Работаем с Git: первые шаги в GitHub



### **Вступление**

Бывает, что начинающие разработчики проблематично осваивают Git и не с первого захода понимают логику работы сервиса. Но стоит создать пару репозиториев или, ещё лучше, погрузиться в реальную историю по установке стартапа на рельсы DevOps, как работа с ветками станет дружелюбной, а PR и MR больше не вызовут путаницы. Ошибки в любом случае появятся, но вы будете к ним готовы!

### **Начало**

Когда вы пишете первую программу, всё кажется таким лаконичным, простым и понятным. Но по мере развития ваша программа обрастает новой функциональностью, становится сложнее и больше. Потом и вовсе появляются первые баги. И было бы здорово помнить или иметь возможность смотреть историю изменений, что добавили или убрали в коде, по какой причине мог появиться баг.

Первое и самое просто решение — «А давайте перед каждым изменением сохранять копию программы (просто копировать папку с кодом)?»

На самом деле это будет работать, но до поры до времени. Проект продолжит расти и станет полезным не только вам, но и вашим друзьям, которые захотят добавить в код что-то своё. В рядах программистов прибывает, и надо как-то договариваться, кто какой кусочек кода трогает, а потом ещё синхронизировать изменения, чтобы все фичи добрались до прода.

Настал звёздный час для систем контроля версий, которые запоминают, какое изменение и в каком файле было сделано, а также могут показать историю этих изменений.

### **Про Git**

Существует несколько систем контроля версий: Git, Subversion, Team Foundation Server, Mercurial. Сегодня познакомимся с Git — самой популярной из них, по скромному признанию более 90% разработчиков.

Git появился 7 апреля 2005 года и был создан для управления разработкой ядра Linux. Кстати, создал его тот самый Линус Торвальдс, а сегодня его развитием и поддержкой занимается Дзюн Хамано.

Git — это распределённая система управления версиями: есть один сервер, через который разработчики обмениваются кодом. Разработчик копирует (клонирует) проект к себе на локальную машину, делает изменения и сохраняет их на удалённый сервер. При необходимости другие разработчики могут скопировать эти изменения к себе.

История и копия проекта хранятся локально и чаще всего не нужна дополнительная информация с других клиентов. Вы можете работать с репозиторием и при отсутствии интернета (например, в самолёте), а когда он появится, просто загрузить изменения в удалённый репозиторий на выделенном сервере.

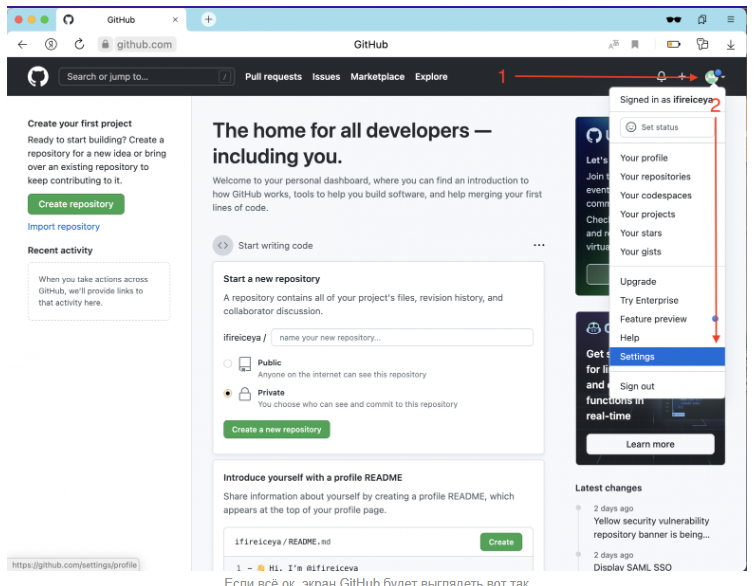
Если у разработчика сломается компьютер, то проект не потеряется, а будет лежать на выделенном сервере. Такой выделенный сервер можно [поднять и настроить самостоятельно](https://about.gitlab.com/) либо использовать готовые решения.

GitHub — крупнейший веб-сервис, который позволяет заниматься совместной разработкой с использованием Git и сохранять изменения на своих серверах. На самом деле [функциональность](https://github.com/about) GitHub намного больше, но сейчас нас интересует только совместная разработка и история изменений. Ещё есть [Gitlab](https://about.gitlab.com/), [Bitbucket](https://bitbucket.org/product/ru/guides/basics/bitbucket-interface) и другие, но мы будем использовать GitHub как самый популярный в настоящее время.

### **Предварительная настройка**

Займёмся предполётной подготовкой.

Для начала [зарегистрируйтесь на GitHub](https://github.com/join?source=header-home): задайте логин, почту и придумайте пароль. После «Создать аккаунт» не забудьте проверить почту и подтвердить её (опрос от Github после подтверждения почты можно пропустить).



В GitHub есть разграничение прав на работу с репозиториями. Можно задавать различные политики: сделать репозиторий публичным и [приватным](https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/managing-repository-settings/setting-repository-visibility), ограничить права кругу пользователей или кому-то одному, например, разрешить просматривать репозиторий, но не изменять в нём данные.

Для того чтобы сервис определил, кто вы и имеете ли право работать с тем или иным репозиторием, нужно представиться — пройти процесс [аутентификации](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F).

GitHub поддерживает безопасность за счёт двух сетевых протоколов, [HTTPS](https://ru.wikipedia.org/wiki/HTTPS) и [SSH](https://docs.github.com/en/authentication/connecting-to-github-with-ssh), и вся работа с сервисом происходит через один из них.

Работать с GitHub будем через терминал по SSH. Для этого один раз сгенерируем специальные ключи и добавим один из них в наш аккаунт на GitHub.

Можно работать и через HTTPS, но нужно будет каждый раз вводить пароль и специальный [token](https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token).

Пара слов про SSH и как он работает. SSH — это сетевой протокол для зашифрованного соединения между клиентом и сервером, через который можно безопасно передавать данные.

При подключении используется пара ключей — открытый (публичный, public) и закрытый (приватный, private). Пользователь создаёт пару ключей при помощи специальной команды и сохраняет закрытый ключ у себя, а открытый кладёт на сервер (в нашем случае на GitHub). А работает это всё благодаря [асимметричному шифрованию.](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%BC_%D0%BA%D0%BB%D1%8E%D1%87%D0%BE%D0%BC)

Алгоритм следующий: отправитель (GitHub) шифрует сообщение публичным ключом и передаёт сообщение клиенту (нам), а мы его расшифровываем при помощи приватного ключа, который предусмотрительно сохранили у себя. То, что зашифровано публичным ключом, расшифровать сможет только приватный ключ.

Давайте создадим пару ключей и добавим открытый ключ на GitHub.

Чтобы создать пару ключей, в терминале нужно ввести команду, задать путь для хранения ключей и указать пароль к ключу (необязательно).

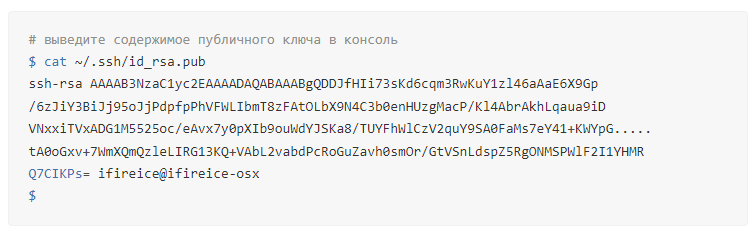
Далее будем опираться на то, что путь для ключей дефолтный и пароль на ключи не установлен.

Пароль для ключей нужен как дополнительная мера безопасности, если вдруг ваш приватный ключ попадёт не в те руки.

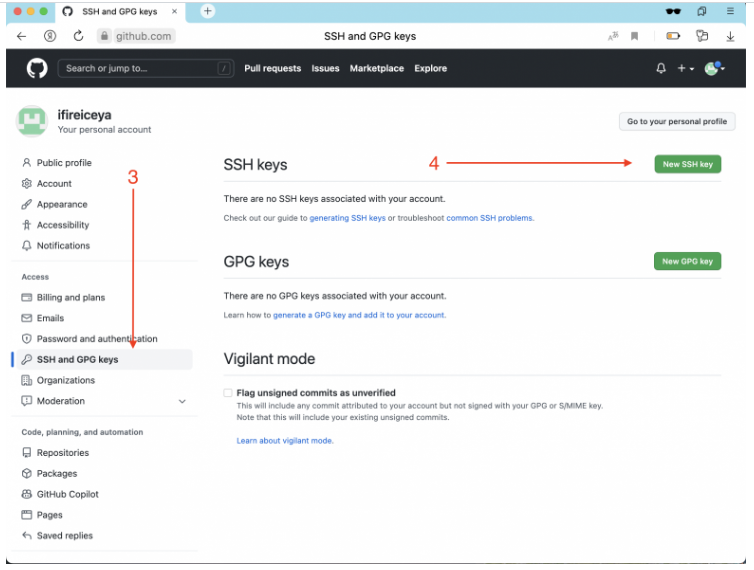


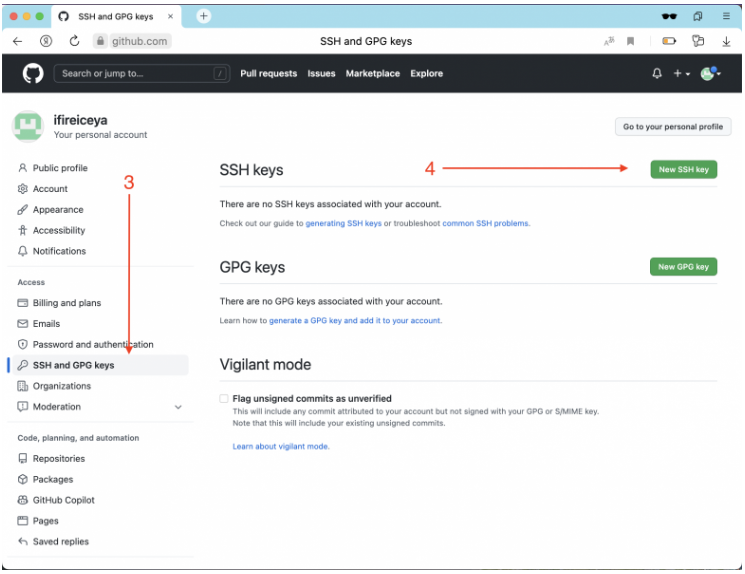
Бинго, ключи сгенерированы: в заданной директории появятся два файла, id\_rsa и id\_rsa.pub.

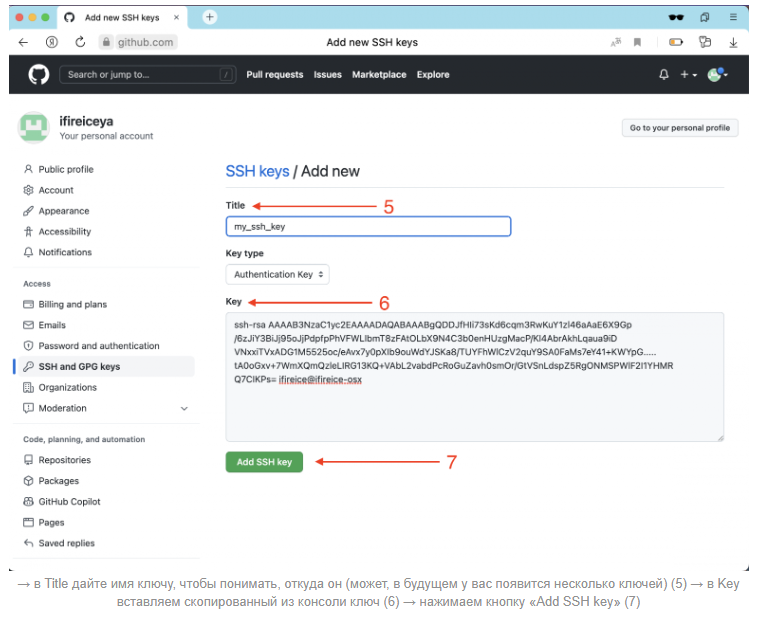
Теперь надо добавить публичный ключ в аккаунт на GitHub:

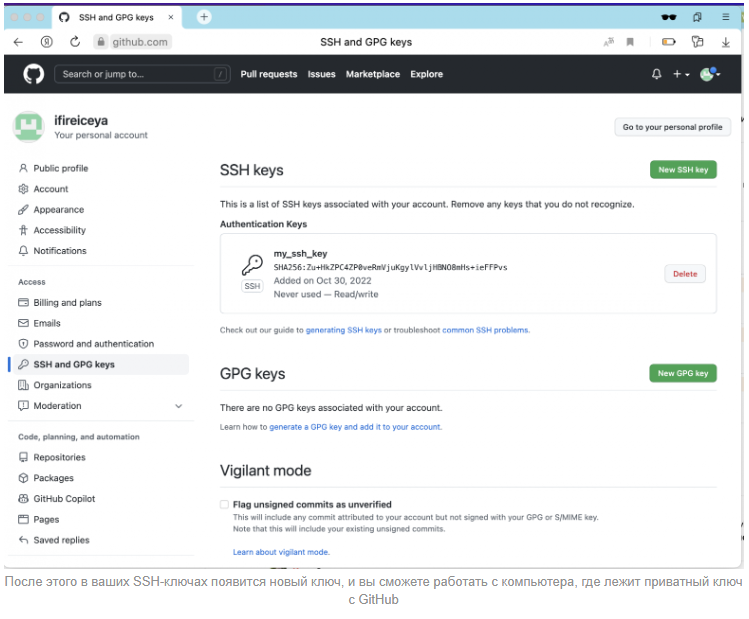


Скопируйте ключ от символов ssh-rsa и до конца файла и вставьте его в ваш аккаунт на GitHub.









Ну что, с настройкой GitHub пока закончили, осталось установить Git на компьютер. Сделать это можно по официальной инструкции (выберите пункт для вашей ОС).

### **Терминология**

Самое время пополнить ваш Git-словарик, прежде чем создадим первый Pull Request.

**Репозиторий (repository)** — директория проекта, который отслеживается Git. В директории хранится проект, история изменений и мета-информация проекта (в скрытой директории .git).

**Индекс** — хранилка, где лежат имена файлов и их изменения, которые должны быть в следующем коммите. По факту индекс — просто файл. В индекс файлы сами не попадают, их нужно явно добавлять при помощи git add.

**Коммит (commit)** — это фиксация изменений в истории проекта (изменения, которые внесены в индекс). Коммит хранит изменённые файлы, имя автора коммита и время, в которое был сделан коммит. Кроме того, каждый коммит имеет уникальный идентификатор, который позволяет в любое время к нему откатиться. Можете считать коммит этакой точкой сохранения.

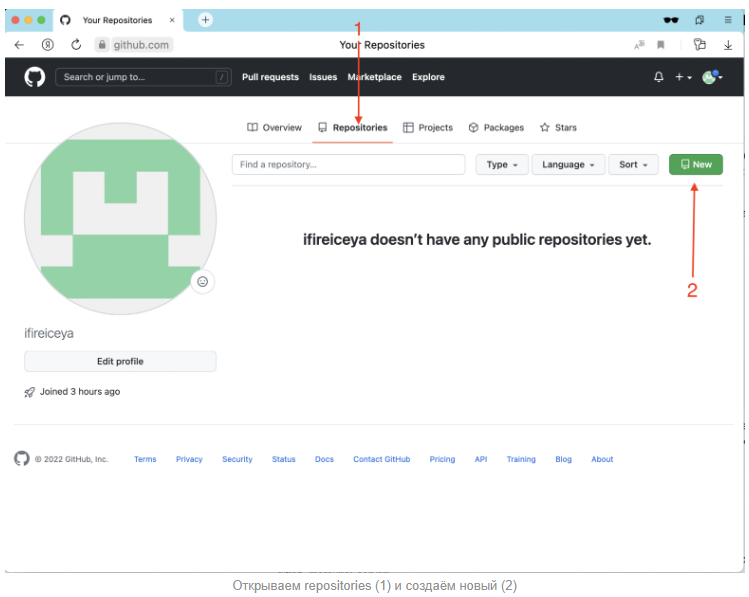
**Ветка (branch)** — последовательность коммитов. По сути — ссылка на последний коммит в этой ветке. Ветки не зависят друг от друга — можно вносить изменения в одну, и они не повлияют на другую (если вы явно этого не попросите). Работать вы начинаете в одной ветке — main, увидите чуть позже.

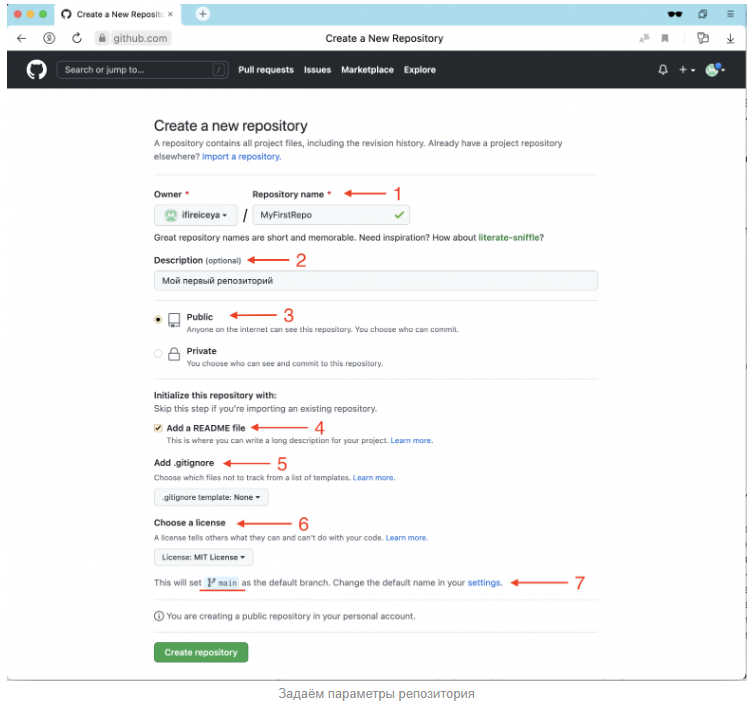
**Форк (Fork)** — собственное ответвление (fork) какого-то проекта. Это означает, что GitHub создаст вашу собственную копию проекта, данная копия будет находиться в вашем пространстве имён, и вы сможете легко делать изменения путём отправки (push) изменений.

**Пул-реквест — pull request PR (пиар, он же merge request MR(мр))** — предложение изменения кода в чужом репозитории. Допустим, вы забрали к себе чужой репозиторий, поработали с ним и теперь хотите, чтобы ваши изменения попали в оригинальный репозиторий — тогда вы создаёте создаёте PR с просьбой добавить ваши изменения в репозиторий.

### **Начало работы**

Начнём с простого — создадим свой репозиторий и сделаем наш первый коммит.

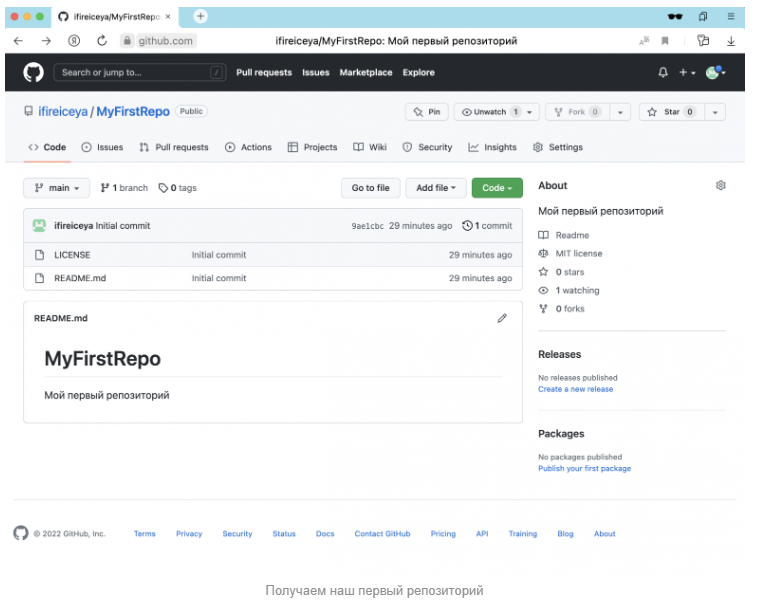




Зададим параметры:

* (1) Repository name: имя репозитория.
* (2) Description: описание репозитория.
* (3) Тип репозитория: Public (публичный) или Private (приватный). Сейчас выберем публичный — кто угодно может видеть содержимое репозитория.
* (4) Ставим галку на «Создать README файл». В этом файле в формате MarkDown описывают проект или прочую документацию. Именно содержимое этого файла можно увидеть, когда заходим на главную страницу репозитория. Примеры [1](https://github.com/prometheus/prometheus), [2](https://github.com/django/django), [3](https://github.com/ohmyzsh/ohmyzsh).
* (5) Если известно, на каком языке будет проект, можем добавить шаблон .gitignore для этого языка. Сейчас у нас нет какого-то языка, поэтому не будем создавать .gitignore.
* (6) Выбираем [тип лицензии](https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository) для нашего кода. В лицензии оговариваются права на проект. Стоит обратить внимание на [BSD 3](https://opensource.org/licenses/BSD-3-Clause) или [MIT](https://opensource.org/licenses/MIT), так как они предоставляют хороший баланс прав и ответственности.

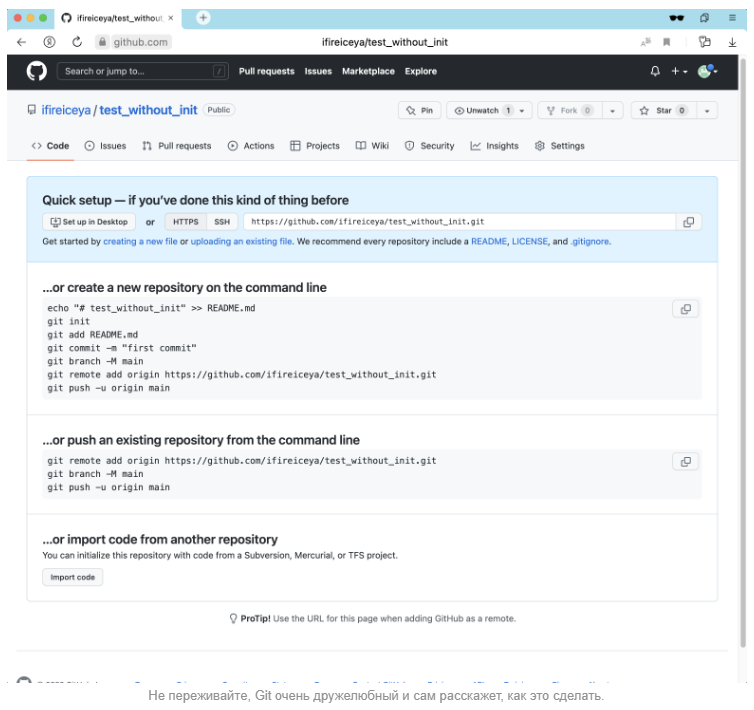
(7) По умолчанию имя основной ветки в GitHub носит имя main, но до [недавнего времени](https://www.theserverside.com/feature/Why-GitHub-renamed-its-master-branch-to-main) было master.



И нажимаем кнопку «Create repository». Успех, у нас есть первый репозиторий!

#### **А что будет, если не добавим README и .gitignore?**

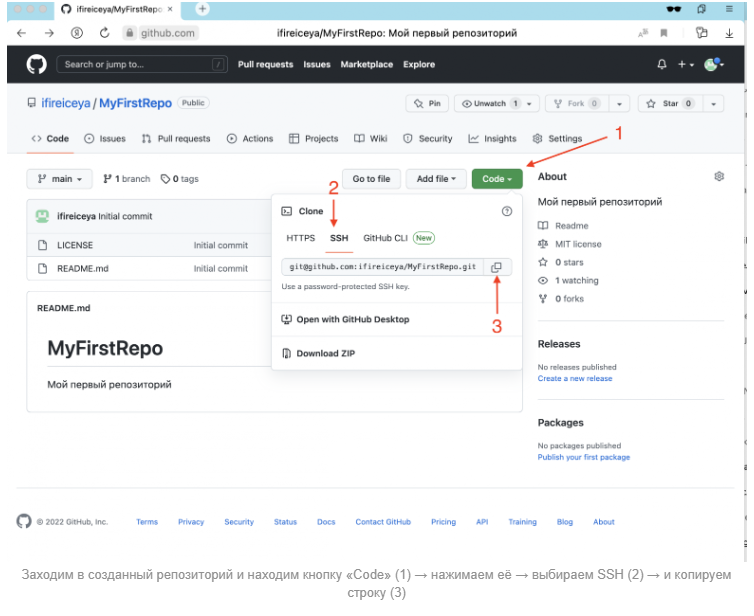
На самом деле ничего страшного не произойдёт, но придётся выполнить ещё ряд шагов, чтобы [проинициализировать](https://git-scm.com/docs/git-init) git-репозиторий, прежде чем начать с ним работать.



Итак, мы создали репозиторий на удалённом сервере, теперь пора «забрать» его к себе на локальную машину и внести какие-то изменения.

Чтобы забрать репозиторий, его надо склонировать к себе при помощи команды git clone и пути до репозитория.

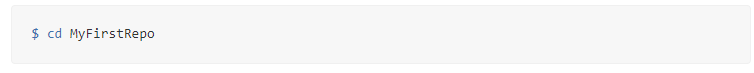
Для начала получим путь до репозитория.



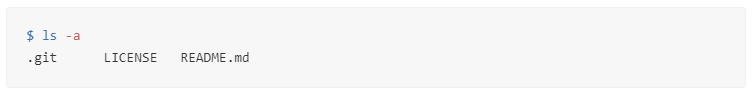
Теперь идём в консоль, переходим в директорию, где хотим хранить проекты, и выполним (git@github.com:ifireiceya/MyFirstRepo.git — путь, который мы скопировали ранее):



Переходим в новый каталог, где теперь лежит копия нашего проекта с GitHub:

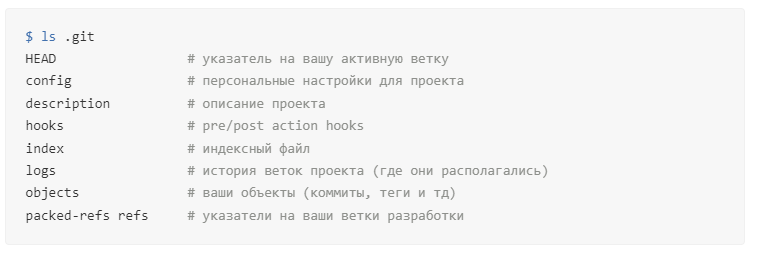


Можем посмотреть, что уже есть в этой директории:



Видим два знакомых файла,LICENSE и README.md, а также одну скрытую директорию .git.

В .git хранится метаинформация и вся история для проекта. На каждый проект есть только одна директория .git, и лежит она в корне проекта.

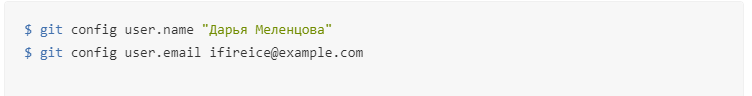


Давайте немного настроим Git под себя. Делать это нужно только один раз, потом настройки сохранятся, но при необходимости их можно изменить.

При установке Git была добавлена утилита git config, которая позволяет просматривать и изменять большинство параметров работы Git’а. Если речь о данных пользователя или способе работы репозитория — git config будет самым удобным способом настройки.

Настроим имя пользователя и адрес электронной почты. Эта информация важна, потому что включается в каждый коммит.

Поэтому в терминале переходим в Git-репозиторий, для которого задаём настройки, и выполняем:



Чтобы настраивать ещё больше параметров с помощью git config, прочитайте [эту документацию](https://git-scm.com/docs/git-config).

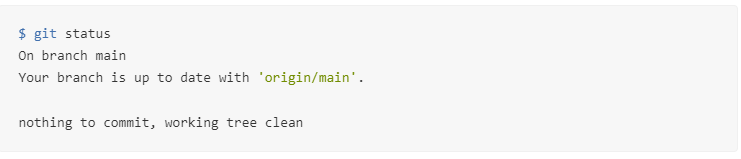
#### **Вносим изменения**

Теперь нужно внести изменения в проект. Но перед этим посмотрим две полезных команды:

* git status — показывает текущее состояние файлов в репозитории (какие файлы изменились, удалились, добавились);
* git log — показывает историю изменений (это про зафиксированные изменения, то есть коммиты).

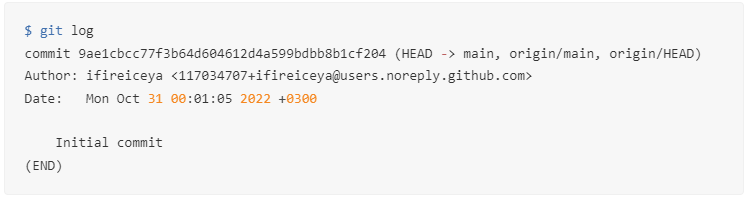
Выполним эти команды и посмотрим, что они выведут для нашего репозитория.

Вбиваем git status:



И видим, что у нас нет изменений. Говорят, «нет коммитов в репозитории». Конечно, мы успели только клонировать репозиторий и ещё ничего не делали.

Идём дальше и пробуем git log, который покажет, что в проекте был только один Initial commit — когда мы создавали репозиторий с README.md:



Убедились, что у нас нет неучтённых изменений. Пора бы уже что-то сделать!

Открываем любимый текстовый редактор и создаём новый файл с именем hw.py.

Это будет небольшая программа на Python, которая при запуске печатает «Hello World!» (внезапно):

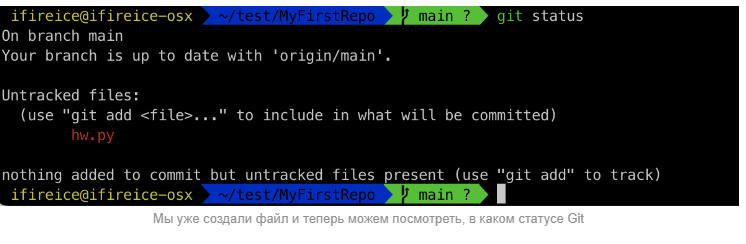


Отлично, код написан и даже хранится локально в нашем репозитории (мы же в директории проекта всё делали).

Теперь наша задача — сохранить изменения в «оригинальный» (удалённый) репозиторий. Для этого нужно:

1. Познакомить Git с новым файлом, то есть добавить файл в индекс — git add.
2. Зафиксировать (закоммитить) изменения — git commit.
3. Синхронизировать изменения с сервером — git push.
4. Посмотреть в репозиторий и убедиться, что всё сработало.

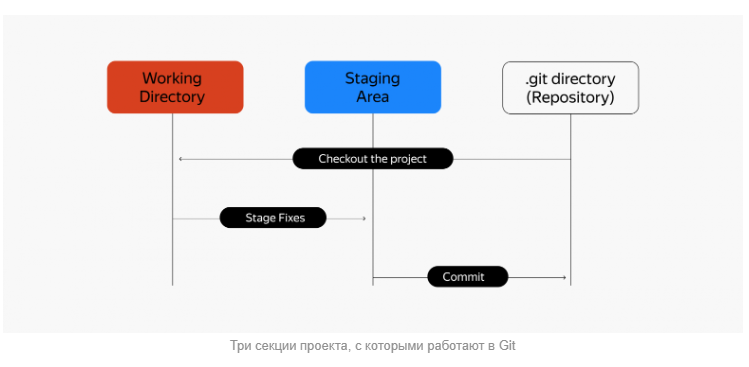
Делаем!

Появился файл hw.py, но он красный. Паника! Всё сломалось?!

Нет, всё идёт по плану, но прежде чем продолжить, стоит обсудить состояние файлов с точки зрения Git’а.

По мнению Git’а, файл может пребывать в одном из четырёх состояний:

1. Неотслеживаемый (untracked).
2. Изменённый (modified) — файл, в котором есть изменения, но он ещё не добавлен в коммит (не зафиксирован).
3. Отслеживаемый (staged) — файл, который добавили в индекс.
4. Зафиксированный (committed) — файл уже сохранён в локальной базе, и в нём не было изменений с последнего коммита.



В связке с состоянием файлов используют три основных секции проекта:

* **Рабочая директория (working directory)** — это директория, которая содержит в себе то, с чем вы работаете, или то, что вы извлекли из истории проекта в данный момент. Рабочая директория — это временное место, где вы можете модифицировать файлы, а затем выполнить коммит.
* **Область индексирования (staging area)** — индекс-файл в каталоге Git, который содержит информацию о том, что попадёт в следующий коммит.
* **Каталог Git** — место, где Git хранит метаданные и базу объектов вашего проекта. Помните ещё про .git?

#### **Что происходит на практике**

Мы добавили новый файл hw.py и видим, что у него состояние untracked, то есть неважно, что мы делаем с файлом, Git проигнорирует любые изменения в нём.

Чтобы Git начал следить за изменениями в файле, его нужно добавить в индекс.

Для этого используем команду git add <имя файла>.

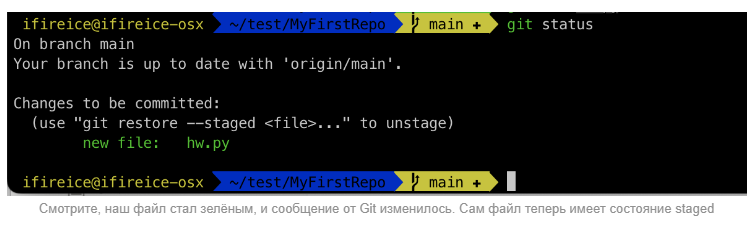
Кстати, вы заметили, что Git довольно дружелюбный и часто подсказывает команды, которые нужно выполнить?



И ещё не забывайте о файле .gitignore, где перечислены папки и файлы репозитория, которые Git не должен отслеживать и синхронизировать их состояние (не добавлять их в индекс). Обычно в него добавляют файлы логов, результаты сборки и другое. Поддерживает [шаблоны](https://git-scm.com/docs/gitignore). Кстати, .gitignore — тоже файл, который надо добавить в индекс.

* Если файл попадает в правила.gitignore, то он не появится в git status.
* Если файл был добавлен в индекс, а потом добавлено правило для файла в .gitignore — файл всё равно будет отслеживаться и его надо явно [удалить](https://git-scm.com/docs/git-rm) из индекса.

Посмотрим, как изменилось состояние нашего файла:



Давайте зафиксируем изменения, так как наша задача сейчас решена: мы написали программу на Python и хотим сказать Git, что вот теперь мы закончили работать с файлом и надо запомнить текущее состояние.

Для этого нужно закоммитить файл с помощью команды git commit.

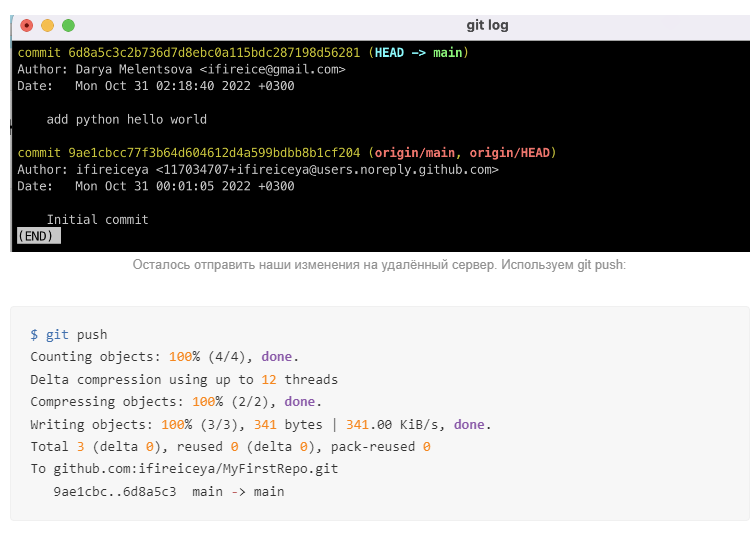
При создании обычно лаконично описывают коммит, используя ключ -m:



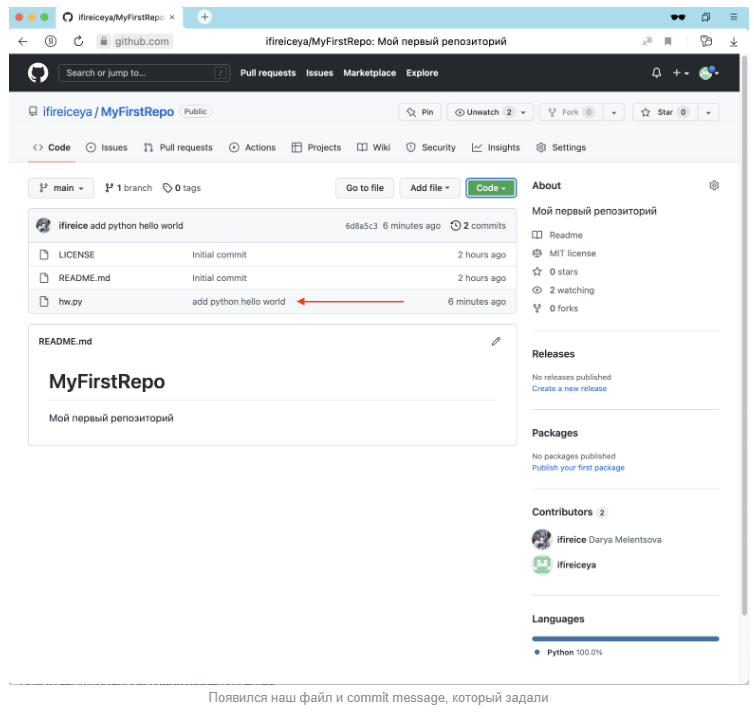
Пара слов о том, как писать сообщения для коммитов:

* максимум 50 символов;
* осознанно и понятно, как будто пишете для человека, который должен понять, что происходит внутри коммита;
* сообщение стоит начинать с заглавной буквы;
* если меняли код, пишите исходный код в сообщении.





Предлагаем проверить, что наши изменения есть на GitHub. Идём в репозиторий и смотрим на него.



### **Задача немного посложнее**

Всё здорово, но мы не всегда создаём репозитории, и часто нам нужно добавлять новые фичи или исправления в уже существующий репозиторий, да ещё и в чужой.

Например, есть у нас любимый опенсорсный проект, в который мы хотим принести добро и закрыть им какой-нибудь [Issue](https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues).

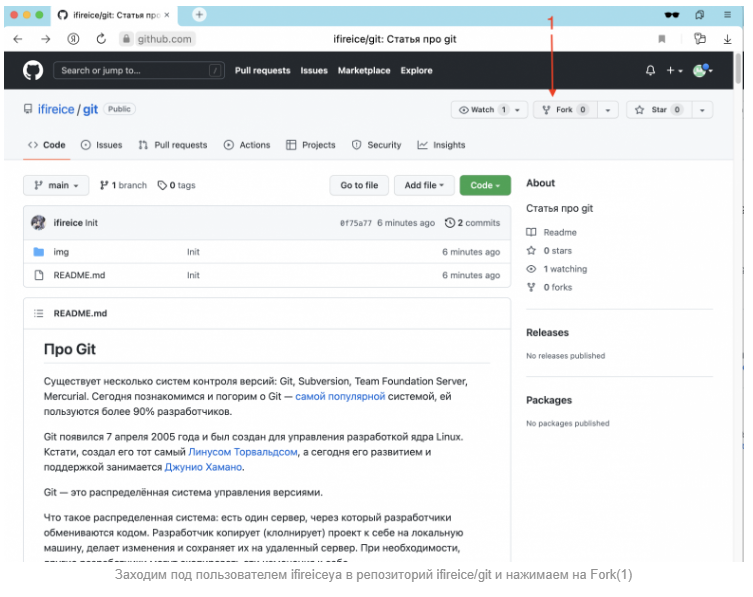
В учебных целях используем репозиторий из этой статьи на GitHub. Там нужно исправить опечатку, которую нашли в статье. Например, вот эту очепятку.

Но будем делать это с позиции внешних пользователей в чужом репозитории.

Репозиторий хранится в ifireice/git, а изменения делает пользователь ifireiceya.

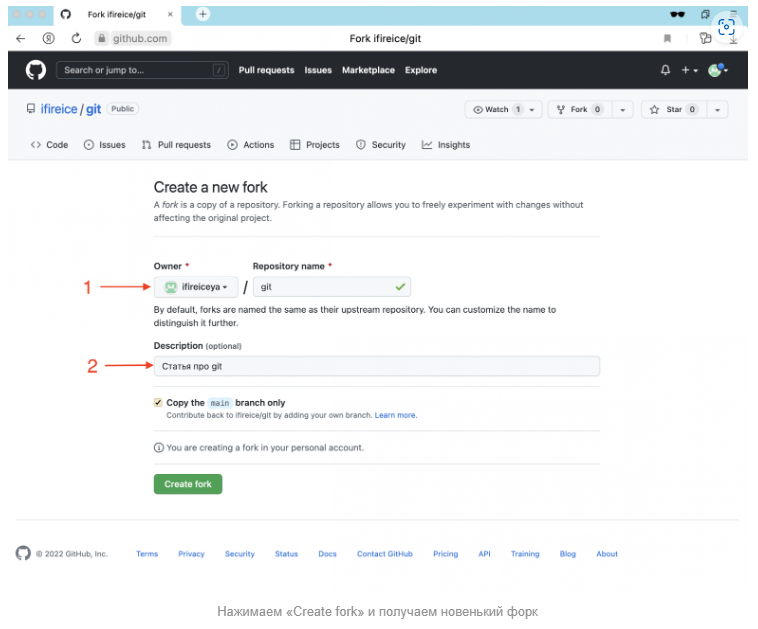
Пользователь ifireiceya не имеет доступа в ifireice/git, и ему придётся работать через Fork, то есть нужно сперва сделать копию этого репозитория к себе и вести разработку у себя, а потом отправить в основной репозиторий запрос на изменения — Pull Request.

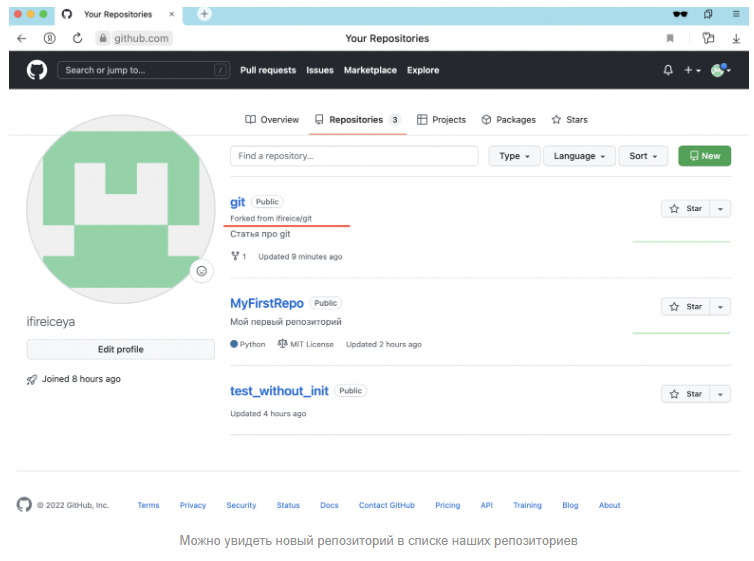
Но обо всём по порядку. Сначала делаем Fork.



Откроется окно для создания нового форка (fork).

Изменится владелец репозитория (1), и опционально можно изменить описание проекта.





Вы можете делать любые изменения в собственной копии, и они никак не отразятся в оригинальном репозитории.

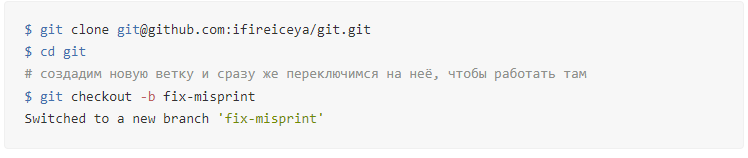
Теперь клонируем форк-репозиторий к себе на машинку и ведём разработку.

Только мы будем работать чуть-чуть по-другому, не как с нашим репозиторием.

В нашем репозитории мы работали в ветке main и все изменения сохраняли в ней.

А теперь у нас большой проект, и над ним одновременно могут трудиться несколько разработчиков. Чтобы разные изменения не смешивались в кучу и чтобы один разработчик не мешал другому, разработка ведётся в разных независимых версиях продукта — ветках (branch). Когда работа закончена, все изменения сливаются в одну главную ветку.

Клонируем репозиторий и создаём отдельную ветку, в которой будем устранять опечатку:

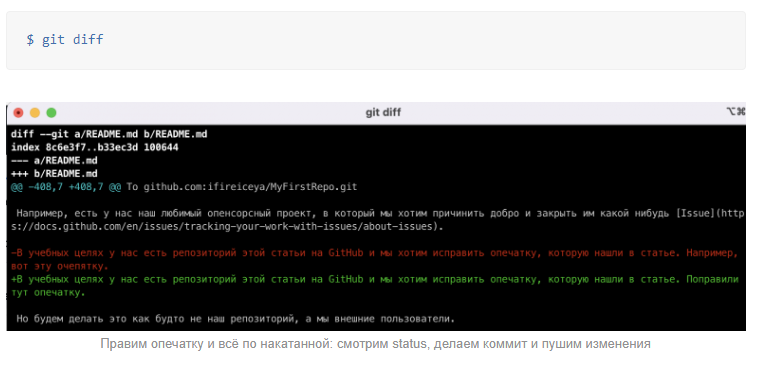


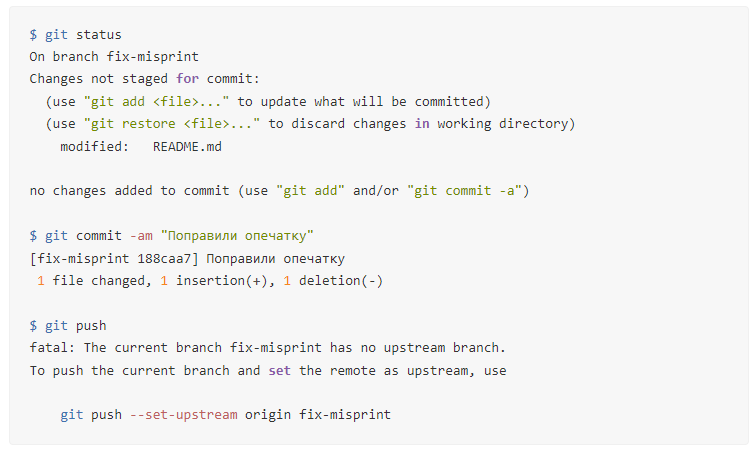
Чтобы посмотреть, какие ветки есть в проекте и какая сейчас активна, используется команда git branch:



На самом деле практика работать с ветками распространена не только при разработке в чужих репозиториях (collaborators), куда у вас нет доступа, но и в своих. Есть несколько стратегий выделения веток, но об этом не сейчас. Просто знайте, что есть ветки и с их помощью удобно вести разработку.

Можем посмотреть, что изменилось с последнего коммита, при помощи команды git diff (красным с “-” то, что было, зелёным с “+” — то, что стало):





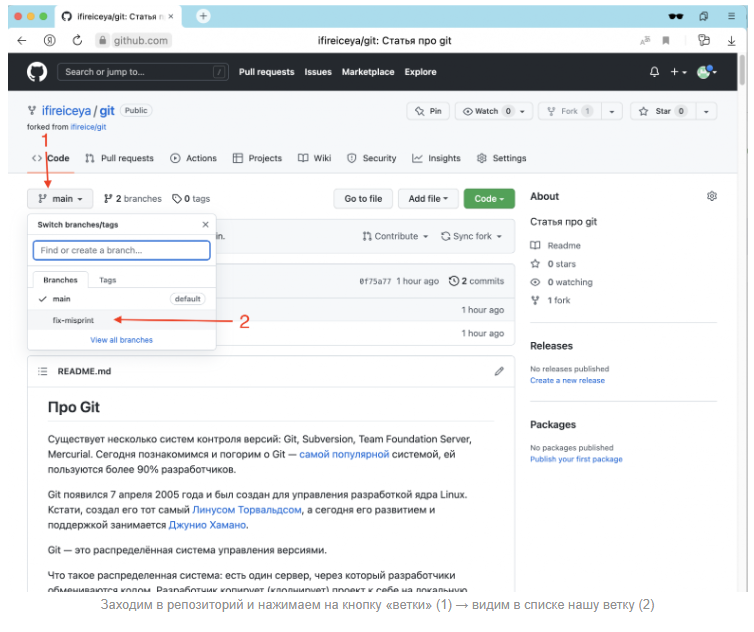
Упс, fatal. Читаем подсказку от Git и выполняем:



Успех!

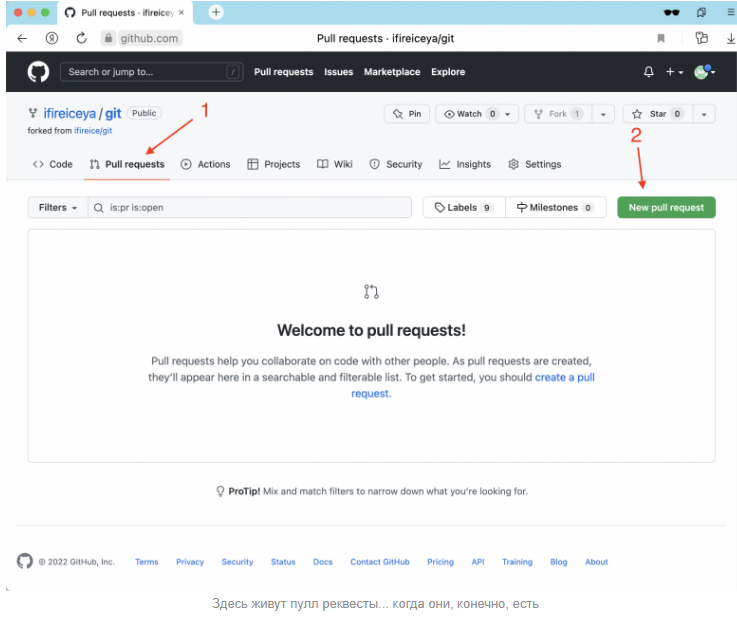
Почему произошёл fatal: простой git push предполагает, что ветка, которую отслеживает текущая локальная ветвь, уже существует на удалённом сервере. У нас ветка новая и была создана только локально, поэтому нам нужно её создать, указав --set-upstream.

Проверим, что ветка появилась на GitHub.



Форк сделали, ветку отвели, ошибку поправили, осталось отправить изменения в оригинальный репозиторий.

Для этого создаём Pull request.



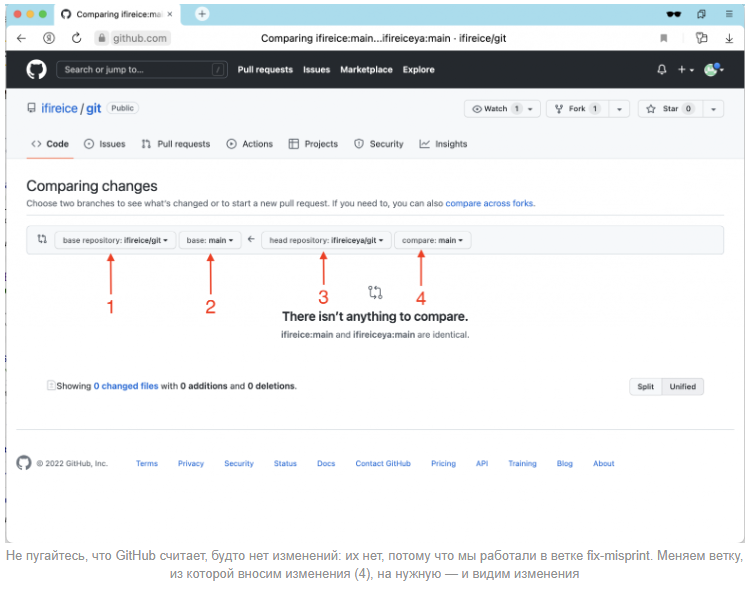
И увидим такую картину.

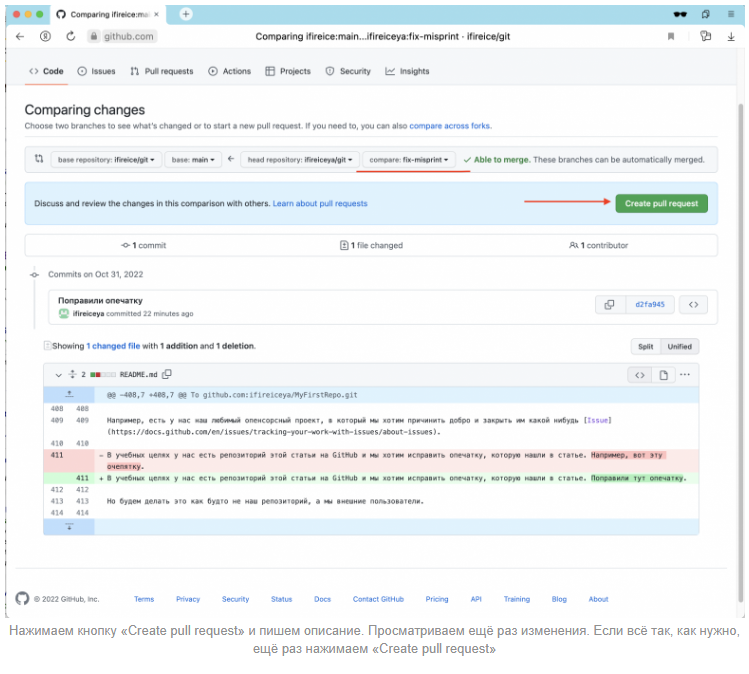
(1) репозиторий, в который хотим добавить изменения. — ifireice/git

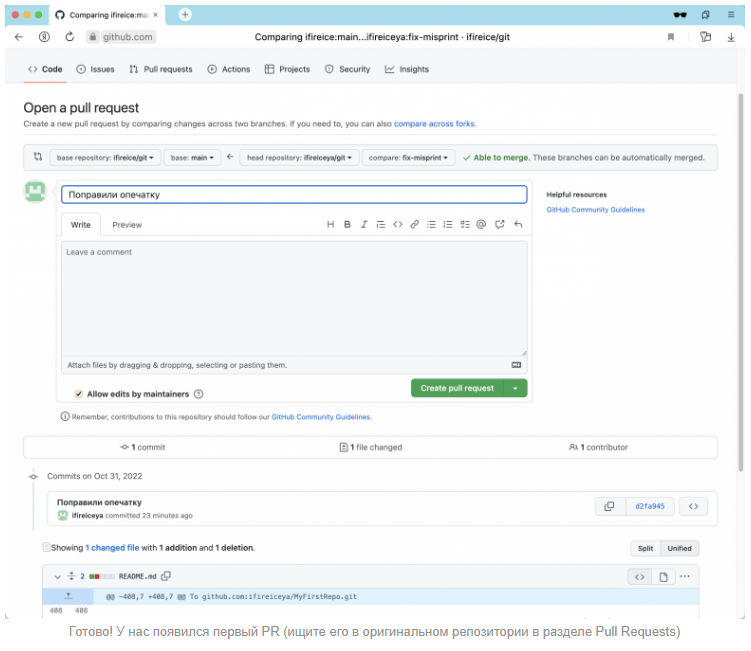
(2) ветка, в которую хотим добавить изменения — main

(3) репозиторий, из которого хотим добавить изменения — ifireiceya/git

(4) ветка, из которой хотим добавить изменения — main



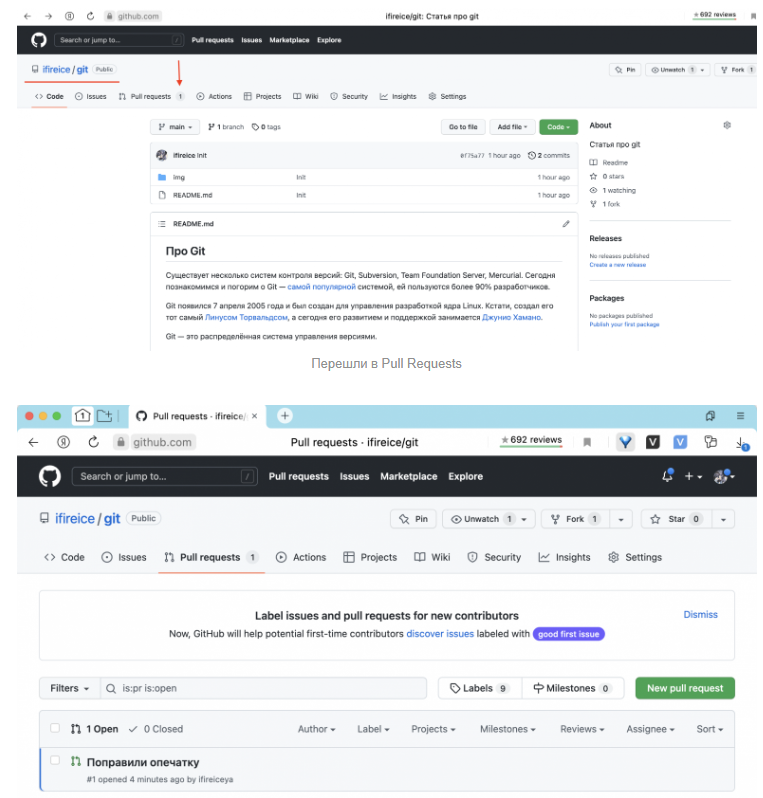


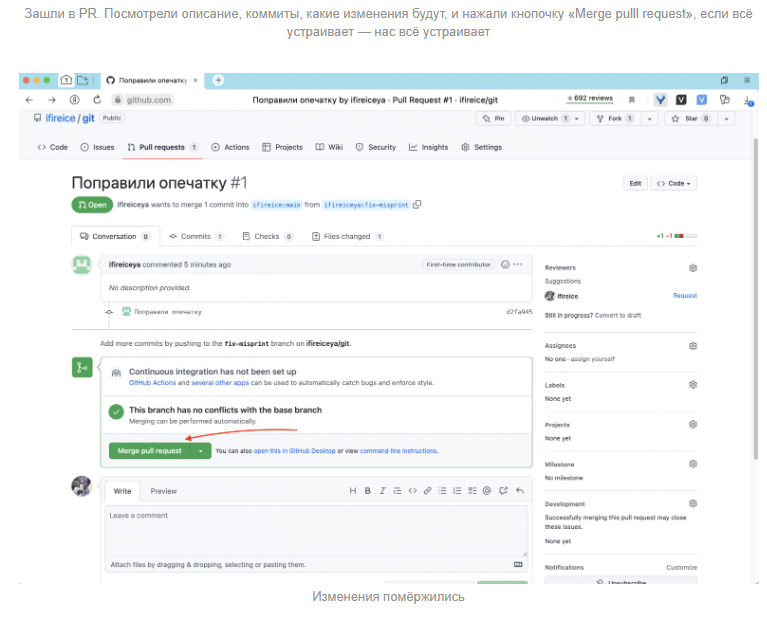


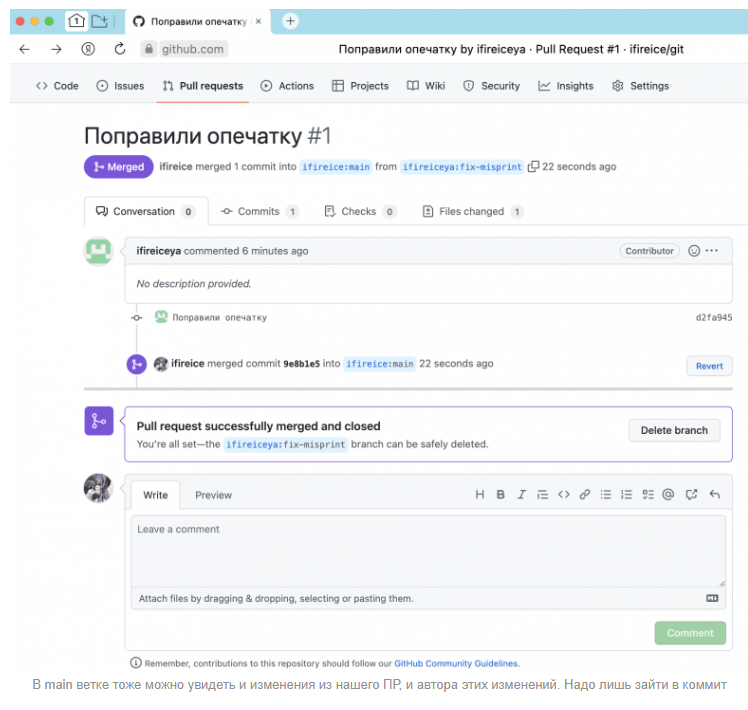
На этом сейчас наша работа завершена. Ответственные за репозиторий посмотрят ваши изменения, примут их, или попросят что-то дописать, или отклонят изменения.

Будем считать, что у нас всё хорошо и наши изменения приняли без вопросов.

Как это выглядит на стороне ревьюверов:









### **Что изучили**

* Поговорили про системы контроля версий
* Настроили себе GitHub
* Создали первый репозиторий и внесли в него изменения
* Узнали про ветки, форки и остальное
* Сделали первый ПР

### **Что НЕ изучили**

* [Конфликты](https://git-scm.com/book/en/v2/Git-Tools-Advanced-Merging)
* Откат изменений ([reset](https://git-scm.com/docs/git-reset), [revert](https://git-scm.com/docs/git-revert))
* Как забрать файл из другой ветки ([Cherry-pick](https://git-scm.com/docs/git-cherry-pick))
* [rebase](https://git-scm.com/docs/git-rebase.html)

### **Что ещё почитать**

* Отличная книга [Pro Git](https://git-scm.com/book/ru/v2) (есть на русском и английском)
* [Курс «DevOps для эксплуатации и разработки»](https://practicum.yandex.ru/promo/devops-course/?utm_source=pr&utm_medium=content&utm_campaign=pr_content_devops-course&utm_content=22november2022)
* [Trunk-Based Development](https://habr.com/ru/company/avito/blog/680522/) — стратегия отведения веток
* [GitFlow](https://github.com/SergeFocus/git-flow) — другая стратегия отведения веток