении команды **rebase** (состояние после разрешения)



Основы работы с удаленными репозиториями

Введение

- Мы уже достаточно много знаем о работе с локальным репозиторием, однако есть одна проблема. Дело в том, что если репозиторий находится у нас на компьютере, другие разработчики не смогут вносить изменения в наш репозиторий, создавать отдельные ветки, сливать и перебазировать ветки и т.д. - короче говоря, в этом случае у нас не получится командной разработки.
- Решение обозначенной проблемы очевидно - использование не локального, а удаленного репозитория. В данном случае данные будут храниться в специальной директории на каком-то сервере, мы получим возможность подключаться к этому серверу, загружать из него изменения, сделанные другими разработчиками, отправлять на сервер свои изменения - другими словами говоря, работать вместе с другими людьми.
- Есть множество специальных сайтов и сервисов для размещения удаленных репозиторий (Github, Bitbucket, Gitlab), кроме того, мы можем даже создать свой сервер для командной разработки. Конечно, в процессе обучения полноценный сервер создавать не будем, однако в образовательных целях реализуем концепцию удаленного репозитория на нашем собственном компьютере!

Создание удаленного репозитория на своем компьютере

- Для начала мы должны создать пустой (**--bare**) репозиторий в какой-либо директории у нас на компьютере - например, сделаем это в **/home/user/remote** (если такой директории не существует, то ее надо создать). Это и будет наш условный удаленный репозиторий:

```
git init --bare
```

- Теперь создадим локальный репозиторий в какой-либо другой директории, например, в **/home/user/local** (опять же - если ее нет, следует ее создать):

```
git init
```

- Далее нам надо связать локальный репозиторий с нашим удаленным репозиторием. Это осуществляется при помощи команды **git remote add** (внутри локального репозитория), после которой надо указать название удаленного репозитория (название может быть любым, но обычно используют слово **origin**), а после этого следует вставить ссылку или путь к удаленному репозиторию:

```
git remote add origin /home/user/remote
```

Просмотр и модификация существующих удаленных репозиториев

- Для просмотра списка доступных удаленных репозиториев (их можно добавлять в неограниченном количестве) используется следующая команда:

```
git remote -v
```

- Если мы хотим посмотреть ссылки только по одному репозиторию, то нужно использовать команду **git remote show**, а после нее - название репозитория:

```
git remote show origin
```

- Если необходимо изменить ссылку на репозиторий, необходимо применить команду **git remote set-url**, потом название репозитория, а потом - новую ссылку:

```
git remote set-url origin /home/user/remote
```

- Если же мы хотим отключить ссылку для вставки данных в удаленный репозиторий, то следует использовать следующий подход:

```
git remote set-url --push origin DISABLE
```

- Переименование имени удаленного репозитория (из **origin** в **destination**):

```
git remote rename origin destination
```

Подключение веток к удаленному репозиторию

- Само по себе подключение к удаленному репозиторию не имеет значения без подключения веток. Для начала создадим в нашем локальном репозитории (/home/user/local) файл **app.py** и первый коммит, который, как мы помним, будет автоматически вставлен в ветку **main** (настройка **init.defaultBranch**):

```
touch app.py
git add .
git commit -m "initial commit"
```

- Теперь настало время заняться непосредственно загрузить нашу ветку **main** со всем содержимым в удаленный репозиторий **origin**. Это осуществляется следующим образом:

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

- Если бы в удаленном репозитории **origin** уже была бы ветка **main**, то мы могли бы не загружать туда нашу ветку, а просто установить связь. Это можно сделать так:

```
git branch --set-upstream-to=origin/main main
```


Команда **clone** - клонирование удаленного репозитория

- Когда удаленный репозиторий настроен и в него уже загружены какие-либо ветки (через первый подключенный в нему локальный репозиторий), другие репозитории могут начать пользоваться этим удаленным репозиторием без явного привязывания ссылок и веток. Это осуществляется с помощью команды **git clone**, после которой идет путь или ссылка к удаленному репозиторию - в этом случае привязка произойдет автоматически. Создадим в нашей файловой структуре директорию **/home/user/additional-local** и клонируем туда удаленный репозиторий:

```
# создание директории для нового репозитория  
mkdir /home/user/additional-local  
# переход в директорию нового репозитория  
cd /home/user/additional-local  
# клонирование удаленного репозитория в новую директорию  
git clone /home/user/remote .
```

В последнем выражении в конце стоит точка - это сделано для того, чтобы репозиторий клонировался именно в директорию, где мы находимся. Если точку не поставить, Git создаст новую директорию с названием репозитория (в нашем случае - **remote**) и расположит клонированные файлы уже там.

Команда **push** - загрузка данных в удаленный репозиторий

- На данный момент у нас есть один удаленный репозиторий **/home/user/remote** и два условно локальных - **/home/user/local** и **/home/user/additional-local**. Теперь представим, что в первом локальном репозитории был добавлен файл **utils.py** - как нам его добавить в удаленный репозиторий, чтобы изменения были доступны другому локальному репозиторию? Это осуществляется одной командой - **git push**, которая автоматически загрузит все изменения в **remote**:

```
# переход в директорию с локальным репозиторием  
cd /home/user/local  
# создание файла utils.py  
touch utils.py  
# добавление файла utils.py в индекс  
git add utils.py  
# создание нового коммита  
git commit -m "added utils.py"  
# загрузка всех новых коммитов в удаленный репозиторий  
git push
```

Команда **pull** - выгрузка данных из удаленного репозитория

- У нас сложилась следующая ситуация - глобальный удаленный репозиторий **remote** и локальный репозиторий **local** полностью синхронизированы (хранят по 2 коммита), но локальный репозиторий **local-additional** отстает на один коммит. Чтобы в последний репозиторий подгрузить недостающие коммиты с изменениями из удаленного репозитория, следует выполнить всего одну маленькую команду - **git pull**:

```
# переход в директорию с локальным репозиторием  
cd /home/user/local-additional  
# выгрузка данных из удаленного репозитория в локальный  
git pull
```

Команда fetch - выгрузка данных из удаленного репозитория без слияния

- Иногда нам может потребоваться просто узнать, есть ли какие-то обновления в удаленном репозитории, одновременно при этом применять эти обновления в локальном репозитории. Для этого используется команда **git fetch**:

```
# переход в директорию с локальным репозиторием
```

```
cd /home/user/local-additional
```

```
# выгрузка данных из удаленного репозитория без обновления
```

```
git fetch
```

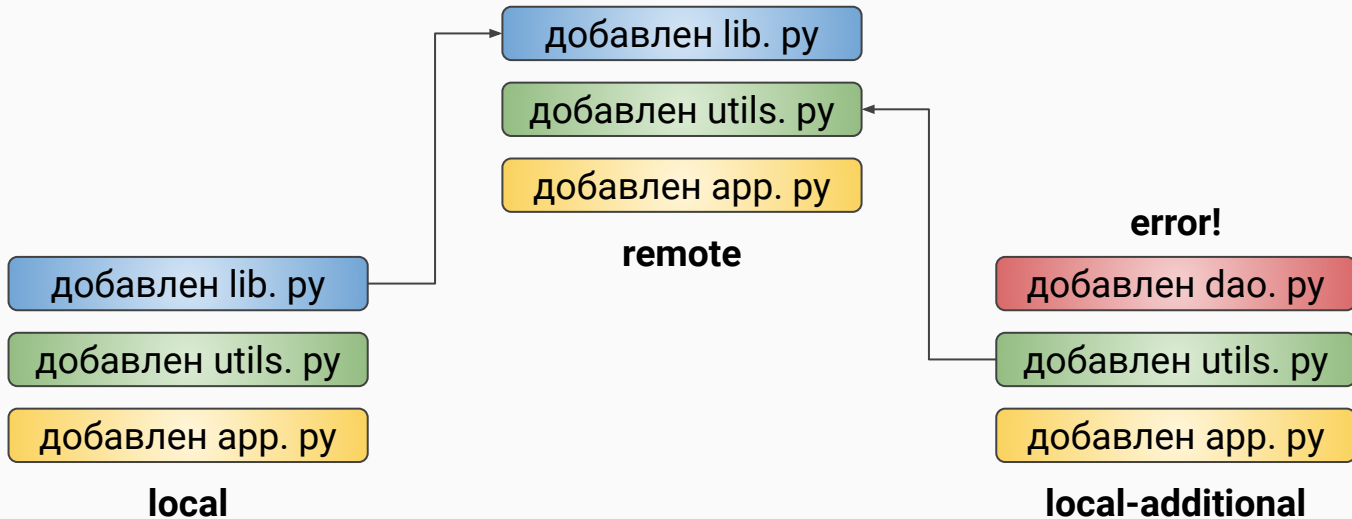
```
# показать отличия между локальным и удаленным репозиторием
```

```
git show
```

Разрешение конфликтов при
работе с удаленным репозиторием

Введение в конфликт историй

- Представим, что у нас есть все те же два локальных репозитория (**local** и **local-additional**) и один удаленный (**remote**). Везде хранится два коммита - с добавлением **app.py** и **utils.py**. Теперь предположим, что в репозитории **local** был добавлен файл **lib.py**, сделан коммит и загружен в удаленный репозиторий. В свою очередь в репозитории **local-additional** был добавлен файл **dao.py**, также сделан коммит и произошла попытка сделать коммит в удаленный репозиторий (**git push**). Очевидно, что попытка закончится ошибкой (истории различаются локально и удаленно) и далее мы разберем возможности решения этой ошибки:



Разрешение конфликта историй посредством слияния (**merge**)

- Самое главное, что мы вначале должны сделать - подгрузить изменения из удаленного репозитория. Это можно осуществить при помощи команды **fetch** или при помощи команды **pull**:

```
# первый вариант подгрузки изменений  
git fetch
```

```
# второй вариант подгрузки изменений  
git pull
```

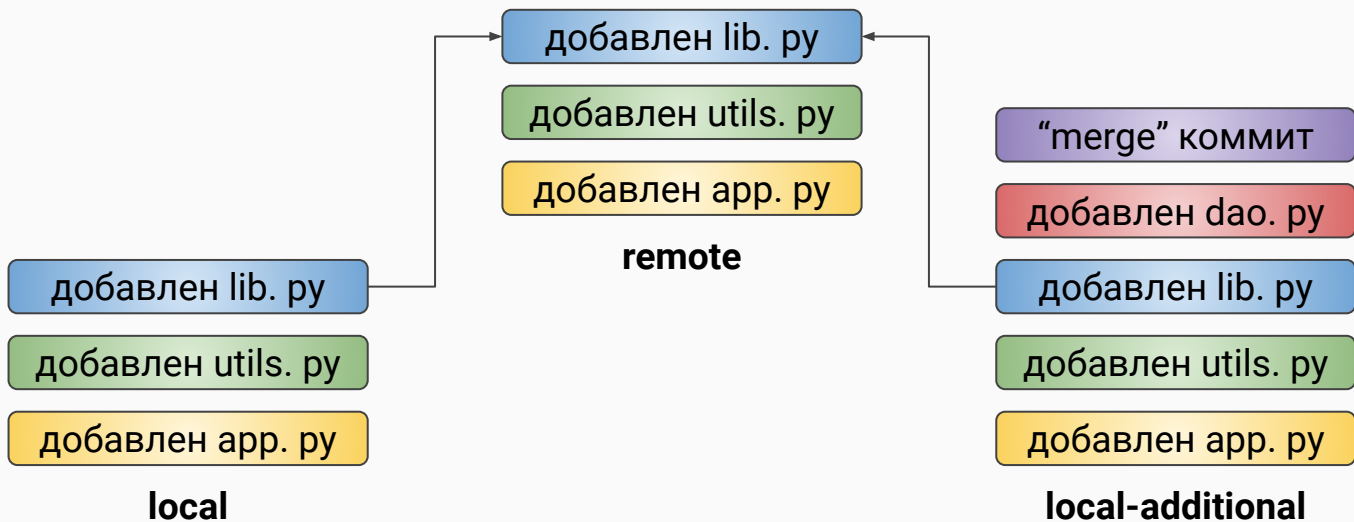
- Далее следует выполнить команду **git merge** - она подставит нужные коммиты в наш локальный репозиторий, но сделает дополнительный объединяющий коммит и потребует ввести его имя в текстовом редакторе.

```
git merge
```

- Данный процесс можно настроить заранее (для **git pull** без всяких параметров) - если мы установим параметр **pull.rebase** в значение **false**, то при выполнении команды **git pull** сразу будет сделан объединяющий коммит с предложением ввести его описание:

```
git config pull.rebase false
```

Состояние репозитория после разрешения конфликта историй (merge)



Разрешение конфликта историй посредством перебазирования (**rebase**)

- Если мы хотим воспользоваться перебазированием, то мы точно также вначале должны сделать команду `git pull` с параметром `--rebase`, но без дополнительных аргументов:

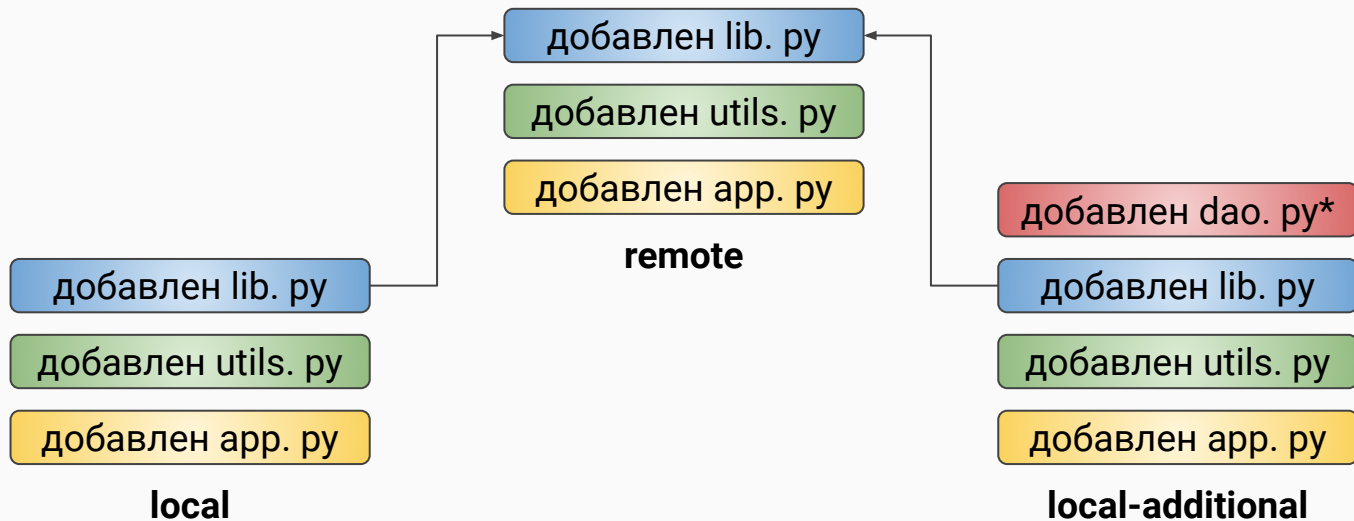
```
git pull --rebase
```

Эта команда подставит коммиты из удаленного репозитория перед конфликтующими коммитами локального репозитория, а у коммитов локального репозитория изменит хэши. Никакого объединяющего коммита не потребуется. Это следует делать только в том случае, если на ветку локального репозитория не ссылаются другие ветки.

- Эту логику также можно настроить заранее (для **git pull** без всяких параметров) - для этого надо подставить значение `true` для настройки репозитория **pull.rebase**:

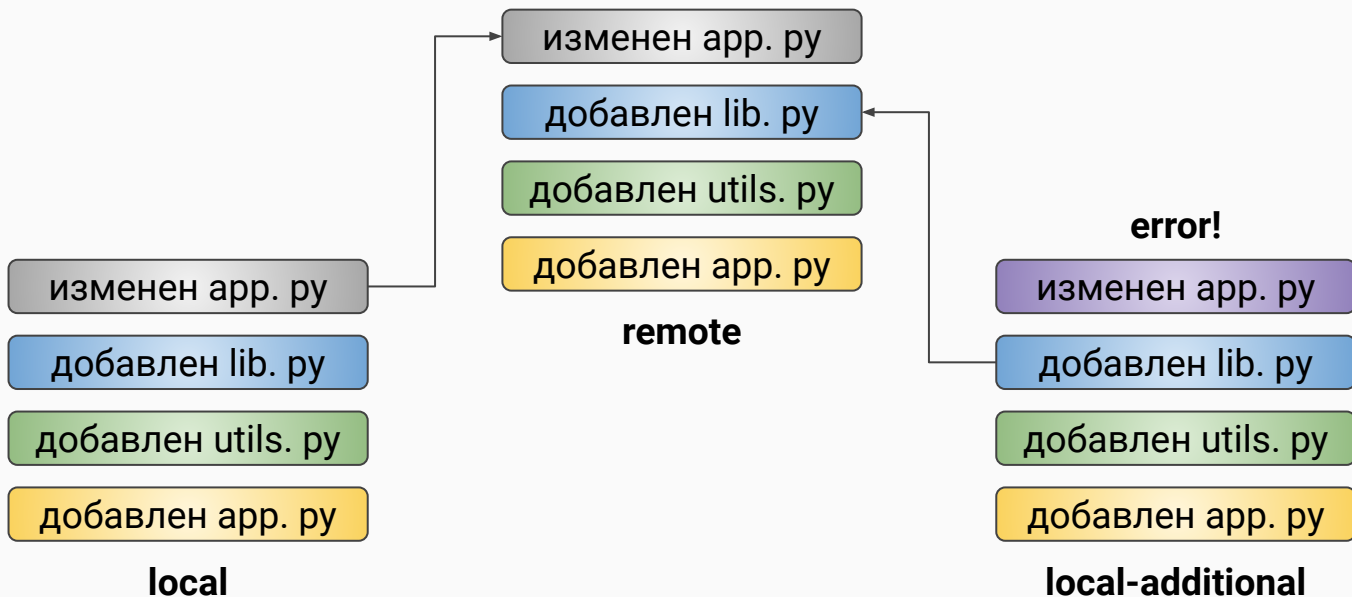
```
git config pull.rebase true
```

Состояние репозитория после разрешения конфликта историй (rebase)



Введение в конфликт изменений

- Представим себе другой вид конфликта - репозиторий **local** изменил первую строку файла **app.py** и загрузил изменения в удаленный репозиторий **remote**. Репозиторий **local-additional** также изменил первую строку файла **app.py** и попытался загрузить изменения в **remote**, что закономерно кончилось ошибкой. Т.е. у нас не только несовместимая история, но и несовместимы изменения:



Разрешение конфликта изменений посредством слияния (merge)

- Для решения конфликта посредством слияния мы опять должны выполнить уже знакомую нам команду для получения данных - **git pull**:

```
git pull
```

Далее выполняем также знакомую нам команду **git merge** для слияния данных:

```
git merge
```

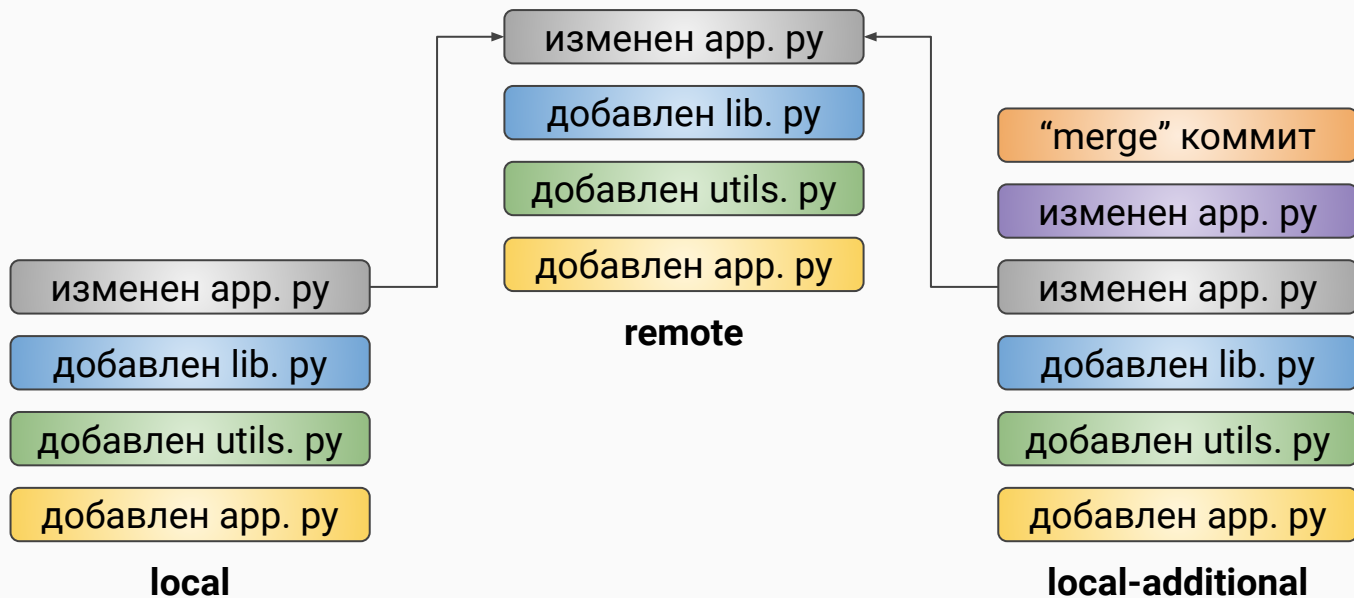
- Удаленные коммиты будут успешно подгружены, но объединяющий коммит не произойдет, т.к. есть конфликтующие изменения. Вместо этого Git изменит проблемные файлы, предложив нам выбрать один из вариантов. Мы определим при помощи **git status** все нужные файлы, вручную разрешаем конфликты, а затем добавляем всю нашу работу в индекс:

```
git add .
```

- После этого выполняем команду **git commit -m**, в пояснении к которой объясняем, что мы сделали:

```
git commit -m "resolved pull conflict by using merge commit"
```

Состояние репозитория после разрешения конфликта изменений (merge)



Разрешение конфликта изменений посредством перебазирования (**rebase**)

- Для решения конфликта посредством перебазирования сначала выполним команду **git pull** с параметром **--rebase**:

```
git pull --rebase
```

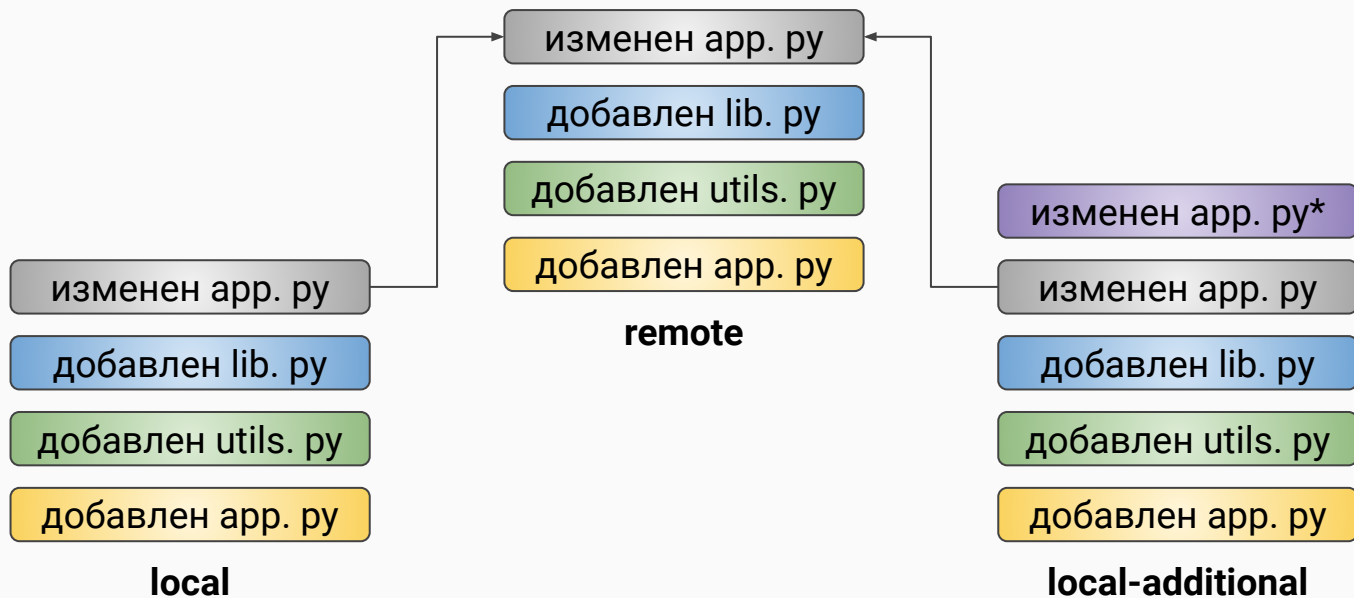
- Далее произойдет закономерный конфликт, при котором Git изменит файлы, предложив их исправить. При помощи команды **git status** находим эти файлы, исправляем их и заносим в индекс (с помощью команды **git add .**).

```
git add .
```

- Затем вводим команду **git rebase --continue**, после этого перед нами появится предложение переименовать коммит, который мы только что исправили. Если надо - исправляем, далее закрываем редактор. В результате конфликт будет разрешен, единственная потенциальная угроза - хэш проблемного коммита будет перезаписан:

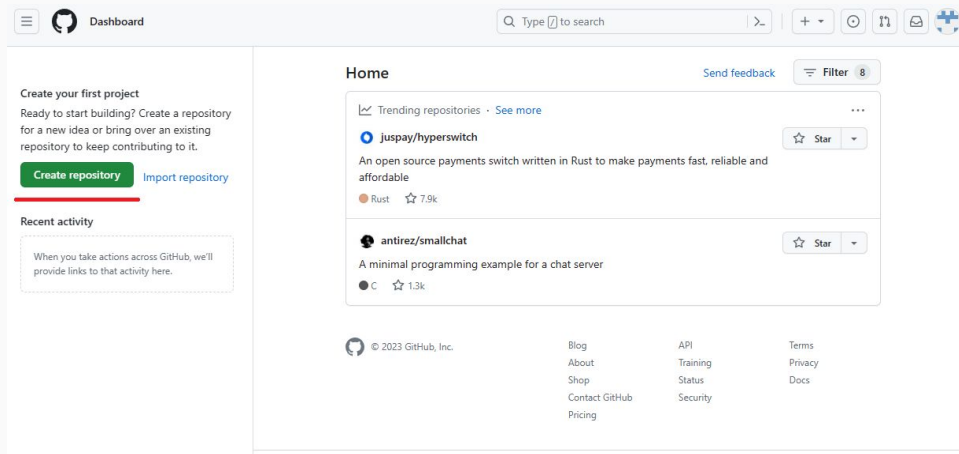
```
git rebase --continue
```

Состояние репозитория после разрешения конфликта изменений (rebase)



Подключение репозиториев к GitHub

- Поддержка своего удаленного репозитория требует специфических знаний и специальной инфраструктуры. Это может быть выгодно и даже необходимо для больших корпораций, но для большинства случаев подойдут уже готовые решения, которые во множестве представлены в сети Интернет. Самым популярным решением является сайт GitHub - <https://github.com/> - он позволяет бесплатно создавать и хранить свои репозитории, создавать специальные политики для работы с репозиториями и ветками, а также многое другое.
- После процедуры регистрации мы попадем на главную страницу нашего профиля, где нам будет предложено создать новый репозиторий:



Создание нового удаленного репозитория

- На странице создания репозитория обязательным условием является установка названия (**Repository name**) репозитория - он должен быть уникальным в контексте профиля. Также важно выбрать, будет ли репозиторий публичным или приватным (будет ли он виден только его хозяину или всем). Затем есть три менее важные опции - можно добавить файл с описанием (**README.md**), файл с неотслеживаемыми файлами (**.gitignore**) и файл с типовой лицензией.
- Для подтверждения создания репозитория следует нажать зеленую кнопку **“Create repository”**.

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 *  GreenBerret / Repository name *

✔ example is available.

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

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: **None**

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.](#)


 You are creating a public repository in your personal account.

Create repository

Подключение удаленного репозитория к локальному репозиторию

- После создания нашего удаленного репозитория, мы автоматически попадаем на страницу этого репозитория (который пока является пустым) и видим варианты его подключения к локальному репозиторию на нашем компьютере. С удивлением можно обнаружить, что все эти способы нам уже знакомы, когда мы имитировали удаленный репозиторий на своем компьютере - только вместо пути к директории нам предлагается использовать ссылку на сайт GitHub:

Quick setup — if you've done this kind of thing before

 Set up in Desktop

 or

HTTPS

SSH

<https://github.com/GreenBerret/example.git>

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# example" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/GreenBerret/example.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/GreenBerret/example.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

Проблема с аутентификацией на GitHub

- Если мы пойдем по какому-либо пути, предложенному на прошлой странице, нас будет ожидать очень интересная вещь - будет запрошен логин и пароль от GitHub, и даже если мы введем все правильно, нам будет отказано в доступе. Дело в том, что с августа 2021 года GitHub не разрешает использовать традиционный подход для аутентификации. Вместо того на данный момент нам доступны другие опции (перечислим самые популярные):
 - генерация классического токена (кода) доступа
 - генерация гранулированного токена (кода) доступа
 - использование SSH для аутентификации

Генерация классического токена (кода) доступа (инструкция)

- Для генерации классического токена (кода доступа) следует перейти в профиль (кнопка в верхнем правом углу), затем зайти в раздел **“Settings”**, после этого слева внизу в раздел **“Developer Settings”**, затем выбираем **“Personal access tokens”** и вариант **“Tokens (classic)”**. В открывшемся разделе даем название токenu, устанавливаем срок годности (можно бессрочно), а также из всех опций помечаем те, которые находятся в разделе **repo** (это минимальная набор для получения доступа). Потом подтверждаем создание, нажав зеленую кнопку внизу страницы. Нам отобразится код, который надо сразу же скопировать, т.к. в явном виде он больше не будет доступен.
- Теперь при привязке локальных и удаленных репозиторий вместо ссылки: <https://github.com/ПРОФИЛЬ/РЕПОЗИТОРИЙ.git> используем ссылку: <https://TOKEN@github.com/ПРОФИЛЬ/РЕПОЗИТОРИЙ.git>
- Данный подход предоставит право доступа ко всем репозиториям, которые доступны создателю этого токена.

Генерация классического токена (кода) доступа (вид раздела)

GitHub Apps

OAuth Apps

Personal access tokens

Fine-grained tokens

Tokens (classic)

Beta

New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

ACCESS_TOKEN

What's this token for?

Expiration *

90 days

The token will expire on Wed, Jan 31 2024

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events

Генерация гранулированного токена (кода) доступа (инструкция)

- Для генерации гранулированного токена следует перейти в профиль (кнопка в верхнем правом углу), затем зайти в раздел **“Settings”**, после этого слева внизу в раздел **“Developer Settings”**, затем выбираем **“Personal access tokens”** и вариант **“Fine-grained tokens”**. В открывшемся разделе даем название токenu, устанавливаем срок годности (бессрочно нельзя, только конкретный срок), а также помечаем те репозитории, для которых мы хотим разрешить доступ. Затем в подразделе **“Repository permissions”** для опции **“Contents”** назначаем значение **“Read and write”**, а для опции **“Metadata”** - значение **“Read only”**. Потом подтверждаем создание, нажав зеленую кнопку внизу страницы. Нам отобразится код, который надо сразу же скопировать, т.к. в явном виде он больше не будет доступен.
- Теперь при привязке локальных и удаленных репозиторий вместо ссылки: <https://github.com/ПРОФИЛЬ/РЕПОЗИТОРИЙ.git> используем ссылку: <https://ТОКЕН@github.com/ПРОФИЛЬ/РЕПОЗИТОРИЙ.git>

Генерація гранулированного токена (кода) доступа (вид раздела)

88 GitHub Apps

OAuth Apps

Personal access tokens

Fine-grained tokens Beta

Tokens (classic)

New fine-grained personal access token Beta

Create a fine-grained, repository-scoped token suitable for personal API use and for using Git over HTTPS.

Token name *

A unique name for this token. May be visible to resource owners or users with possession of the token.

Expiration *

30 days ▾

The token will expire on Sat, Dec 2, 2023

Description

What is this token for?

Resource owner

GreenBerret ▾

Repository access

☐ Public Repositories (read-only)

☐ All repositories
This applies to all current and future repositories owned by the resource owner.
Also includes public repositories (read-only).

☒ Only select repositories
Select at least one repository. Max 50 repositories.
Also includes public repositories (read-only).

Select repositories ▾

Selected 1 repository.

GreenBerret/example ×

Permissions

Read our [permissions documentation](#) for information about specific permissions.

Repository permissions 2 Selected ▾

Repository permissions permit access to repositories and related resources.

Использование SSH для аутентификации (инструкция)

- Если мы не хотим возиться с ключами и иметь доступ ко всем репозиториям, можно попробовать воспользоваться публичным ключем SSH. Сначала генерируем его на нашем компьютере:

```
ssh-keygen
```

Затем идем в наш профиль на GitHub, переходим в **“Settings”**, затем выбираем раздел **“SSH and GPG keys”** и нажимаем на кнопку **“New SSH key”**. В поле **“Title”** задаем название нашего подключения, в поле **“Key”** копируем публичный ключ с нашего компьютера (находится в каталоге пользователя во внутреннем каталоге **.ssh**). Затем нажимаем зеленую кнопку **“Add SSH key”**.

- Теперь при привязке локальных и удаленных репозиторий мы будем использовать следующую ссылку:
git@github.com:ПРОФИЛЬ/РЕПОЗИТОРИЙ.git

Использование SSH для аутентификации (вид раздела)

Public profile

Account

Appearance

Accessibility

Notifications

Access

Billing and plans

Emails

Password and authentication

Sessions

SSH and GPG keys

Organizations

Enterprises

Moderation

Code, planning, and automation

Repositories

Codespaces

Packages

Copilot

Pages

Saved replies

Add new SSH Key

Title

Key type

Authentication Key

Key

Begin with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'

Add SSH key

Оформление Pull Request

Создание копии репозитории (форка)

- Иногда на просторах GitHub мы можем встретить какой-либо чужой репозиторий, который привлечет наше внимание. В результате мы захотим внести какие-то дополнительные изменения, но, как можно понять, это будет запрещено - ведь репозиторий то чужой!
- Однако GitHub предлагает решение этой проблемы, которое состоит в создании т.н. форков - по сути, независимых копий чужих репозиториях, в которых можно вести свою собственную разработку на основе первичного репозитория.
- Для создания форка нам необходимо перейти на страницу интересующего нас чужого репозитория, в правом верхнем углу рядом с надписью **"Fork"** нажать треугольник для открытия специального меню, а затем в этом меню выбрать пункт **"Create a new fork"**. На открывшейся странице можно ввести новое имя для этого репозитория, добавить какое-то описание, а также установить так, чтобы была скопирована только главная ветка, а не все ветки (этого мы делать не будем). Затем надо подтвердить создание форка нажатием зеленой кнопки **"Create fork"**.

Создание копии репозитории (вид раздела)

Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Required fields are marked with an asterisk ().*

Owner *



Repository name *

/ example

✔ example is available.

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

☐ Copy the **main** branch only

Contribute back to GreenBerret/example by adding your own branch. [Learn more.](#)

 You are creating a fork in your personal account.

Create fork

Создание Pull Request

- Теперь мы можем делать коммиты в нашей копии репозитория и пользоваться копией по нашему усмотрению. Однако у нас есть возможность предложить авторам оригинального репозитория вставить наши изменения в себе - т.е. мы можем создать т.н. **pull request**.
- Для создания **pull request** нам необходимо перейти в наш скопированный репозиторий, в настройках репозитория выбрать опцию **“Contribute”**, а затем нажать на кнопку **“Open pull request”**. В открывшемся разделе мы сможем указать ветку нашего и первоначального репозитория, вписать название нашего **pull request** и добавить описание, а затем, чтобы окончательно создать запрос, нажать большую зеленую кнопку **“Create pull request”**.
- Владелец оригинального репозитория получит сообщение о том, что мы предлагаем ему наши изменения. Затем он сможет их просмотреть и решить - нужны они ему или нет. Если решение будет положительным (это осуществляется нажатием одной кнопки), то наши коммиты успешно попадут в чужой репозиторий.

Создание Pull Request (вид раздела)



Add a title

Title

Helpful resources

[GitHub Community Guidelines](#)

Add a description

Write

Preview

H B I | | @

Add your description here...

☒ Allow edits by maintainers

Create pull request



Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

1 commit

1 file changed

1 contributor

Ограничение доступа к веткам на GitHub

Ограничение доступа (инструкция)

- При разработке мы не всегда должны разрешать всем разработчика вносить изменения в определенную ветку - как правило, в ветку **main**. Сама разработка и тестирование обычно производится в ветке **development**, а ветка **main** предназначена для выпуска финальной версии продукта.
- Github предоставляет все возможности для такого подхода - для этого стоит перейти на страницу репозитория, потом зайти в его настройки ("**Settings**"), а после этого слева выбрать пункт "**Branches**". В этом разделе нам будет предложено создать правило для ограничения пользования веткой, что мы и сделаем, нажав кнопку "**Add branch protection rule**".
- В открывшемся окне находим поле "**Branch name pattern**" и вписываем туда название нужной нам ветки - "**main**". Чуть ниже выбираем то, что именно будем ограничивать - как правило, выбирается **pull request** для внесения изменений в ветку "**Require a pull request before merging**" и запрет владельцам репозитория обходить назначенные правила "**Do not allow bypassing the above settings**". После этого подтверждаем создание, нажав зеленую кнопку "**Create**" внизу страницы.

Ограничение доступа (вид раздела)

General

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Rules

Actions

Webhooks

Environments

Codespaces

Pages

Security

Code security and analysis

Deploy keys

Secrets and variables

Integrations

GitHub Apps

Email notifications

Branch protection rule

Protect your most important branches

[Branch protection rules](#) define whether collaborators can delete or force push to the branch and set requirements for any pushes to the branch, such as passing status checks or a linear commit history.

Your [GitHub Free plan](#) can only enforce rules on its public repositories, like this one.

Branch name pattern *

Protect matching branches

☒ **Require a pull request before merging**

When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

☒ **Require approvals**

When enabled, pull requests targeting a matching branch require a number of approvals and no changes requested before they can be merged.

Required number of approvals before merging: 1

☐ **Dismiss stale pull request approvals when new commits are pushed**

New reviewable commits pushed to a matching branch will dismiss pull request review approvals.

☐ **Require review from Code Owners**

Require an approved review in pull requests including files with a designated code owner.

☐ **Require approval of the most recent reviewable push**

Whether the most recent reviewable push must be approved by someone other than the person who pushed it.

☐ **Require status checks to pass before merging**

Choose which [status checks](#) must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

☐ **Require conversation resolution before merging**

When enabled, all conversations on code must be resolved before a pull request can be merged into a branch that matches this rule. [Learn more about requiring conversation completion before merging.](#)

☐ **Require signed commits**

Commits pushed to matching branches must have verified signatures.

☐ **Require linear history**

Prevent merge commits from being pushed to matching branches.

☐ **Require deployments to succeed before merging**

Choose which environments must be successfully deployed to before branches can be merged into a branch that matches this rule.

Внесение изменений в ветку

- Теперь изменения в ветке **main** можно произвести двумя способами:
 - Во-первых, можно сделать копию (форк) первичного репозитория (если он является публичным - **public**), изменить ветку **main** внутри копии и предложить **pull request** внутри первичного репозитория.
 - Во-вторых, в первичном репозитории можно сделать новую ветку (если у нас есть на это право), например, **development**, затем произвести изменения в ней, а потом запросить **pull request** уже в контексте объединения этой новой ветки с веткой **main**.