

Содержание:

[СОЗДАНИЕ АККАУНТА](#)

[GITHUB REPOSITORIES](#)

[COMMITTS](#)

[GIT PUSH](#)

[ВЕТКИ](#)

[PULL-REQUEST](#)

[GIT PULL](#)

[GIT FETCH](#)

[GIT STASH](#)

[SSH KEY](#)

[КЛЮЧИ И ПАРОЛИ В КОДЕ.](#)

[МУСОР В КОДЕ И CODE STYLE.](#)

[УДАЛЕНИЕ ВЕТКИ](#)

[УДАЛЕНИЕ РЕПОЗИТОРИЯ:](#)

[README.md](#)

[ОСНОВНЫЕ КОМАНДЫ:](#)

[Полезные ссылки для работы с гитом:](#)

СОЗДАНИЕ АККАУНТА

При создании аккаунта имейте ввиду, что ваш юзернейм должен содержать только ФИО и ничего другого

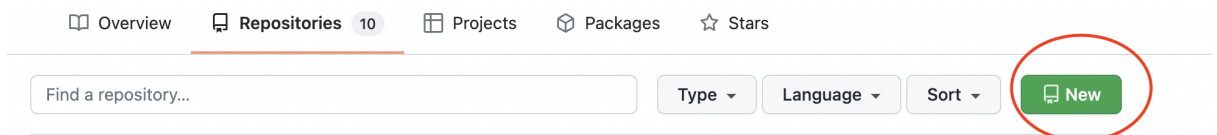
Плохо: frontendMenExceedTeam

Хорошо: IvanovIvanIvanovich

GITHUB REPOSITORIES

Гит репозиторий - это виртуальное хранилище проекта. В нем можно хранить версии кода для доступа по мере необходимости.

Новый репозиторий создается на главной странице аккаунта GitHub во вкладке Repositories при нажатии new.



Старайтесь для front-end и back-end part одного приложения вести разные репозитории, чтобы всем кто просматривает Ваш GitHub было проще ориентироваться. Это больше рекомендация, чем обязательное правило.

Репозиторий именуется понятно и информативно. Если название дано верно - можно будет сразу понять что это за проект и не будет путаницы между кучей однотипных неинформативных названий.

Примеры названия репозитория:

Плохо: “project1” - что такое project1? Это самый первый в жизни проект? Самый первый на этом гитхабе? Какую информацию в себе несет это название? Будет ли интересно вашему будущему работодателю, заказчику заглянуть в этот проект?

нормально: “todo” - такое название несет в себе уже больше информации, уже хорошо, что у Вашего проекта хотя бы есть название) Но можно лучше

хорошо: ToDo-list-front-end - таким образом Ваш заказчик сразу глядя на название репозитория сможет понять: что это за проект? какая его часть тут написана? а так же внизу названия появится ярлык с информацией про язык на котором этот проект написан. Вся остальная важная информация будет указана в README файле.

При **создании** репозитория всегда добавляем файлы для гит игнора, чаще всего по умолчанию ставим игнор для node-модулей.

Просмотр удалённых репозиторий

Для того, чтобы просмотреть список настроенных удалённых репозиторий, используется команда `git remote` из инициализированной папки. Она выведет названия доступных удаленных репозиторий. Если репозиторий клонирован, то будет показано как минимум `origin` — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование.

COMMITTS

Что такое `commit`? закоммитить - зафиксировать свои изменения. Помните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем локальном устройстве. Если есть сомнения - что проиндексировано - воспользуйтесь командой `git status`. Простейший способ зафиксировать изменения — это набрать `git commit`:

Начинайте свой коммит с того, что вы что-то добавили, удалили или изменили. Затем напишите, что получилось в результате. Проверьте себя: читая хороший коммит, посторонний человек может понять, что было сделано и зачем. Например: `fix bugs in edit function`. Никаких знаков препинания и прописных букв, комментарий пишется на английском языке без использования временных форм.

(`git commit --amend` -добавление проиндексированных изменений в последний коммит, т.е. обновление последнего коммита)

Примеры:

полная команда: `git commit -m 'fix bugs in adding tasks'` где `git commit` - команда для фиксации изменений, `-m` - команда которая позволяет добавить комментарий к коммиту

Плохо: `git commit -m "LALALALALA"`

хорошо: `git commit -m "add functionality for edit"`

`git commit -m "add edit functional"`

`git commit -m "fix code style"`

`git commit -m "fix bugs in adding tasks"` эти комментарии отвечают на вопросы: что я сделал? для чего? В каком месте кода?

В сложных проектах длина комментария для одного коммита может достигать 50 символов.

GIT PUSH

Git push — передаёт все новые данные (те, которых ещё нет в удалённом репозитории) из локального репозитория в репозиторий удалённый. Для исполнения этой команды необходимо, чтобы удалённый репозиторий не имел новых коммитов в себя от других клиентов, иначе `push` завершается ошибкой, и придётся делать `pull` и слияние.

Полное написание команды: `git push origin feature/edit-task` - где *origin* указание на то, что мы работаем с удаленным репозиторием, а *feature/edit-task* -это название ветки с которой мы работаем.

Иногда мы можем использовать просто `git push` и программный интерфейс все поймет. Для этого мы должны находиться в ветке с которой работаем локально.

Иногда при выполнении команды `git push` - программный интерфейс выдает нам ошибку.

```
fatal: The current branch feature/router has no upstream branch.
```

За нас эту проблему решает сам интерфейс, если прочитать сообщение ошибок которое он прислал, то ниже можно увидеть, что он нам уже предложил команду для установки ветки на удаленном репозитории и последующей отправки туда последний изменений.

```
To push the current branch and set the remote as upstream, use  
git push --set-upstream origin feature/router
```

ВЕТКИ

По общепринятым правилам ветки feature и hotfix практически всегда создаются от главной ветки, которую показывают заказчику/контрибьютору (чаще всего это main или master).

Под каждую задачу стараемся вести отдельную ветку.

Именование ветки должно начинаться с:

-*feature*/ если это новая ветка с новым функционалом.

-*fix*/ если это ветка с правками.

-*hot-fix*/ если здесь хранятся срочные правки, без которых Ваш проект не может нормально функционировать или не может функционировать вообще.

Затем добавляется основное название ветки, которое будет соответствовать главной задаче этой ветки (например - подключение к базе данных - connect-bd, редактирование - edit-task и т. д.)

Ветки сопровождения или исправления (hotfix) используются для быстрого внесения исправлений в рабочие релизы. Это единственная ветка, которую нужно обязательно создавать напрямую от ветки, которую демонстрируют заказчику (в нашем случае - main/master).

Как только исправление завершено, эту ветку следует слить с той - от которой она была создана.

Команды для работы с ветками:

git branch nameBranch - эта команда только создает новую ветку, но не переключает на неё.

git checkout -b nameBranch эта команда создает новую ветку и затем переключает на неё.

git checkout nameBranch - находит уже существующую ветку и переключает на неё.

Пример создания ветки:

плохо: `git checkout -b first`

`git checkout -b sasa`

`git checkout -b project`

нормально: `git checkout -b add-tasks`

`git checkout -b delete-tasks`

`git checkout -b edit-tasks`

хорошо: `git checkout -b feature/edit-tasks` - при создании ветки с абсолютно новым функционалом.

`git checkout -b fix/edit-tasks` - когда нам нужно что-то исправить в основной ветке `edit-tasks`, но работа кода после этих правок под вопросом

`git checkout -b hot-fix/edit-task` - когда нам нужно чтобы фиксы по задаче `edit-task` были срочно проверены, потому что без них наш проект не может правильно или вообще не может функционировать.

PULL-REQUEST


Pull-request — это запрос (англ. request — «запрос») на интеграцию изменений из одной ветки в другую. Причем в ветке может быть всего один коммит одного разработчика, а может быть несколько коммитов разных авторов. В большинстве случаев *pull-request* используется для интеграции нового функционала или для исправления бага в основной ветке проекта.

Pull-request также содержит короткое описание изменений и причин, по которым эти изменения вносятся. Обычно между автором пул-реквеста и ревьюерами происходит обсуждение этих изменений.

Ревьюеры — это другие разработчики, работающие над этим проектом и способные дать обратную связь по вносимым изменениям. Как правило ревьюерами бывают более опытные коллеги (разработчики-сеньоры), наставники или заказчик. Они сначала проверяют Ваш код и затем уже либо просят внести правки либо дают добро на слияние.

Т. Е. самостоятельно мы ничего не добавляем сразу в главную ветку - мы ждем подтверждения или разрешения от ревьюера.

GitHub при создании и заливке ветки на удаленный репозиторий сам предлагает Вам открыть пулл-реквест на слияние информации с созданной ветки с какой либо другой.

 feature/router had recent pushes less than a minute ago

Compare & pull request

алгоритм создания первого пулл реквеста и отправки ссылки на проверку заказчику/контрибютору:

1. Создать новую ветку от главной и начать работать в ней.
2. После того, как код проверен, убран весь мусор(см. раздел “мусор в коде”), заиндексируйте и зафиксируйте все изменения:
git add .

3. *git commit -m 'comment'*

4. Затем залейте эти изменения на удаленный репозиторий:
git push origin nameBranch

мы не пушим изменения сразу в главную ветку!!!!

5. Переходим на удаленный репозиторий, открываем пулл реквест
6. Находясь в разделе pull-request, проверяем из какой ветки в какую мы запрашиваем у контрибьютора слияние.
7. Проверьте файлы которые отправляем на проверку.
8. Копируем ссылку на pull-request в адресной строке.
9. Отправьте заказчику ссылку на pull-request с кратким описанием того что там находится:

Hello)

How are you?

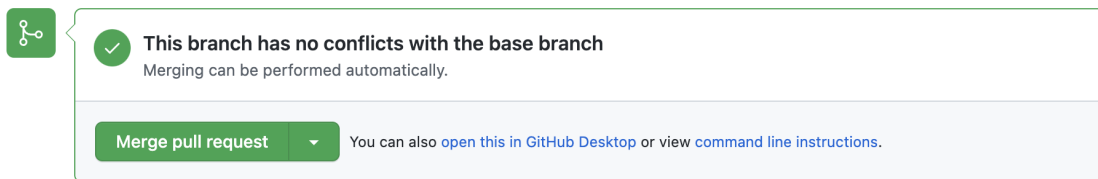
I fixed all comments

Could you check it out when you are free please?

Link:

Have a good day

Эта кнопка нужна только для контрибьютора.



GIT PULL

Git pull извлекает (*fetch*) данные с сервера и автоматически делает слияние (*merge*) их с кодом текущей ветки.

Полная команда:

```
git pull origin branch
```

GIT FETCH

Git fetch — связывается с удаленным репозиторием и получает данные, которые отсутствуют в локальном. При выполнении этой команды слияние не происходит.

GIT STASH

Git stash — команда сохраняющая измененное состояние рабочей директории или отдельного файла в хранилище незавершенных изменений. Это дает возможность в любой момент применить их обратно. Например, если нужно переключиться между ветками без фиксации изменений, можно применить команду *git stash*, рабочая директория останется без изменений, данные будут сохранены в специальном хранилище. Для просмотра скрытых изменений нужно вызвать команду *git stash list*, а для применения — *git stash apply*.

для того, чтобы снова показать скрытые локальные изменения: *git stash pop*.

Эта команда будет полезна при работе с несколькими ветками. После ее применения Вам будут видно только те изменения, которые прямо сейчас есть на вашей удаленной ветке. Таким образом, скрыв локальные изменения, Вы можете без потерь перейти на другую ветку, пофиксировать

комментарии, запустить изменения, вернуться на предыдущую ветку и сделать `git stash pop`. После этого продолжить работу.

SSH KEY

Нужны для установки безопасного соединения. Чтобы подключить гитхаб через ssh нужно сначала проверить наличие этих ключей на локальном хранилище.

Проверка существующего ключа:

<https://docs.github.com/en/authentication/connecting-to-github-with-ssh/checking-for-existing-ssh-keys>

Генерация НОВОГО КЛЮЧА

<https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent#generating-a-new-ssh-key>

Использование ключа:
[https://docs.github.com/en/developers/overview/managing-deploy-keys#de
ploy-keys](https://docs.github.com/en/developers/overview/managing-deploy-keys#deploy-keys)

КЛЮЧИ И ПАРОЛИ В КОДЕ.

Не оставляйте в ваших репозиториях ключи и пароли — это дурной тон. Храните такие данные в переменных окружения или в `.env`-файле, добавив его в `.gitignore`. Так же туда желательно переносить ссылки на базы данных, урлы

МУСОР В КОДЕ И CODE STYLE.

Код не должен содержать: *console.log*, *принты*, *выводы ошибок* и *ненужные комментарии*. Допустимы комментарии на *английском* языке, которые кратко будут пояснять задачу определенного куска кода, чтобы в дальнейшем при просмотре можно было ориентироваться и быстрее восстановить свой ход мыслей.

При выгрузке проекта в GitHub и отправке pull-request заказчику - проверьте свой проект на соблюдение некоторых правил:

1. Код должен быть читабельным
2. Нейминг сразу должен говорить о том, что хранится или происходит в переменных, классах, функциях, модулях или файлах — это всё важно в работе.
3. Одна функция должна выполнять одну задачу. Если функция выполняет несколько задач - дробите ее на несколько.
4. Сложный код - не всегда хороший код. Ищите возможности упростить код.

УДАЛЕНИЕ ВЕТКИ

Локальные ветки — это ветки на вашем компьютере, которые не влияют на ветки удаленного репозитория.

Команда для удаления локальной ветки в Git:

git branch -d local_branch_name - где:

-*git branch* - опция для обнаружения всех веток

-*d* — флаг, опция команды *git branch*, сокращенный вариант записи *--delete*. Как и следует из названия, предназначен для удаления ветки.

-*local_branch_name* имя локальной ветки, которую мы хотим удалить.

Команда для удаления удаленной ветки в Git:

git push origin -d remote_branch_name - где:

git push - команда для выгрузки на удаленный репозиторий

origin - указатель на то что мы работаем с удаленной веткой

-*d* - флаг для удаления

-*remote_name* - имя удаляемой ветки

УДАЛЕНИЕ РЕПОЗИТОРИЯ:

1. Откройте репозиторий
2. Settings
3. пролистываем в самый низ

GitHub Pages

Pages settings now has its own dedicated tab! [Check it out here!](#)

Danger Zone

Change repository visibility

This repository is currently public.

Change visibility

Transfer ownership

Transfer this repository to another user or to an organization where you have the ability to create repositories.

Transfer

Archive this repository

Mark this repository as archived and read-only.

Archive this repository

Delete this repository

Once you delete a repository, there is no going back. Please be certain.

Delete this repository

- 4.
5. репозиторий удален

README.md

README.md первое, что видят посетители Вашего репозитория, контрибуторы, заказчики и работодатели.

README должен отвечать на следующие вопросы:

- О чем (для чего создан) этот проект? (1-2 предложений будет достаточно).
- Как его установить и запустить проект
- Что нужно настроить для запуска проекта
- Какие команды потребуются в работе проекта

Команда запуска

README.md должен содержать команду для запуска приложения.

Даже, если это простые команды по типу: *npm init* и *npm start*. Не стоит забывать о том, что очевидное вам вовсе не обязательно будет понятно тому, кто читает ваш проект.

ОСНОВНЫЕ КОМАНДЫ:

- **git clone** для клонирования репозитория на локальное хранилище. При первом клонировании терминал предложит установить git на ваше устройство

- **git pull** для подгрузки последних данных с удаленного репозитория и объединения с локальной веткой

- **git push** - отправить изменения в удаленный репозиторий

- **git commit** - фиксация изменений

- **git add** -индексация изменений перед фиксацией

- **git branches** - покажет все ветки

- **git diff** -Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите git diff без аргументов:

Если вы хотите посмотреть, что вы проиндексировали и что войдет в следующий коммит, вы можете выполнить git diff --staged. Эта команда сравнивает ваши проиндексированные изменения с последним коммитом:

- **git stash** -скрыть локальные изменения

- **git stash pop** -показать скрытые локальные изменения

- **gitignore**

-**git status.** -проверить какие файлы готовы к фиксации

-**git branch -d local_branch_name** - удаление локальной ветки

-**git push origin -d remote_branch_name** - удаление удаленной ветки

Полезные ссылки для работы с гитом:

Работа с SSH-ключами

<https://www.atlassian.com/ru/git/tutorials/git-ssh>

<https://confluence.atlassian.com/bitbucketserver/creating-ssh-keys-776639788.html>

Учебник по гит:

<http://git-scm.com/book/en/v2>

<https://www.atlassian.com/ru/git/tutorials/learn-git-with-bitbucket-cloud>

Еще раз про работу с гитом, лайфхаки от опытных пользователей гита:

<https://eax.me/git-commands/>

Интерактивное изучение:

https://learngitbranching.js.org/?locale=ru_RU

<https://githowto.com/ru>

Хорошие названия коммитов:

<https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html>

<https://github.com/torvalds/subsurface-for-dirk/blob/master/README.md#contributing>

https://www.git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project#_commit_guidelines

<http://who-t.blogspot.com/2009/12/on-commit-messages.html>

<https://github.com/erlang/otp/wiki/writing-good-commit-messages>

Правила коммитов и работы с ветками, которые приветствуются:

<https://github.com/maxim-oleinik/symfony-dev-rules/blob/master/git.txt>

Полезное дополнение для работы с гитом: [git-flow](#)