Термины.

* Репозиторий — папка с файлами и настройками Git. Настройки расположены в скрытой папке .git.
* Форк (fork) — копия репозитория. GitHub создаст копию фокрнутого репозитория у вас на аккаунте. После этого в копию репозитория можно будет вносить изменения. Внесенные изменения можно предложить добавить в основной репозиторий через пул реквест.
* Клонить (clone) — скачивать репозиторий с удалённого сервера на локальный компьютер.
* Ветка / бранч (branch) — это параллельная версия репозитория. Ветки можно объединять между собой, например соединять параллельную версию с главной версией репозитория.
* Мастер (master) — главная или основная ветка репозитория.
* Коммит (commit) — фиксация изменений или запись изменений в репозиторий. Коммит происходит на локальном компьютере.
* Пул (pull) — получение последних изменений с удалённого сервера репозитория.
* Пуш (push) — отправка всех неотправленных коммитов на удалённый сервер репозитория.
* Пул реквест (pull request) — запрос на слияние форка репозитория с основным репозиторием. Пул реквест может быть принят или отклонён владельцем репозитория.
* Мёрдж (merge) — слияние изменений из какой-либо ветки репозитория с любой веткой этого же репозитория. Чаще всего —  слияние изменений из ветки репозитория с основной веткой репозитория.
* Код ревью (code review) — процесс проверки кода на соответствие определённым требованиям, задачам и внешнему виду.
* Обновиться из апстрима — обновить свою локальную версию форка до последней версии основного репозитория, от которого сделан форк.
* Обновиться из ориджина — обновить свою локальную версию репозитория до последней удалённой версии этого репозитория.

Работа с Git.

A. Настройки конфига.

Перед работой с Git нужно настроить своё имя и почту.

1. **Настройка имени**.

git config --global **user.name** **"YourName"**

* git — все команды для Git начинаются с этого слова.
* config — конфигурация.
* --global — означает, что настройка будет применена для всех проектов.
* user.name — настройка, которую нужно установить.
* YourName — имя, которое нужно установить.

2. **Настройка электронной почты**.

git config --global **user.email** **"name@mail.ru"**

3. **Проверить, что настройки сохранились** (имя и электронная почта).

git config --global **--list** (--global, если настройки имени задавались для всех проектов и нужно вывести именно их)

Среди прочих настроек (если они имеются), должны быть такие:

user.name=YourName

user.email=name@mail.com

Дополнительно:

4. **Открыть файл с настройками**. Настройки хранятся в файле .gitconfig

**cat** ~/.gitconfig (такого файла может не быть, если изначально не прописать настройки)

Если настройки успешно сохранились, там будет информация такого плана:

[user]

        name = YourName

        email = name@mail.com

\*в том числе и другие настройки, если они есть\*

5. **Узнать, как работает команда в консоли**.

git **help** **название\_команды** (показать справку по выбранной команде)

git help (покажутся самые популярные команды)

git help config (покажется справка для команды config)

B. Работа с локальными папками.

1. **Перемещение по папкам**.

**pwd** — узнать полный путь к папке, в которой сейчас находимся.

**cd** название\_папки — для перехода в выбранную папку.

cd **..** — переместиться на одну папку назад.

сd **/** — переместиться на одну папку вперед.

cd **-** — вернуться к предыдущей папке.

cd **~** — перейти в папку, которая назначена домашней.

Пример:

cd /d (перейти на диск D)

cd code/'RS School' (перейти в папку "code", а затем в папку "RS School" — в коде она указана в кавычках, так как содержит пробелы)

2. **Обзор папки**.

**ls** — какие файлы есть в папке.

ls **-1** — какие файлы есть в папке, вывод в столбик.

ls -1 **-a** — какие файлы есть в папке, вывод в столбик, в том числе скрытых файлов.

Быстрые клавиши:

cd ~/D + tab — покажет, какие папки начинаются на букву "D".

cd ~/Des + tab — терминал сам допишет вместо "Des" — "Desktop".

3. **Открыть папку**.

**start .** — открыть папку, в которой сейчас находимся.

**code .** — открыть папку в редакторе кода.

4. **Создание файлов и папок через консоль**.

**touch** index.html — создать файл index.html.

**mkdir** project — создать папку с именем project.

С. Привязать папку с проектом к Git (превратить проект в репозиторий).

После этого в проекте можно будет отслеживать изменения.

1. **Подготовить к работе репозиторий** (если недавно его создали / он ещё не привязан к Git).

git **init**

После этого в папке с проектом появится скрытая папка ".git", в которой содержится вся история изменений.

2. **Посмотреть состояние репозитория** — проверяем, были ли добавлены новые изменения, которые ещё не внесены в историю изменений.

git **status** (эта команда может использоваться довольно часто для проверки состояния репозитория)

Обратить внимание:

* Команда сработает, если напротив директории, в которой находимся, есть надпись (master).
* Если такой надписи нет — или нахожусь не в той папке или перейти к шагу 1.

Возможные варианты:

* Если выводятся файлы, выделенные красным — значит есть новые изменения, которые не добавлены. Иначе говоря — есть не отслеживаемые Git файлы.
* Если файлы выводятся зеленым — недавно измененные файлы добавлены. Иначе говоря — все файлы отслеживаются Git.
* Если нет подсвеченных файлов и надпись такого плана: "nothing to commit, working tree clean" — значит новых не внесенных изменений нет. Все файлы отслеживаются Git.

3. **Проиндексировать новые или измененные файлы** для последующего сохранения состояния.

git **add .** (проиндексировать все файлы в папке).

git **add** **index.html** (проиндексировать только файл index.html)

git **add** **css/button.css** **css/main.css** (проиндексировать сразу 2 файла, которые находятся в папке "css": button.css и main.css, указанные через пробел)

После: git status — можно проверить, проиндексировались ли все файлы.

4. **Зафиксировать изменения** — закоммитить. Сделать новые изменения отслеживаемыми.

Благодаря этой процедуре, в случае надобности — можно откатиться к предыдущему закоммиченному изменению. Иначе говоря: коммит — это как save в игре.

git **commit -m** **"Описание, что изменилось"**

После: git status — проверить, закоммитились ли файлы. Если да, будет сообщения такого плана: "nothing to commit, working tree clear".

Коммитить нужно всегда, когда состояние части работы обрело законченный вид: добавили новый блок, исправили ошибку, сделали изменения, потеря которых будет критична и т.п.

Дополнительно:

5. **Посмотреть, что изменилось**.

Посмотреть непроиндексированные изменения:

git **diff** (гит покажет, что именно изменилось в файле: красным — удаленное, зеленым — добавленное)

Посмотреть проиндексированные изменения:

git diff **--staged**

6. **Посмотреть историю коммитов**.

git **log** (покажет всю историю коммитов этого репозитория, если она есть)

git log **-2** (покажет 2 последних коммита этого репозитория)

В консоли появится информация подобного рода:

commit **c73af11e0830fe25cv4648c2356742f253cc000d** (HEAD -> master, origin/master)

Author: YourName <name@mail.com>

Date:   Sun Feb 24 22:01:09 2019 +0300

    second commit

commit **e5cdl7321di6love8the2rs1school687dde6282**

Author: YourName <name@mail.com>

Date:   Sun Feb 24 21:50:20 2019 +0300

    first commit

c73af11e0830fe25cv4648c2356742f253cc000d — хэш второго коммита. Зная хэш коммита, можно заглянуть в лог коммита или откатиться к этому коммиту.

Также можно вывести историю всех коммитов в одну строку.

git log **--oneline**

Появится информация такого плана:

5c71719 (HEAD -> master) Уменьшили шрифт на кнопках

b69e1e8 Уменьшили шрифт телефона

2c8bd55 Заменили span на botton в кнопках для лучшей доступности

6bf8b41 Сделали рефакторинг

0c5e111 Починили ошибку фильтрации

5949091 Изменили основной шрифт

5d33d23 Установили цвет кнопок и фона

4f485ba Начали вести историю

7. **Узнать, какие изменения были внесены в определенном коммите**.

git **show** e5mnd6f07fb3n72f9ccc3b24839687d61cde6282

D. Если нужно что-то восстановить.

1. **Откатить файл до** **последнего** **коммита**. Вернуть файл в то состояние, в котором он был в последнем коммите. Например, если файл удален.

git **checkout** index.js (теперь, если прописать git status, то будет видно, что файл не изменен)

2. **Откатить файл до** **определенного** **коммита**.

git checkout **c73af1de0877fe25eb4648c2356742f253cc000d** index.js

c73af1de0877fe25eb4648c2356742f253cc000d — хэш того коммита, до которого нужно восстановиться.

3. **Убрать файл из индекса** (перед коммитом). Когда изменения подготовлены и уже добавлены в индекс (git add), но выяснилось, что какой-то файл индексировать не нужно.

git **reset HEAD** index.js (теперь, после проверки изменений git status, файл index.js не будет включен в индекс)

4. **Отменить последний коммит**.

git **revert** **HEAD** (HEAD указывает на последний коммит, поэтому выполнили команду через него)

Но команда revert оставляет историю, показывающую, что был и оригинальный и отмененный коммит. А иногда ненужный (или добавленный с ошибке) коммит нужно удалить из истории.

5. **Исправить последний коммит**. Например, текст коммита был добавлен с ошибкой.

git commit **--amend** -m "Правильное сообщение коммита" (после этого последний коммит с неправильным текстом исчезнет из логов и хэш последнего коммита изменится — то есть не старый коммит поменялся, а создался новый коммит вместо старого)

6. **Убрать файл из последнего коммита и удалить этот файл**.

git **rm** index.js

Осталось внести изменения в коммит, без внесения изменений в комментарий к нему:

Если прописать git status, будет следующее сообщение:

deleted: index.js

Поэтому можно изменить предыдущий коммит, убрав из него изменения:

git commit --amend **--no-edit**

7. **Убрать файл из последнего коммита, но не удалять этот файл**.

git rm **--сached** index.js

Снова можно изменить прошлый коммит от внесения в него изменений:

Если прописать git status, будет такого плана информация:

deleted: index.js

Untrasked file: index.js (файл останется, но от теперь будет неотслеживаемым)

git commit --amend --no-edit

E. Появление веток, слияние.

Ветки — это когда в некоторой точке история коммитов разделяется. Они помогают разным людям работать параллельно над одним проектом — каждый в своей ветке. А затем эти ветки соединяются воедино — в один проект.

**Вывести содержимое коммита в удобном для чтения виде**.

git log --oneline (вывести коммит в 1 строку, а не в 5 как обычно)

git **cat-file -p** b69e1e8

-p — выводит данные в удобном для чтения виде.

b69e1e8 — хэш коммита.

Выведется информация такого плана:

tree 9f390a90d40889a27d85910cb8b4530a02fd441e (состояние файлов на момент коммита)

parent 4f585bafe99v8bd57e2c82cf6a8b07569df8b810 (хэш родительского коммита)

author Eugene Nedelski <name@mail.com> 1518978605 +0300 (два числа - это время ... )

committer Eugene Nedelski <name@mail.com> 1518978828 +0300 (... когда был создан коммит )

* Обычно author и committer совпадают. Но это могут быть разные люди, если один разработчик изменил коммит другого.
* Коммит хранит не только состояние, но и: кто, когда, почему его сохранил. И ещё знает предыдущий коммит — коммит родителя.
* У самого первого коммита не будет родительского.

Визуальные примеры.

1. **Расположение коммитов**.

Если есть 3 коммита — мы по умолчанию находимся в последнем.



2. **Новый коммит**.

Когда мы коммитим — фиксируем состояние, то создается новый коммит и мы перемещаемся в него.

git commit -m "Описание, что изменилось"



3. **Вернуться на любой прошлый коммит**.

Мы в любое время можем переместиться в состояние коммитов, которые остались позади.

git checkout 6bf8b41



Если теперь запросить историю коммитов (git cat-file -p 8d54f00), то будет видна история только до 2-го коммита. Но это не значит, что все последующие коммиты исчезли.

**Посмотреть все коммиты**.

git log --oneline **--all**

**Посмотреть все коммиты в виде схемы**.

git log --oneline --all **--graph**

4. **Возвратиться к последнему коммиту**.

git checkout хэш\_последнего\_коммита



Git хранит кодовое слово, которое является указателем на последний коммит. Поэтому последнему коммиту всегда легко вернуться. По умолчанию это кодовое слово — master. А для сервера кодовое слово удаленного репозитория по умолчанию — origin.

5. **Создать новую ветку (создать указатель на коммит)**. На любой коммит можно создать указатель.

Создать ещё один указатель на последний коммит:

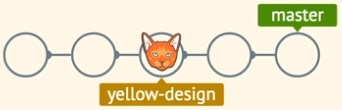
git **checkout -b** **yellow-design**

(yellow-design — имя нового указателя)

Создать указатель на один из прошлых коммитов:

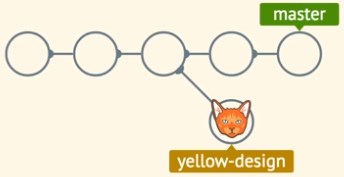
git checkout -b yellow-design **хэш\_коммита\_которому\_нужно\_создать\_указатель**

Допустим, мы указали хэш третьего коммита с конца, тогда получится следующее:



6. **Создание нового коммита из другого указателя**.

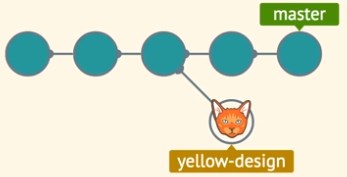
При создании нового коммита с нового указателя, происходит ветвление коммитов и появляются ветки.



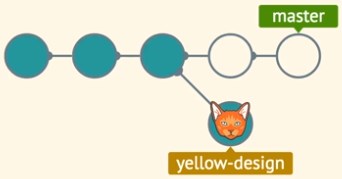
7. **Понятие веток**.

Ветка — вся история коммитов, приводящая в текущую точку.

Одна ветка master:

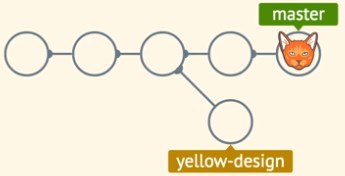


Вторая ветка yellow-design:



8. **Перемещение на другую ветку**.

git **checkout** **master** (перейти к ветке master)



**Как понять, где мы находимся?**

В логах. При запрашивании логов (git log --oneline), можно посмотреть по слову HEAD.

Слово HEAD в логах показывает на наше текущее положение. Например:

(HEAD -> yellow-design) (показывает, что мы находимся в ветке yellow-design)

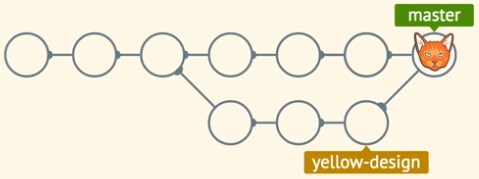
В текущем состоянии. При запрашивании текущего состояние (git status) будет сообщение такого плана:

On branch yellow-design (находимся в ветке yeallow-design)

...

9. **Слияние веток (мёрдж)**.

git **merge** yellow-design -m "Объединили желтый дизайн и мастер"



У нового коммита имя указателя будет тем, от которого мы шагнули при слиянии веток. То есть мы шагнули от ветки master, поэтому и имя у нового указателя такое же.

Обычно основной веткой считается ветка master. От этой ветки могут отводиться другие ветвления. Которые в конечном счете могут снова объединиться в одну примкнув к основной ветке master.

F. Конфликты веток.

Конфликт — это когда один и тот же файл, в одном и том же месте, но в разных ветках, менялся по-разному. Например, в ветке master первая строчка была изменена, а в ветке yellow-design — удалена. При слиянии таких веток возникнет конфликт.

Git сам не знает, какие изменения — правильные. Он предложит решить конфликт участвующим.

При слиянии веток будет информация такого плана:

CONFLICT (content): merge conflict in index.js

В этой ситуации есть 2 решения:

1. **Исправить конфликты и закоммитить**. Для этого нужно отредактировать конфликтующие файлы вручную.

При решении вручную, в редакторе можно увидеть оформление кода такого плана:

<html>

  <head>

<<<<<<< HEAD

    <link rel="stylesheet" type="text/css" href="css/style.css" />

=======

    <link type="text/css" rel="stylesheet" href="style.css" />

>>>>>>> master

  </head>

  <body>

    <h1>Hello,World!</h1>

  </body>

</html>

Нужно убрать информацию от Git, а затем решить, какую строчку кода оставить.

После этого закоммитить изменения:

git commit -m "Решил конфликт при слиянии веток"

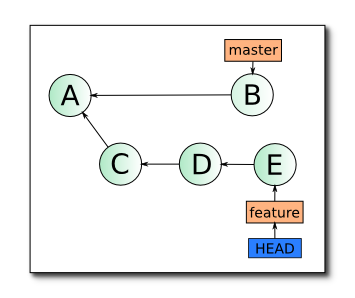
2. **Отменить слияние** (например, если оно было вызвано по ошибке) следующей командой:

git merge **--abort**

G. Перебазирование веток (совмещение / преобразование).

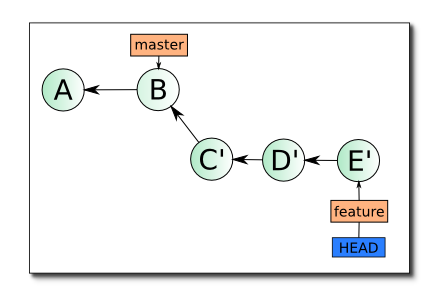
Есть 2 ветки: master и feature.

git checkout feature (переходим на ветку feature)



1. **Совместить ветки**.

git **rebase** master (перенести изменения из master в ветку yellow-design)



Теперь хэш коммитов C', D', E' не равен старому хэшу коммитов C, D, E в ветке feature. Но изменения в коммитах остались ровно такие же.

2. **Слияние vs Преобразование**.

Преобразование делает дерево коммитов более линейным и красивым. Но оно не всегда уместно.

Когда НЕ использовать преобразование (git rebase):

* Если ветка является публичной и расшаренной. Переписывание общих веток будет мешать работе других членов команды.
* Когда важна точная история коммитов ветки (так как команда rebase переписывает историю коммитов).

3. **Подтверждение при отправке на удаленный сервер**.

Так как коммиты одной из веток полностью новые (с новым хэшем), может понадобиться дополнительное подтверждение при отправке на уделенный сервер. Следующей командой мы принимаем риски, связанные с перезаписью коммитов:

git push **--force**

С этим флагом нужно работать аккуратно, потому что могут случайно уничтожиться какие-то важные файлы.

[Подробнее про перебазирование веток](https://habr.com/ru/post/161009/).

H. Работа с GitHub (или другим сервисом, работающим с Git).

Чтобы добавить файлы на удаленный сервер, нужно проделать 3 операции: сперва — проиндексировать, затем — закоммитить, а только потом — отправлять на сервер.

Удаленная ветка — не от слова "удалена", а от "удаленный сервер".

1. **Добавить удаленный репозиторий**.

git **remote** **add** origin \*ссылка\*

origin — имя основного удаленного репозитория. Это не какое-то ключевое слово, так принято его называть.

Проверяем, что репозиторий добавился:

git remote **-v** (-v — значит подробный)

Выведет:

origin \*ссылка\* (fetch) (забирать из него изменения)

origin \*ссылка\* (push) (отправлять туда изменения)

Также можно проверить или получить ссылку на добавленный репозиторий:

git remote **get-url** origin

2. **Отправить изменения в удаленный репозиторий**.

git **push -u** **origin** master

-u — в случае, если находимся, например в не основной ветке yellow-design, но хотим установить связь с основной веткой master.

Если же находимся в основной ветке, можно приписывать команды без -u:

git push origin master

Проиндексированные файлы, новые коммиты, новые ветки будут доступны только на компьютере, пока мы не отправим (запушим) изменения на удаленный сервер (например, на GitHub). Только после этого их можно будет увидеть в браузере, на сайте в аккаунте.

3. **SSH-ключ**.

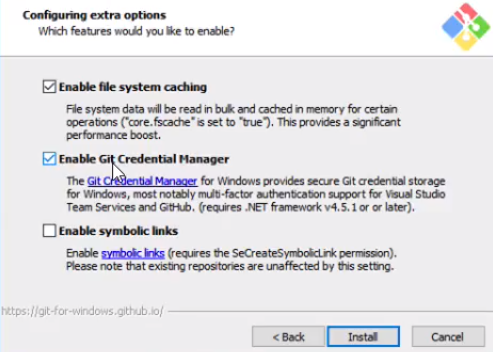
Если нет авторизации в консоли, при попытке запушить изменения в удаленный репозиторий (git push origin master), Git может выдать подобную ошибку:

Permission denied (publickey).

fatal: Could not read from remote repository.

Please make sure you have the correct access rights and the repository exists.

Но так обычно происходит, если при установке GitBash не был включен пункт "Enable Git Credential Meneger". Тогда некоторые средства авторизации, в том числе логин и пароль при каждом запросе понадобится вводить заново.



[Как установить SSH-ключ](https://www.youtube.com/watch?v=KqzVaUTCPbQ).

4. **Если новая ветка создана с опечаткой**.

Например, хотели назвать comments, а напечатали cmments.

4a. **Запушить с одной локальной ветки в другую удаленную ветку**.

git push origin **cmments:comments**

cmments — имя локальной ветки, которую создали с ошибкой.

comments — имя ветки на удаленном сервере с уже правильным названием, куда нужно запушить файлы.

Так можно отправить любую локальную ветку на любую удаленную на ветку.

4b. **Удалить неправильную/ненужную ветку: на сервере**. Ведь после того, как мы запушили в правильно написанную ветку, старая ветка никуда не делась.

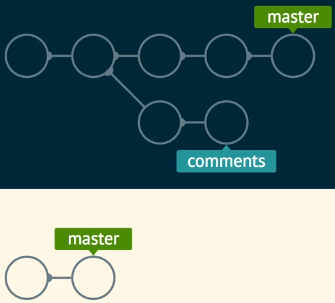
git push origin **:сmments**

4c. **Переименовать неправильную ветку: на компьютере**. Ведь неправильная написанная ветка всё ещё осталась на компьютере.

git **branch -m** **comments**

Команда git branch может: переименовывать, удалять и создавать ветки.

Скачивание и запись на сервер изменений.



На темном фоне — изменения, которые внесены на удаленном сервере.

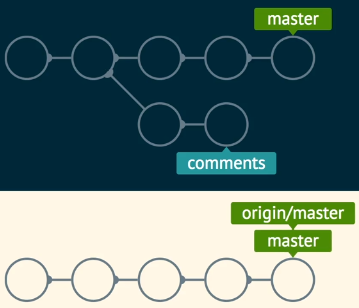
На светлом фоне — изменения, которые есть на локальном компьютере.

5. **Скачать изменения с сервера** (получить изменения из удаленного репозитория).

Например, кто-то ещё работает над теми же файлами. И пока нас не было, он внес какие-то изменения. Чтобы начать работать с уже измененным репозиторием, нам нужно получить изменения. Поэтому команда git clone здесь может быть неудобна — придется заново копировать весь проект, хотя достаточно получить только те фрагменты, которые изменились.

git **pull** origin master (забирает и сливает коммиты и ветки)

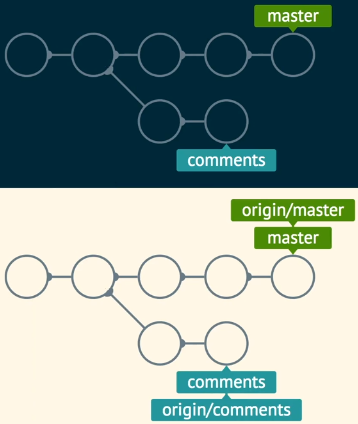
Получится:



Но эта команда скачивает только ту ветку, которую указали — ветку master. В случае, если появились новые ветки и коммиты вне основной ветки — они не будет забраны на компьютер.

6. **Скачать коммиты и ветки с сервера**.

git **fetch** origin (только забирает коммиты и ветки)



Это нужно в том случае, если в репозитории на GitHub были внесены новые ветки и коммиты с другого компьютера. Если не забрав их на свой компьютер, попробовать закоммитить в новую ветку, то ничего не получится.

Команда git pull эквивалентна комбинации git fetch и git merge.

7. **Связать локальную ветку с удаленной**.

git **branch --set-upstrim-to=**origin/comments

8. **Посмотреть, какие локальные ветки связаны с удаленными**.

git branch **-vv**

-v — подробно.

-vv — очень подробно.

I. Упрощение работы с консолью.

**Если редактирование текстовых файлов проходит в консоли GitBash**.

Vim — по умолчанию редактор текстовых файлов в GitBash.

:q — выйти с редактора

:q! — выйти с редактора, если через него что-то менялось

**Создать удобное имя для команды - алиас**.

Алиас — простое кодовое значение команды или нескольких команд.

Некоторые команды в консоли вводить долго и неудобно. Поэтому можно создать их алиас.

git config --global **alias.с config**

с - теперь эта команда в консоли будет работать также, как команда config.

Другой пример:

git config --global alias.**сg** **'config --global'**

cg — теперь набирая эту одну команду будут срабатывать сразу две команды config --global.

Примеры работы.

Работа с тасками по JS.

0. Фокраем нужный репозиторий с тасками.



1. **Копируем репозиторий к себе на компьютер и открываем**.

git clone \*ссылка на репозиторий\* — скопировать репозиторий из GitHub к себе на компьютер (без звездочек).

pwd — посмотреть, где находимся и сориентироваться, куда скопировать/скопирован репозиторий.

cd D/code/ — найти нужную папку с репозиторием.

2. **Открываем/запускаем таск**.

start . — открыть папку, в которой сейчас находимся, в проводнике.

Если указать не папку, а какой-то файл, то он откроется в программе, которая указана как открывающая его по умолчанию.

code . — открыть код в редакторе.

3. **Пишем код, не забываем фиксировать изменения**, чтобы не потерять.

git status — проверить, есть ли проиндексированные файлы.

git add . — проиндексировать файлы.

git status — снова проверить, проиндексировались ли файлы.

git commit -m "Что нового сделано" — закоммитать, создать версию проекта.

4. **Тестим написанный код**.

npm install — устанавливает тестировочную базу, без неё тесты работать не будут.

npm test — тестирует программу.

Если тесты проходят хорошо — таск выполнен, осталось только отправить на удаленный сервер.

5. **Отправляем код с компьютера на GitHub**.

git status — проверить, есть ли проиндексированные файлы.

git add . — проиндексировать файлы.

git status — снова проверить, проиндексировались ли файлы.

git commit -m "финальные изменения" — сделать финальный коммит.

git push origin master — запушить, внести изменение из локальной версии (на компьютере) на GitHub (на удаленный сервер).

**Полезные ресурсы**.

1. Уроки: <https://githowto.com/ru>

2. Руководство: <https://github.com/k88hudson/git-flight-rules/blob/master/README_ru.md>

3. Ещё руководство: <http://firstaidgit.ru/#/>

4. Большое руководство Стэнфорда: <http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/>

5. Книга: <https://git-scm.com/book/ru/v2>

6. Скринкасты: <https://www.youtube.com/playlist?list=PLDyvV36pndZHkDRik6kKF6gSb0N0W995h>