



# > Конспект > 8 урок > Версионирование кода и git

## >Оглавление 8 урока

### >Оглавление 8 урока

#### > Версионирование кода. Введение в git.

##### > Особенности git

##### > Документация

#### > Установка Git

#### > Работа с git

##### > Стадии работы с git

##### > Работа с git в командной строке

#### > Ветки и теги

##### > Создание веток и переключение между ветками

##### > Сравнение изменений между двумя коммитами

##### > Перемещение по коммитам

##### > Объединение изменений

#### > Стратегии ветвления

##### > Подход Git Flow

##### > Плюсы Git Flow

##### > Минусы Git Flow

##### > Подход Trunk based development

##### > Плюсы Trunk based development

##### > Минусы Trunk based development

#### > Конфликт слияния веток

#### > Работа с удаленными провайдерами

##### > Принципы работы с удаленными репозиториями

##### > Этапы загрузки репозитория на удаленный сервер

> [Доступ на GitHub по SSH](#)  
> [Советы по работе с git](#)  
> [Основные команды терминала](#)  
    > [Работа с Vim](#)  
    > [Работа с Nano](#)  
> [Основные команды git](#)

## > Версионирование кода. Введение в git.

Версионирование кода, то есть отслеживание версий кода, применяется в случае, если программист или команда программистов работают над кодом долгое время, постоянно меняя его. Если разработчик передумал и захотел вернуться к предыдущей версии кода, например, если последняя версия кода содержит ошибку, он может сделать это с использованием системы контроля версий.

Наиболее популярная система контроля версий – **git** – была создана программистом Линусом Торвальдсом в процессе разработки операционной системы Linux.

## > Особенности git

- Следит за всеми файлами и папками в директории.
- Работает локально.
- Оперирует состояниями системы, то есть совокупностью всех файлов и папок.

## > Документация

Документация по git очень хорошо представлена в книге [Git Book](#) на нескольких языках.

## > Установка Git

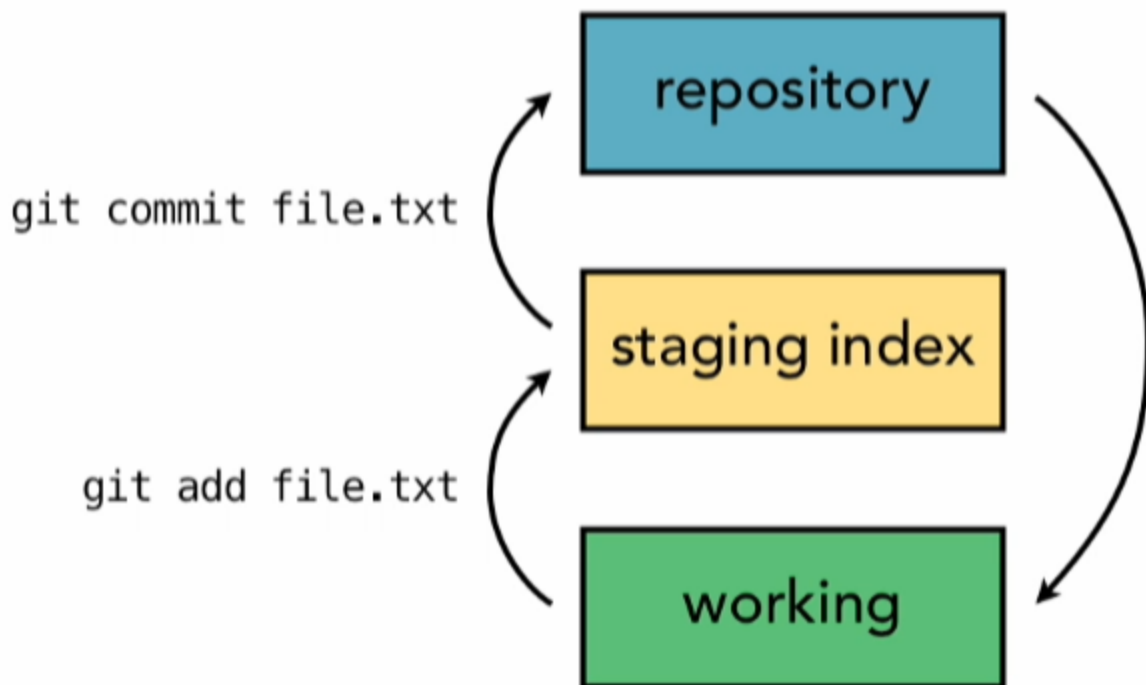
Информацию о том, как установить Git на операционные системы **Window**, **macOS** и **Linux** можно найти [здесь](#).

## > Работа с git

**Коммит** (англ. *commit*) – слепок всех отслеживаемых гитом (индексированных) файлов. Каждый коммит имеет родительский коммит – предыдущее состояние файла, а совокупность всех коммитов формирует дерево коммитов. Название каждого коммита уникально и представляет собой **хэш-сумму** коммита.

## > Стадии работы с git

1. **Редактирование** файла – изменение файла в любом текстовом редакторе.
2. Добавление файла в **staging area** – область с изменениями, которые будут включены в следующий коммит.
3. Сохранение **коммита** – перманентное сохранение всех собранных в **staging area** изменений.



**Схема работы git.** На стадии **working** происходит работа с файлом (создание, изменение, удаление). Затем файл переходит на следующую стадию **staging** - на этой стадии git видит изменения, которые могут войти в следующий

коммит, но их еще можно поменять. На финальной стадии **repository** git навсегда запоминает состояние всех файлов и папок в данный момент времени.

## > Работа с git в командной строке

Для инициализации git в директории, за содержимым которой нужно следить, используется команда `git init`. В результате работы команды создается скрытая папка `.git`, в которой и будут храниться данные о всех коммитах.

```
$ git init
# Initialized empty Git repository in /path/to/git_dir/.git/
$ ls -a
# . .. .git
```

Для того, чтобы добавить изменения в файл в staging area, используется команда `git add`. Для проверки статуса файлов, нужно использовать команду `git --status`.

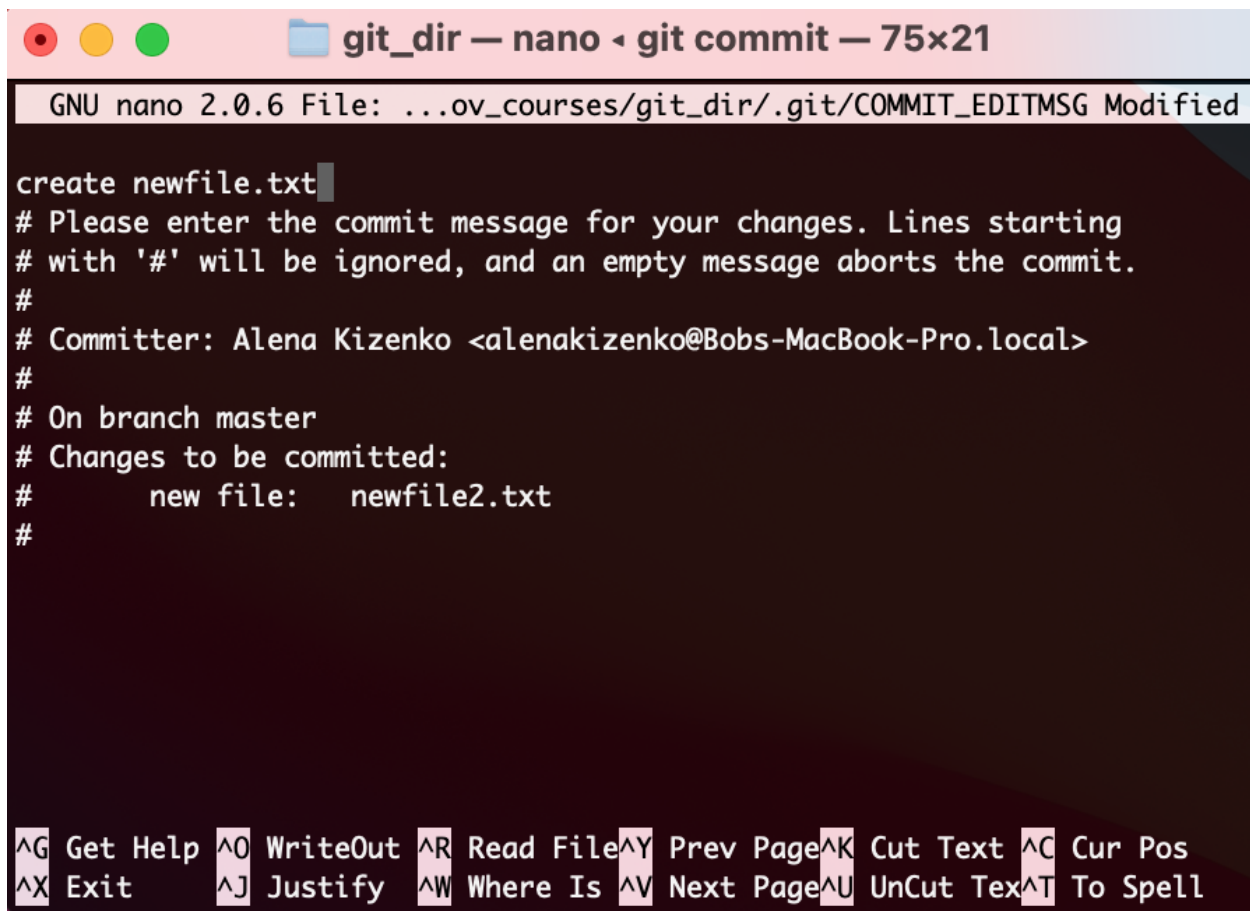
```
$ touch newfile.txt
$ git add newfile.txt
$ git status
# On branch master

# No commits yet

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

Для того, чтобы сделать коммит, используется команда `git commit`. В возникшем окне с текстовым редактором необходимо написать описание коммита в настоящем времени.

```
$ git commit
# 1 file changed, 0 insertions(+), 0 deletions(-)
# create mode 100644 newfile.txt
```



```
git_dir — nano ◀ git commit — 75x21
GNU nano 2.0.6 File: ...ov_courses/git_dir/.git/COMMIT_EDITMSG Modified

create newfile.txt
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
#
# On branch master
# Changes to be committed:
#   new file:   newfile2.txt
#

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is ^V Next Page ^U UnCut Tex ^T To Spell
```

Скриншот из окна текстового редактора для описания коммита

По умолчанию в качестве текстового редактора для git используется **vim** - довольно сложный для вхождения редактор. Чтобы поменять на более простой для новичка редактор, например, **nano**, нужно использовать команду `git config --global core.editor "nano"`.

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

Чтобы показать все коммиты используется команда `git log`; если добавить флаг `-graph`, то будет выведено дерево коммитов.

```
$ git log
# commit 08c823b947d2ff9bee678abbb458a7e147eccc0c (HEAD -> master)
# Author: Alena Kizenko <alena.kizenko@Bobs-MacBook-Pro.local>
# Date: Sun Feb 27 14:19:50 2022 +0100

# create newfile.txt
```

Команда `git diff --staged` выводит изменения, добавленные в **staging area**.

```
$ touch newfile2.txt
$ git add newfile2.txt
$ git diff --staged
# diff --git a/newfile2.txt b/newfile2.txt
# new file mode 100644
# index 0000000..e69de29
```

## > Ветки и теги

**Ветка** (англ. *branch*) – это указатель на коммит с определенным именем; по веткам можно переключаться между коммитами, когда ведется активная разработка программы.

**Тэг** (англ. *tag*) – это тоже указатель на коммит, но, в отличие от ветки, он не изменяется; тэги используются для указания коммита, который находится в релизе.

**HEAD** – это указатель на текущий коммит.

## > Создание веток и переключение между ветками

Используя ветвление, можно отклоняться от основной линии разработки и продолжать работу независимо от неё, не вмешиваясь в основную линию. Например, разработчик пишет код для калькулятора в ветке **master** и решает сделать вывод результата вычислений цветным. Для этого он создаст еще одну ветку **feature** и напишет эту дополнительную функцию, не затрагивая основной код. Далее он будет тестировать и дорабатывать эту функцию (её еще называют "фича") независимо от основного кода.

Создание ветки осуществляется командой `git branch <имя ветки>`. Для переключения между ветками используется команда `git checkout <имя ветки>`. Одновременно создать ветку и переключиться на неё - `git checkout -b <имя ветки>`.

```
$ git checkout -b feature
# Switched to a new branch 'feature'
```

## > Сравнение изменений между двумя коммитами

Команда `git diff <имя одной ветки> <имя другой ветки>` позволяет сравнить изменения между двумя коммитами. Таким образом можно проследить какие изменения файлов и папок потребовались, чтобы получить из одной ветки другую

Здесь в файл **newfile.txt** в ветке feature была добавлена строка *"Learning is great"*.

```
$ git diff master feature
# diff --git a/newfile.txt b/newfile.txt
# index e69de29..9b6ae26 100644
# --- a/newfile.txt
# +++ b/newfile.txt
# @@ -0,0 +1 @@
# +Learning is great
```

## > Перемещение по коммитам

Переместиться в другой коммит можно с использованием команды `git checkout <хэш-сумма коммита>`, предварительно узнав хэш-сумму коммита командой `git log`.

Если нужно перейти на один коммит назад, используется команда `git checkout HEAD~1`.

```
$ git log
# commit 4e66933d8ef597399a5e82fe8a76d310cf47aa0d (HEAD -> feature)
# Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
# Date: Sun Feb 27 14:59:55 2022 +0100
#
#   edit newfile.txt
#
# commit 3f74be1b1a4ac16d42876397666a3839d93d8619 (master)
# Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
# Date: Sun Feb 27 14:31:54 2022 +0100
```

```
#
#      create newfile2.txt

# commit 08c823b947d2ff9bee678abbb458a7e147eccc0c
# Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
# Date:   Sun Feb 27 14:19:50 2022 +0100
#
#      create newfile.txt
$ git checkout 3f74be1b1a4ac16d42876397666a3839d93d8619
# Note: switching to '3f74be1b1a4ac16d42876397666a3839d93d8619'.

# You are in 'detached HEAD' state. You can look around, make experimental
# changes and commit them, and you can discard any commits you make in this
# state without impacting any branches by switching back to a branch.
#
# HEAD is now at 3f74be1 create newfile2.txt
$ git log
# commit 3f74be1b1a4ac16d42876397666a3839d93d8619 (HEAD, master)
# Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
# Date:   Sun Feb 27 14:31:54 2022 +0100
#
#      create newfile2.txt

# commit 08c823b947d2ff9bee678abbb458a7e147eccc0c
# Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
# Date:   Sun Feb 27 14:19:50 2022 +0100
#
#      create newfile.txt
```

## > Объединение изменений

Для объединения изменений из текущей ветки и другой ветки используется команда `git merge <имя ветки>`. В процессе объединения веток будет создан коммит с двумя родителями (англ. *merge commit*), в котором будут записаны все изменения из обоих веток.

```
$ git checkout master
# Switched to branch 'master'
$ git merge feature
# Updating 3f74be1..4e66933
# Fast-forward
#  newfile.txt | 1 +
#  1 file changed, 1 insertion(+)
```

## > Стратегии ветвления



## > Подход Git Flow

- В ветке **master** всегда находится стабильный и работающий код и только он.
- В ветке **dev** находится слияние всех наработок, которые войдут в следующую стабильную версию. В этой ветке тестируется новый функционал.
- В **тематических ветках** находятся решения конкретных задач, например, реализация определенной функции.

Процесс работы строится следующим образом:

1. От ветки **dev** создается тематическая ветка, где ведется работа над одной проблемой. В конце работы тематическая ветка сливается с веткой **dev**.
2. После проверки кода ветка **dev** сливается с веткой **master**.
3. В случае проблемы на ветке **master** коммиты заносятся в ветку **master**, минуя ветку **dev**; это называется **hotfix**.

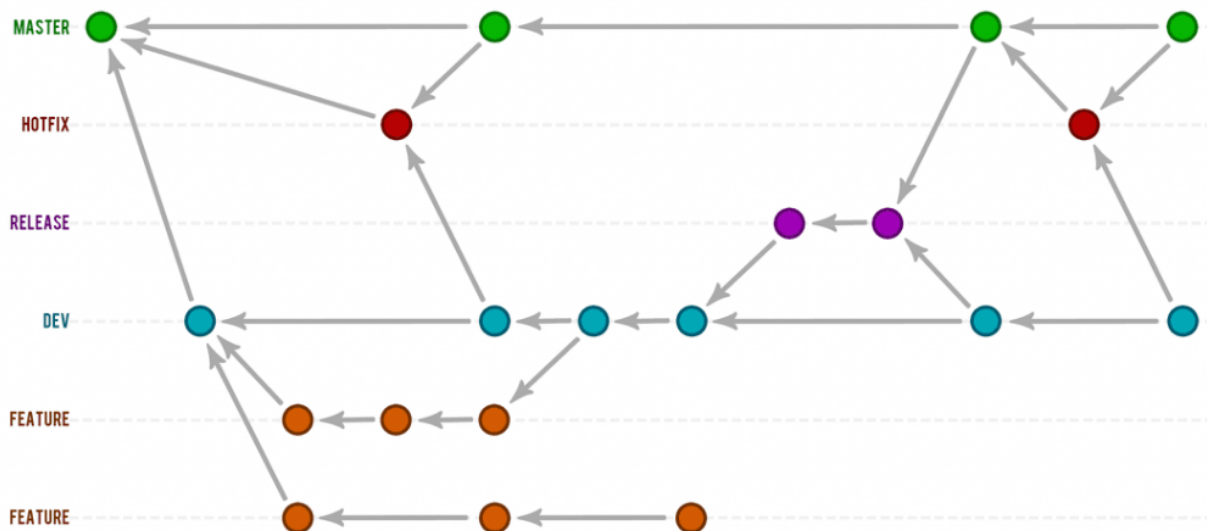


Схема стратегии ветвления Git Flow.

## > Плюсы Git Flow

- ## > Минусы Git Flow

- Из-за большого количества минусов данный подход сейчас не очень популярен.

## > Подход Trunk based development

- 
- The diagram illustrates the Git branching strategy for production-ready code. It shows a 'Master' branch and 'Release Branches'. The 'Master' branch contains production-ready code and is used for creating new production-ready versions. The 'Release Branches' are used for developing new production-ready versions. The diagram shows the flow from Master to Release Branches and back to Master.

Схема стратегии ветвления *Trunk based development*.

## > Плюсы Trunk based development

- Из-за того, что в быстроживущих ветках ведется работа над небольшим функционалом, работа ведется быстро - ветка быстро создается, меняется, чинится.
- В **Trunk base development** есть флаги, которые позволяют включать и выключать фичи, используемые в коде. Поэтому программа всегда готова к релизу, даже если там есть недописанные фичи. [Подробнее](#)
- Частые интеграции, постепенное изменение кода - более плавная разработка.
- Постоянное ревью кода.

## > Минусы Trunk based development

- Так как много людей работает над маленькими частями кода и каждый может слить свою ветку в master, может возникнуть конфликт, особенно если в команде работают junior программисты.

Сравнение двух подходов

## > Конфликт слияния веток

В случае, если изменения в сливаемых ветках противоречат друг другу, возникает конфликт.

```
$ git merge feature
# Auto-merging newfile2.txt
# CONFLICT (content): Merge conflict in newfile2.txt
# Automatic merge failed; fix conflicts and then commit the result.
```

Есть два варианта решения проблемы: **отменить слияние** и **разрешить конфликт**. Для отмены слияния используется команда `git merge --abort`.

```
$ git merge --abort
```

Для разрешения конфликта нужно привести файл, из-за которого конфликт произошел, к желаемому варианту и удалить из него служебную информацию (знаки ->|), записанную в процессе конфликта. Далее нужно добавить файл в staging area и закоммитить изменения.

```
$ nano newfile2.txt
$ git add newfile2.txt
$ git commit
# [master ac774a5] Merge branch 'feature'
```

```
[(base) Bobs-MacBook-Pro:git_dir alenakizenko$ git log --graph
*   commit ac774a5e369ec512c37395465b15d587e202833e (HEAD -> master)
| \ Merge: d100bd0 004bab5
| | Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
| | Date:   Sun Feb 27 16:38:26 2022 +0100
| |
| |     Merge branch 'feature'
| |
| *   commit 004bab50d66ecdbba39b1f03b99f3676edb2e8ba (feature)
| | Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
| | Date:   Sun Feb 27 16:33:49 2022 +0100
| |
| |     edit newfile2.txt
| |
| *   commit d100bd0ec426d6c1b79f2063549e15c00ea05bc7
| / Author: Alena Kizenko <alenakizenko@Bobs-MacBook-Pro.local>
|   Date:   Sun Feb 27 16:32:26 2022 +0100
|   |
|   |     edit newfile2.txt
```

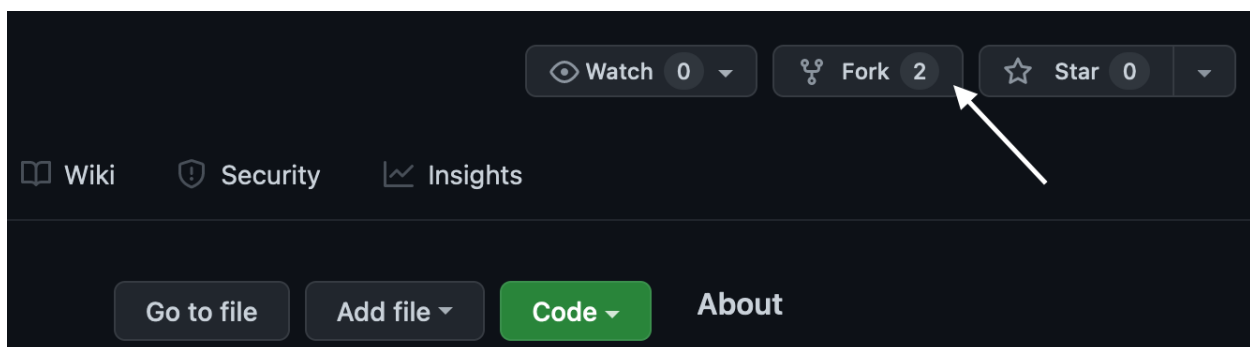
*Разрешение конфликта. Коммит с двумя родительскими коммитами.*

## > Работа с удаленными провайдерами

В git есть возможность работать с удаленными интернет-серверами, например GitHub, GitLab и Bitbucket. Для этого используется абстракция **Remote**.

## > Принципы работы с удаленными репозиториями

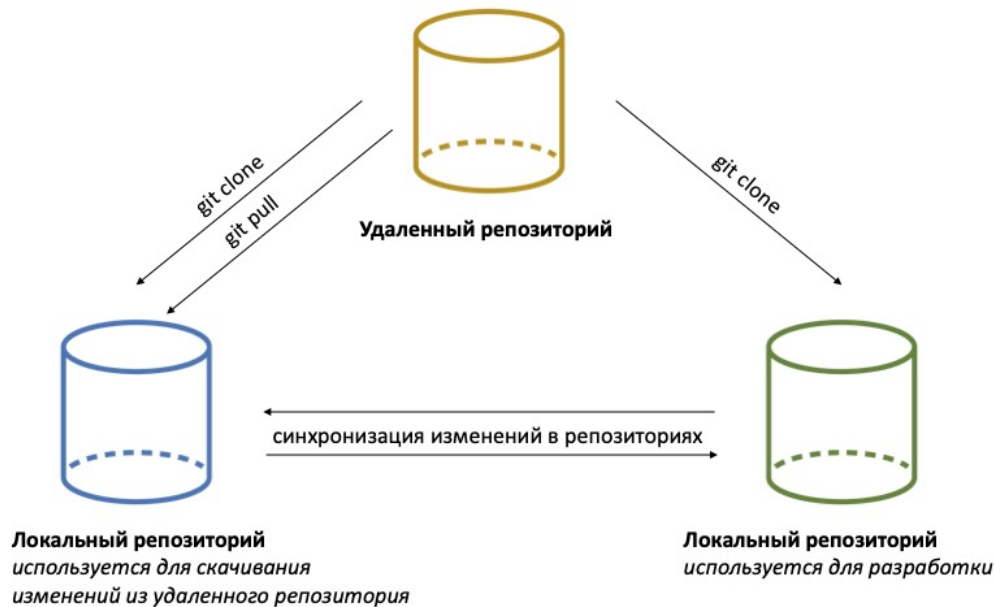
1. Клонирование репозитория. Для скачивания репозитория из интернета используется команда `git clone <ссылка на репозиторий>`. Сервер, с которого производилось клонирование, называется origin.
2. Добавление изменений из удаленного репозитория в локальный репозиторий. Чтобы добавить коммиты из репозитория на сервере в локальный репозиторий нужно выполнить команду `git pull`.
3. Добавление изменений из локального репозитория в удаленный репозиторий. Напротив, чтобы добавить изменения из локального в удаленный репозиторий, используется команда `git push`; эта команда может быть использована только владельцем удаленного репозитория, так как потребуется ввести логин и пароль от репозитория.
4. Копирование удаленного репозитория. Для того, чтобы скопировать чужой удаленный репозиторий и работать над ним в своем репозитории, используется функция **fork**. После окончания работы можно предложить автору репозитория свои изменения.



*Пример того, как выглядит **fork** на платформе GitHub.*

В git существует возможность использования нескольких **remotes**.

В большинстве случаев это используется при работе с чужим удаленным репозиторием. Дело в том, что **fork** не обеспечивает загрузку изменений из удаленного репозитория в локальный. Поэтому обычно remote создается два раза — в одном ведется разработка, в другом обновляется код.



Также, используя несколько `remotes`, можно хранить код в нескольких облачных хранилищах, например, на GitHub и на Bitbucket.

Для того, чтобы просмотреть список настроенных удалённых репозиториев, нужно запустить команду `git remote -v`. Если вы клонировали репозиторий, то увидите как минимум `origin` — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование.

## > Этапы загрузки репозитория на удаленный сервер

1. Создание репозитория на github/gitlab
2. Добавление URL репозитория в remote `git remote add <origin> <URL>`
3. Отправка ветки в репозиторий `git push --set-upstream origin <имя ветки>`
4. Отправка всех веток в репозиторий `git push --all`

5. Отправка тэга в репозиторий `git push --tag`

## > Доступ на GitHub по SSH

Доступ на удаленный сервер может осуществляться с использованием логина и пароля, а также с использованием SSH-ключа. Доступ по ключу SSH является более безопасным, поэтому некоторые удаленные провайдеры, например GitHub, совсем отказались от использования логина и пароля для входа на сервер.

**SSH-ключ** - это уникальный набор знаков, который хранится на локальном компьютере и позволяет получать доступ на сервер по безопасному протоколу SSH без использования логина и пароля.

Более подробно про протокол можно почитать [здесь](#).

Как настроить доступ по SSH на GitHub можно прочитать [здесь](#).

## > Советы по работе с git

### 1. Не стесняйтесь создавать ветки.

Философия git - создавать как можно больше веток и работать с ними. Если вы начали работать над новым функционалом - создайте отдельную ветку, это добавит порядка в ваш репозиторий.

### 2. Старайтесь делать коммиты только исправного кода.

Как только коммит сделан, он остается в памяти git навсегда. Если другой разработчик в будущем захочет вернуться к какому-то коммиту, то будет лучше, если он окажется рабочим.

### 3. Делайте коммиты как можно более точечными.

Старайтесь делать коммиты, соответствующую одному логическому изменению. Например, отдельный коммит на редактирование документации, отдельный - на изменение кода.

### 4. Следите внимательно за тем, что коммитите.

Следите за тем, что находится в staging area, перед тем как коммитить.

### 5. Начинайте рабочий день с `git pull`.

Когда вы открываете репозиторий, с которым давно не работали, прежде всего сделайте `git pull`. Это синхронизирует локальный код с тем, который находится в удаленном репозитории.

## > Основные команды терминала

`mkdir` - создать директорию (папку).

`ls` - увидеть все файлы и папки в директории; флаг `-a` позволяет увидеть скрытые папки и файлы.

`cd` - перейти в другую директорию.

`rm` - удалить файл; флаг `-r` позволяет удалить директорию.

## > Работа с Vim

В текстовом редакторе Vim применяются два основных режима ввода — **командный** и **текстовый**. Режимы переключаются вручную. После запуска редактор автоматически открывается в **командном режиме**. Когда пользователь переключается с одного режима на другой, клавиши клавиатуры начинают работать немного по-другому. В **командном режиме** все введенные символы редактор будет воспринимать как команды.



~~~~~

Чтобы перейти в текстовый режим из командного, нужно использовать команду **i**. Любой введённый символ редактор не будет считать командой, а вставит его в текст. Чтобы обратно перейти в командный режим, нужно нажать клавишу **Esc**.

-- INSERT --

## > Работа с Nano

В данном редакторе есть только один режим - **текстовый**. Редактор разбит на 4 основные части: **верхняя строка** содержит версию программы, текущее имя файла, который редактируется, и были ли внесены изменения в текущий файл.

**Вторая часть** - это главное окно редактирования, в котором отображен редактируемый файл. Строка состояния - **3 строка снизу** - показывает разные важные сообщения. **Две строки внизу** показывают наиболее часто используемые комбинации клавиш.

```
GNU nano 2.0.6           File: newfile.txt           Modified

hello, world!
learning is great
startML is a course about machine learning

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

Больше информации про [nano](#).

## > Основные команды git

`git init` - создать в текущем каталоге новый подкаталог с именем `.git`, содержащий все необходимые файлы репозитория - структуру Git репозитория.

`git add` - начать отслеживать (добавление под версионный контроль) новый файл.

`git commit` - осуществить коммит изменений.

`git status` - определить состояния файлов.

`git diff` - определить состояние файлов и того, что конкретно поменялось, а не только какие файлы были изменены (развернутый вариант `git status`); флаг `--staged` позволяет увидеть проиндексированные изменения.

`git log` - вывести историю коммитов; флаг `--graph` позволяет вывести историю коммитов в виде дерева.

`git checkout` - переключиться на другой коммит.

`git branch` - создать новую ветку.

`git merge` - влить одну ветку в другую.

`git clone` - клонировать удаленный репозиторий.

`git pull` - получить изменения из удалённой ветки и слить их со своей текущей.

`git push` - отправить ветку на сервер.