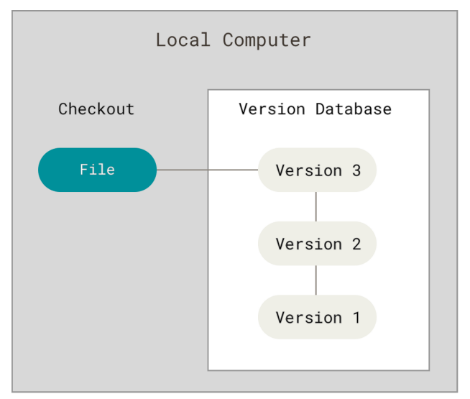
План

1. Что такое система контроля версий и для чего она нужна
2. История возникновения Git
3. Плюсы Git
4. Установка Git на Windows
5. Основные команды Git
6. Удаленный сервер репазиторий GiHub

**Что такое система контроля версий и для чего она нужна?**

Системы контроля версий (сокращено ВКС) - это система, которая отслеживает и запоминает все изменения в файле или наборе файлов с течением времени, чтобы в случае необходимости можно было сделать шаг назад и отменить изменения, вернувшись на предыдущие этапы разработки.

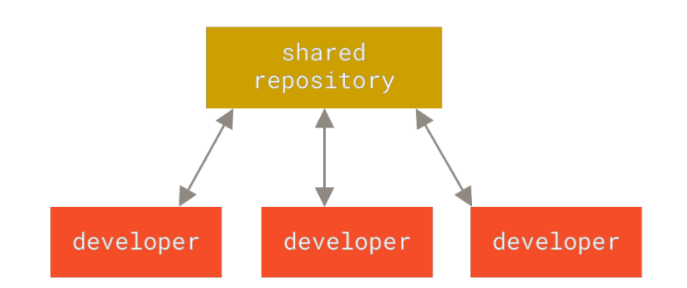
Разработка каждого большого проекта всегда разбивается на этапы для упрощения выполнения поставленной задачи, следовательно, изменения в файле проекта будут вноситься постепенно. Однако что если возникнет необходимость отменить изменения из-за серьезного бага и откатиться на предыдущие версии проекта, а мы уже сохранили проект, закрыли и обычный Ctrl+Z не сработает? Помимо этого, программный продукт, как правило, разрабатывается не одним специалистом, а целой группой разработчиков, которые отвечают за реализацию разного функционала. Они могут выполнять свою работу из разных мест и в разное время. Как бы максимально ускорить и упростить процесс слияния их наработок в единую конструкцию? Со всем этим прекрасно справляется система контроля версий. Она помогает команде разработчиков эффективно передавать и отслеживать все изменения, внесенные в файл проекта, а также информацию о том, кем и какие изменения были сделаны.



Типы систем контроля версий:

* Локальные ВКС
* Централизованные ВКС
* Распределенные ВКС

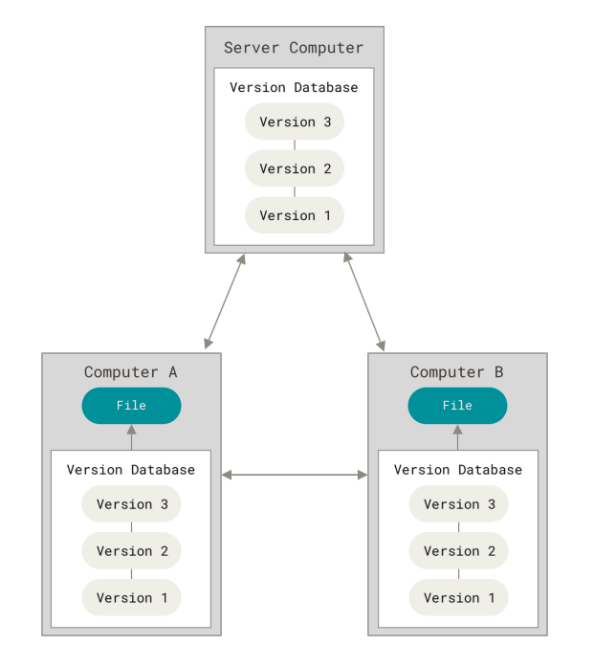
Локальные системы контроля версий – это одна из самых простых форм, имеющая базу данных, в которой хранятся все изменения файлов, на каком-то отдельном локальном компьютере.

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (Centralized Version Control System, далее CVCS)

Она использует единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение CVCS являлось стандартом на протяжении многих лет.

Такой подход имеет множество преимуществ, особенно перед локальными VCS. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать CVCS, чем оперировать локальными базами данных на каждом клиенте. Несмотря на это, данный подход тоже имеет серьёзные минусы.

Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков

Здесь в игру вступают распределённые системы контроля версий (Distributed Version Control System, далее DVCS). В DVCS (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.   
Так как в локальном репозитории храниться только копия, то изменения в нем никак не повлияют на основной репозиторий проекта до тех пор, пока вы сами того не пожелаете, и, как только вы посчитаете это необходимым, вы запросто можете внести изменения в главный репозиторий. Другие пользователи не будут видеть ваших изменений, пока они не решат обновить их локальный репозиторий на удаленный.  
Одной из наиболее популярных распределенных ВКС является Git.

**Рождение Git**

Как и многие вещи в жизни, Git начинался с капелькой творческого хаоса и бурных споров. Ядро Linux — это достаточно большой проект с открытым исходным кодом. Большую часть времени разработки ядра Linux (1991–2002 гг.) изменения передавались между разработчиками в виде патчей и архивов. В 2002 году проект ядра Linux начал использовать проприетарную децентрализованную систему контроля версий BitKeeper. Несмотря на то, что BitKeeper являлся платным для использования, разработчикам Linux удалось договориться о использовании этой ВКС бесплатно. Однако в какой-то момент это соглашение было расторгнуто. Эндрю Триджелл создал клиент с открытым исходным кодом для доступа к системе контроля версий Bitkeeper, переработав протоколы BitKeeper. Это заставило правообладателя отказаться от ранее согласованной политики свободного использования, после чего многие разработчики Linux отказались от использования BitKeeper.

Это сподвигло сообщество разработчиков ядра Linux (а в частности Линуса Торвальдса — создателя Linux) разработать свою собственную утилиту, учитывая уроки, полученные при работе с BitKeeper. Некоторыми целями, которые преследовала новая система, были: • Скорость • Простая архитектура • Хорошая поддержка нелинейной разработки (тысячи параллельных веток) • Полная децентрализация • Возможность эффективного управления большими проектами, такими как ядро Linux (скорость работы и разумное использование дискового пространства)

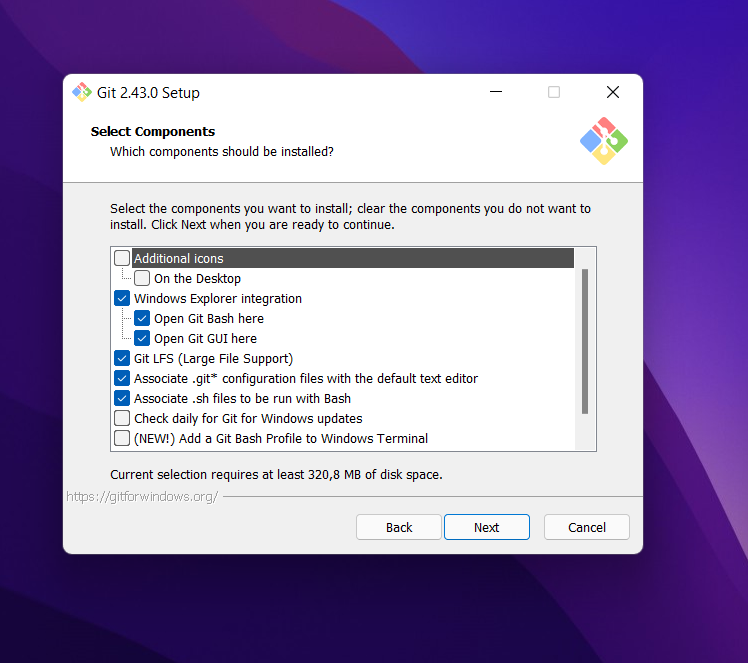
С момента своего появления в 2005 году, Git развился в простую в использовании систему, сохранив при этом свои изначальные качества. Он удивительно быстр, эффективен в работе с большими проектами и имеет великолепную систему веток для нелинейной разработки. С 2005 года 26 июля сопровождение было передано Джунио Хамано, который внес большой вклад в развитие проекта (отвечал за выпуск 1.0) и остается основным сопровождающим проекта. Git стал широко известен и на его основе в 2007 году был создал портал GitHub.

**Плюсы Git**

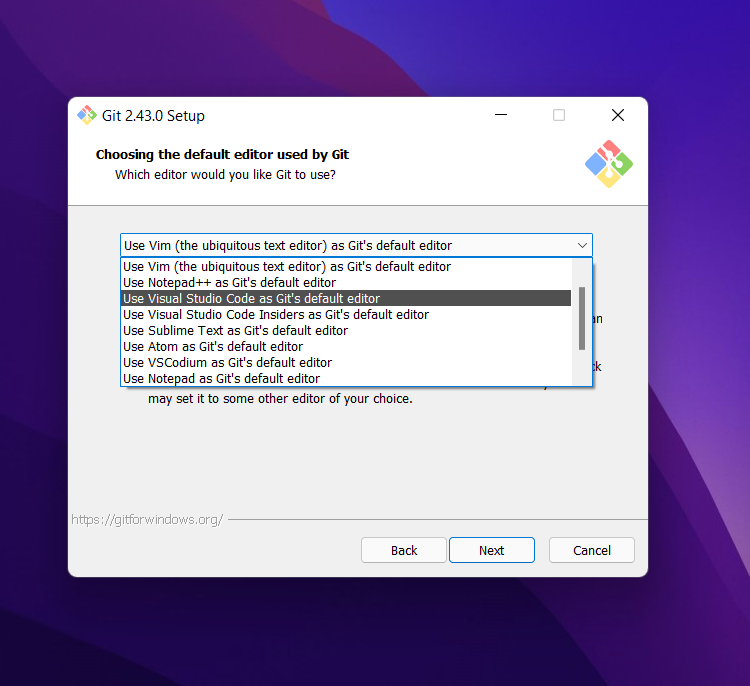
1. Первое, что хотелось бы отметить, это его доступность и открытость. Гит является бесплатная распределенная система контроля версий с открытым исходным кодом.
2. Безопасность. Одним из критериев Линуса при разработке ВКС была защита как от случайных, так и от злоумышленных повреждений файлов репозитория. Используя Git, вы можете быть уверены, что у вас есть подлинная история содержимого вашего исходного кода.
3. Гибкость. Гибкость Git проявляется в нескольких аспектах: в поддержке различных видов нелинейных рабочих процессов разработки, в эффективности при реализации как небольших, так и крупных проектов, в совместимости со многими существующими системами и протоколами.
4. Полная фиксация изменений. Например, операции, влияющие на ветвления и теги (например, слияние или возврат), также сохраняются как часть истории изменений. Не все системы контроля версий поддерживают такой уровень отслеживания.
5. И т.д.

**Установка Git на Windows**

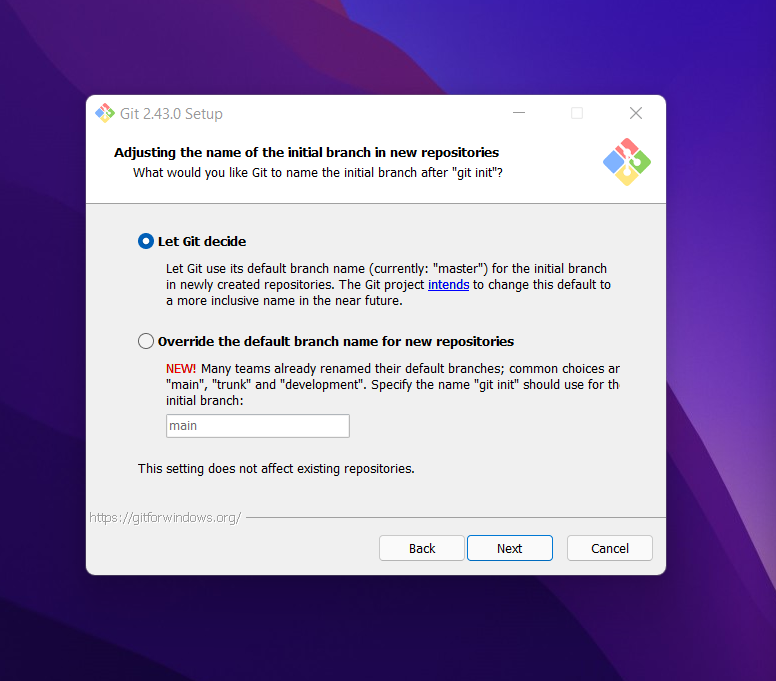
Сама установка Git на операционную систему Windows не является чем-то сверхъестественным, его можно запросто скачать с официального сайта git-scm.com. Однако мы остановимся на этом моменте, чтобы дать вам рекомендации по установке.



При установке гита советуем вам выбирать те же флажки, что и на слайде. Включение флажка Windows Explorer integration является особенно удобным, так как он позволяет нам не прописывать путь к папке с репозиторием, а просто открыть гит в нужной папке через нажатие на папку правой кнопкой мыши.

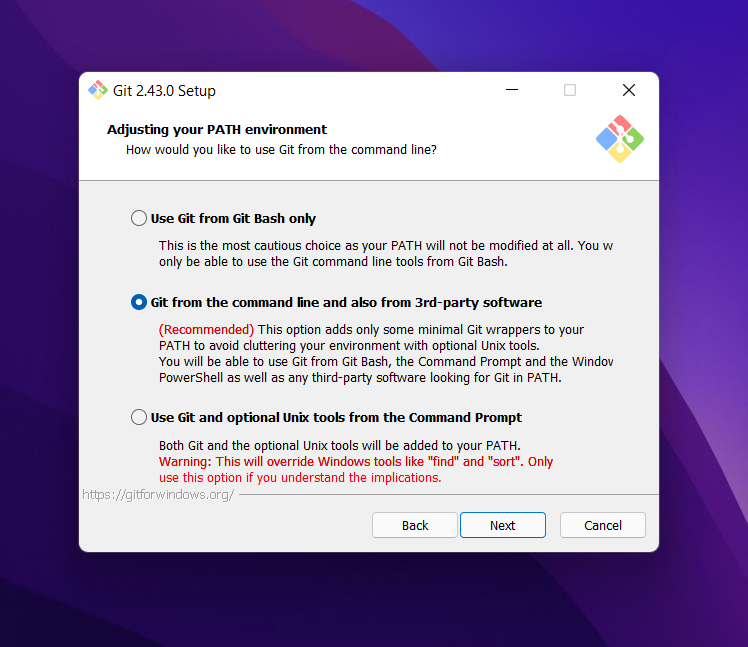


Необходимо выбрать редактор, который будет использовать Git — например, когда вы будете писать сообщение для коммита. Это не обязательно должен быть редактор, в котором вы планируете писать исходный код. По умолчанию в установщике выбран Vim — консольный текстовый редактор, который для многих может показаться сложным в освоении. Если вы не знакомы с Vim и при этом хотите именно консольный редактор — выберите nano. Если у вас уже установлен какой-нибудь текстовый редактор — выбирайте его. Мы для примера будем использовать VSCode.

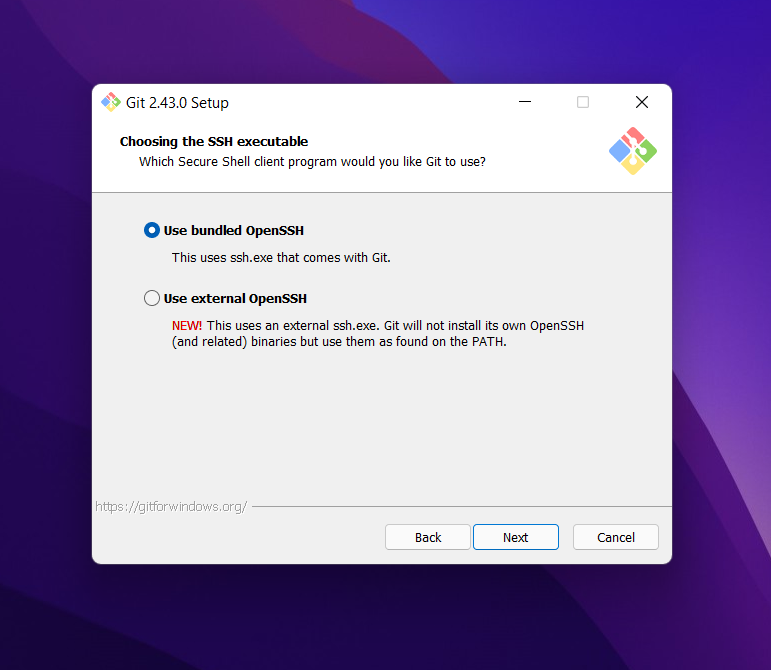


Тут нужно выбрать, как Git будет называть первую ветку в каждом репозитории. Раньше такая ветка всегда называлась master, но со временем это стало напоминать о временах рабства, и многие проекты и компании стали переименовывать ветки в своих репозиториях. Поэтому разработчики Git добавили эту опцию, чтобы название первой ветки можно было изменить. Мы будем придерживаться старого поведения и оставим название master.

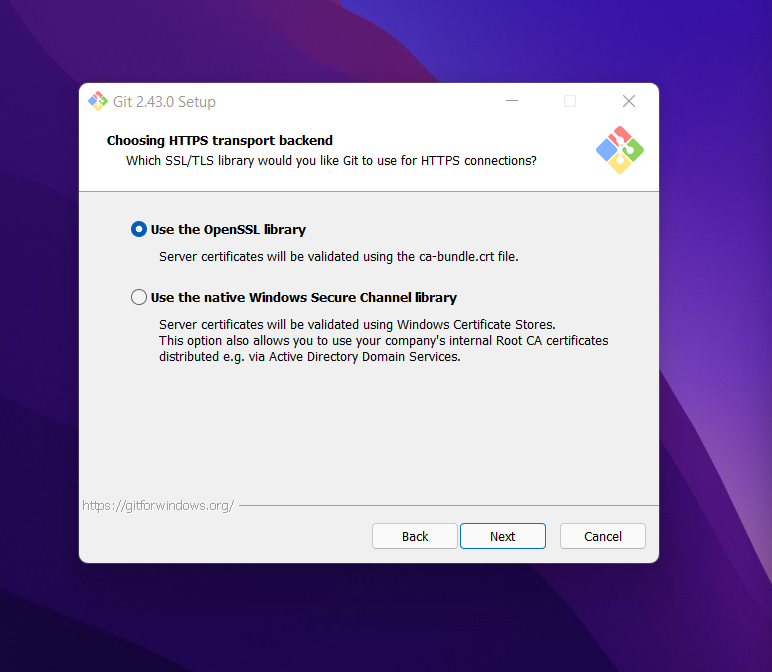
В следующем окне выберите нужно разрешить использовать Git не только через устанавливаемый Git Bash, а еще и через командную строку самой Windows



Изначально Git поставлялся со встроенным SSH-клиентом, но недавно появилась опция, где можно использовать внешний клиент. Если у вас уже что-то установлено на компьютере — можете выбрать вторую опцию. Мы же остановимся на первой, так как предварительно ничего не устанавливали.

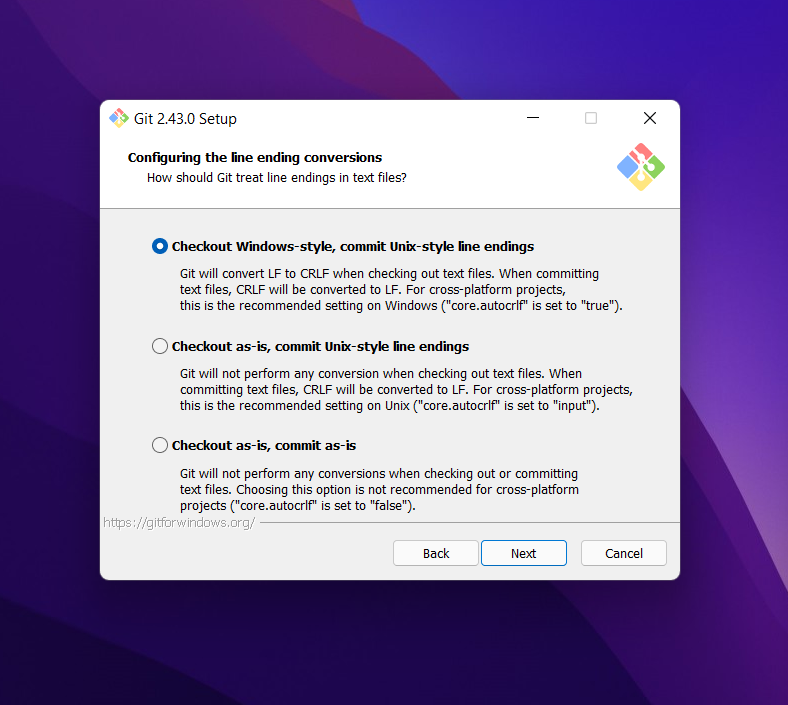


По умолчанию Git будет использовать свою OpenSSL библиотеку с заранее определенным списком корневых сертификатов. Обычно этого достаточно, но если вам нужно работать со внутренними репозиториям внутри компании, которые используют самоподписанные сертификаты, выберите вторую опцию. Тогда Git будет использовать библиотеку и сертификаты из вашей операционной системы.

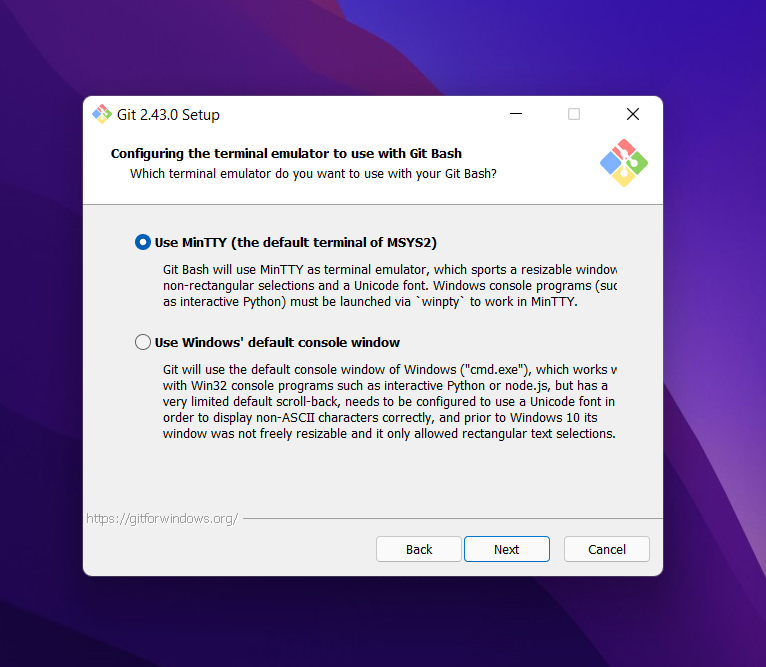


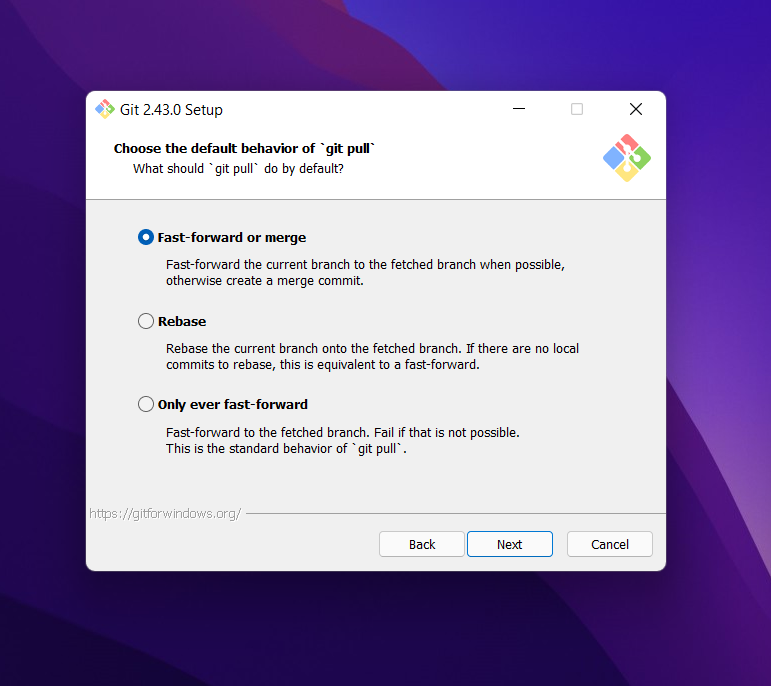
Существует два основных способа формирования конца строки в файлах — CRLF и LF. Первый используется в Windows, второй — в Unix-like системах. Первая опция позволяет извлекать файлы из репозитория в Windows-стиле, при этом отправлять файлы в репозиторий в Unix-стиле. Мы рекомендуем использовать этот вариант, потому что он лучше всего подходит для кросс-платформенной команды, когда над одним кодом могут работать разработчики на разных ОС.

Следующее окно у вас будет спрашивать каким задавать конец строки при вводе команд. Мы советуем вам выбрать дефолтное значение. Это убережет вас от проблем в будущем при работе в мультиплатформенных репозиториях



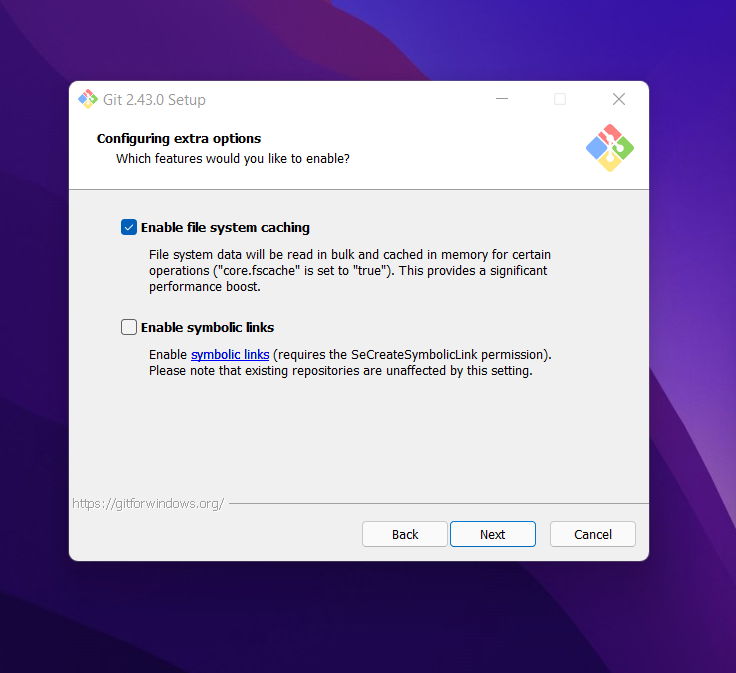
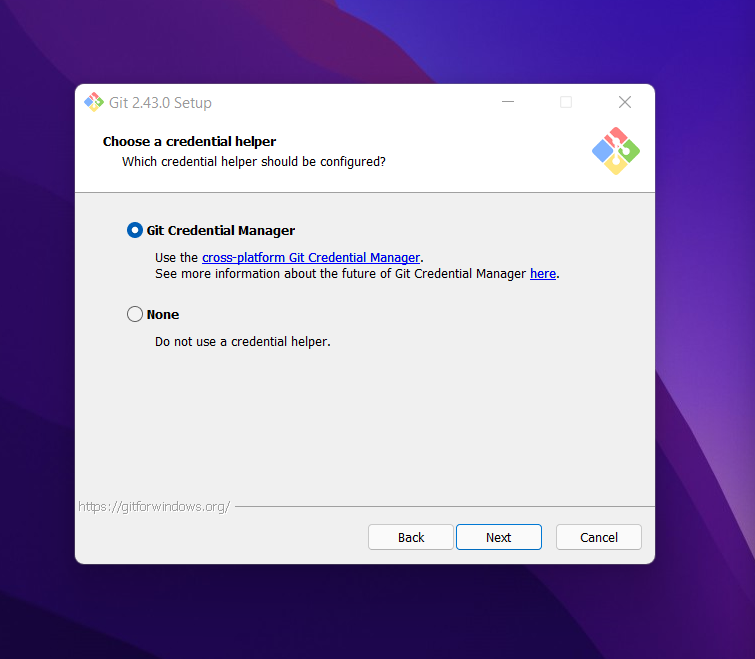
Теперь пришло время выбрать эмулятор терминала для Git. Мы рекомендуем использовать MinTTY, так как это очень хорошая оболочка, полностью настраиваемая и удобная в использовании

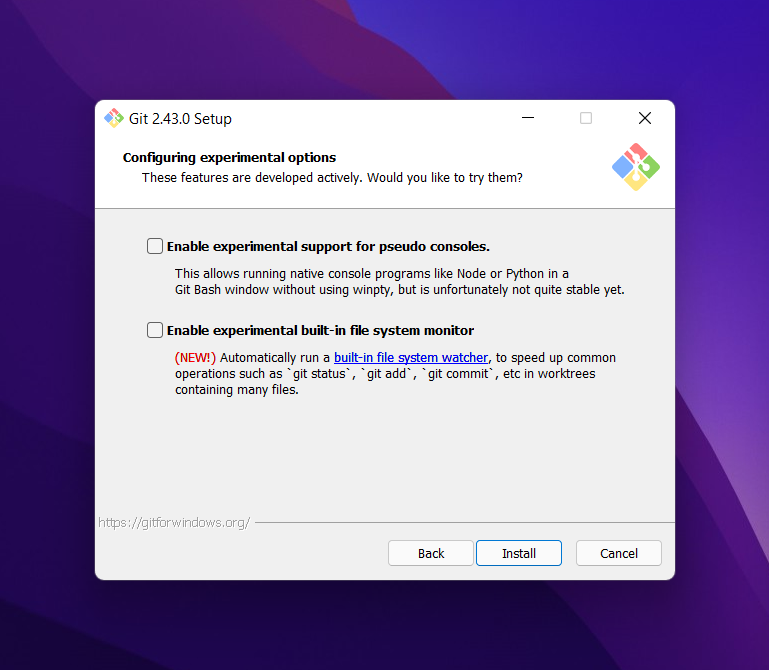




Первая опция будет пытаться обновить историю коммитов без создания коммитов слияния. Это самый оптимальный и часто используемый вариант, оставим его.

Далее нам будут предлагаться на выбор дополнительные функции Git. Мы рекомендуем вам поставить флажок на **File system caching**, она позволяет Git быть быстрее при работе с основными операциями чтения/записи, а также **Git Credential Manager**, она предоставляет как можно более надежное хранение учетных данных, в частности многофакторной аутентификации для Visual Studio Team Services и GitHub.





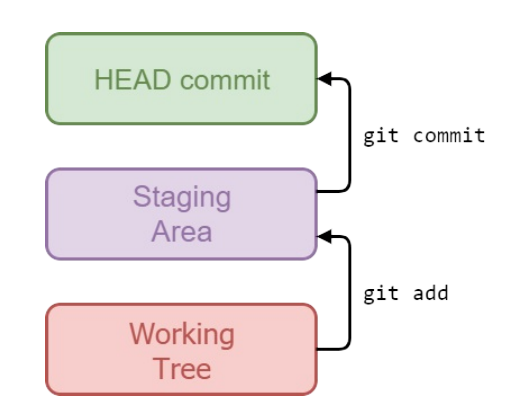
Эти опции еще не переведены в стабильную стадию, поэтому их использование рекомендуется, только если вы точно понимаете, что делаете. Мы не будем ничего отмечать.

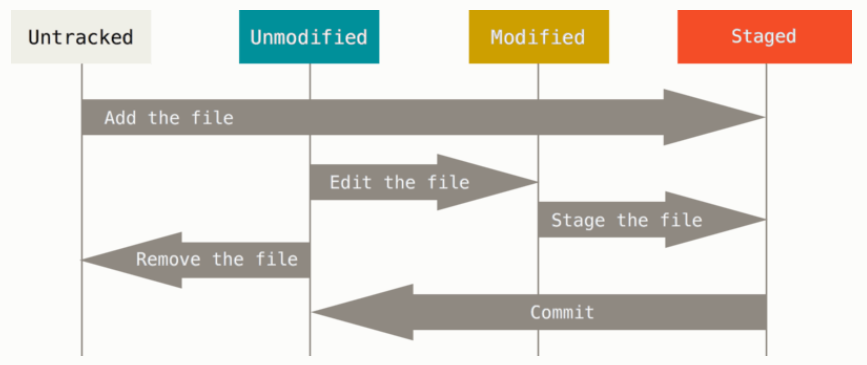
**Основные команды Git**

Изучение гита – это тема не одного часа и не одного дня. Поэтому сейчас мы покажем вам основные функции управления репозиторием при помощи данной системы контроля версий.

Сначала мы расскажем, как выглядит рабочая зона у гита. Гит делится на три зоны: The working tree (or working directory); The staging area; The HEAD commit (или последний коммит на ткущей ветке).

Когда мы модифицируем файл, мы делаем это на уровне The working tree; после мы фактически копируем изменения от The working tree к The staging area. Когда мы делаем фиксацию изменений, мы переносим изменения из The staging area в память репозитория и это становится главным версией, на которую ссылается HEAD, которая станет частью нашего хранилища.





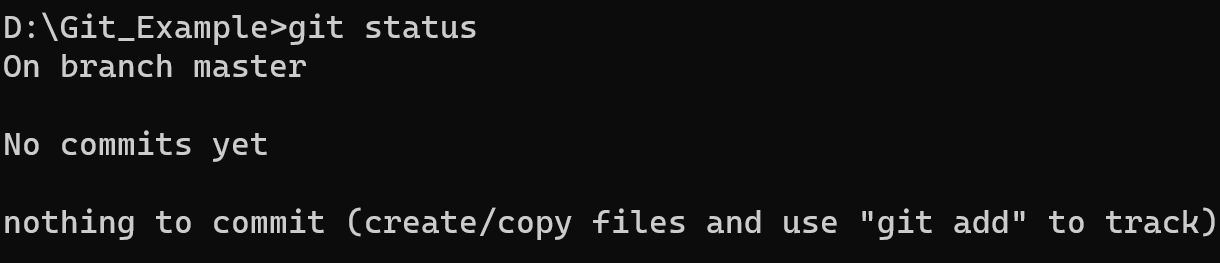
Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы — это те файлы, о которых знает Git.

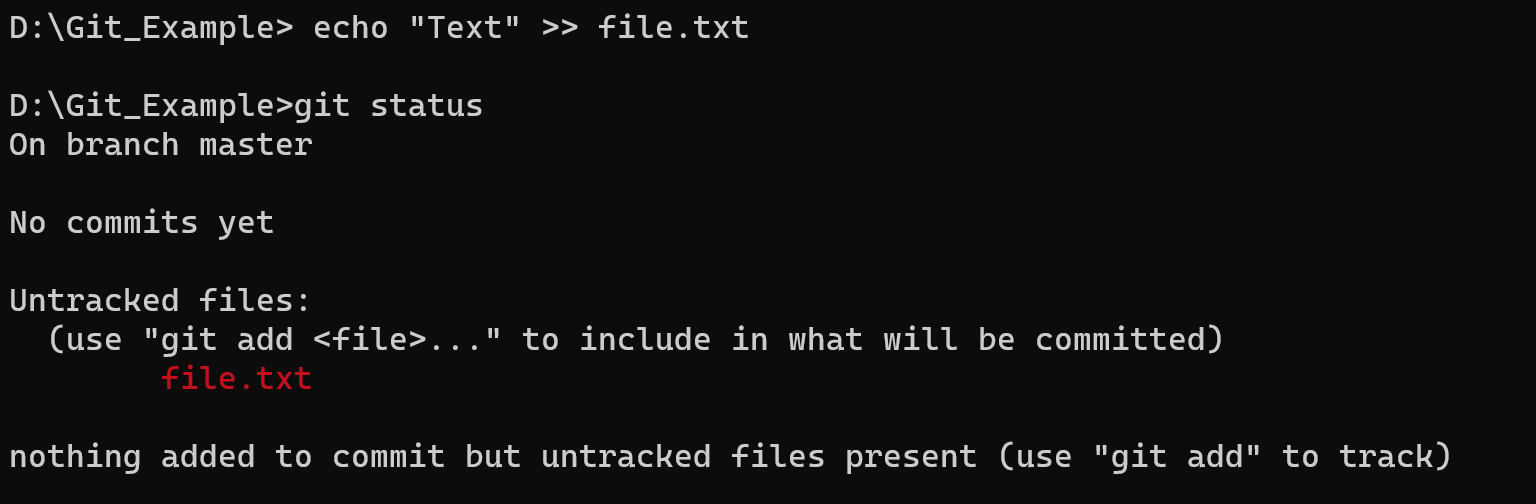
Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что Git только что их извлек и вы ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.

1. Для начала узнаем, как создать репозиторий при помощи гит. Если у вас уже есть проект в каталоге, который не находится под версионным контролем Git, то для начала нужно перейти в него. Если вы не делали этого раньше, то для разных операционных систем это выглядит по-разному



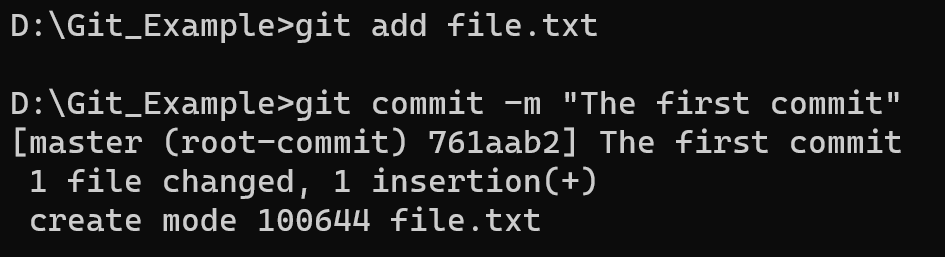
1. Следующая базовая команда является “git status”. Она нам показывает состояние репозитория: были внесены какие-то изменения в папку проекта или нет Здесь мы видим информацию о текущем состоянии репозитория.
2. Сейчас давайте внесем изменения и посмотрим на такую команду как “git add”

Предположим, вы добавили в свой проект новый файл file.txt. Если этого файла раньше не было, и вы выполните git status, вы увидите свой неотслеживаемый файл вот так:

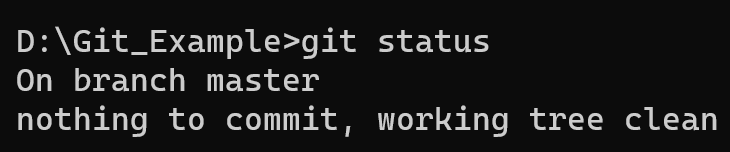
Теперь у нас есть какие-то изменения в нашем области The working tree

Понять, что новый файл неотслеживаемый можно по тому, что он находится в секции «Untracked files» в выводе команды status. Статус Untracked означает, что Git видит файл, которого не было в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить данный файл, так давайте сделаем это.

1. Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда git add. С помощью команды “git add” переместим файл с изменениями из The working tree в The staging area и потом добавим коммит с помощью команды “git commit”



Атрибут –m обозначает что мы будем создавать новый коммит. После этого атрибута должно следовать название коммита. Давайте взглянем как теперь выглядит рабочая область гита

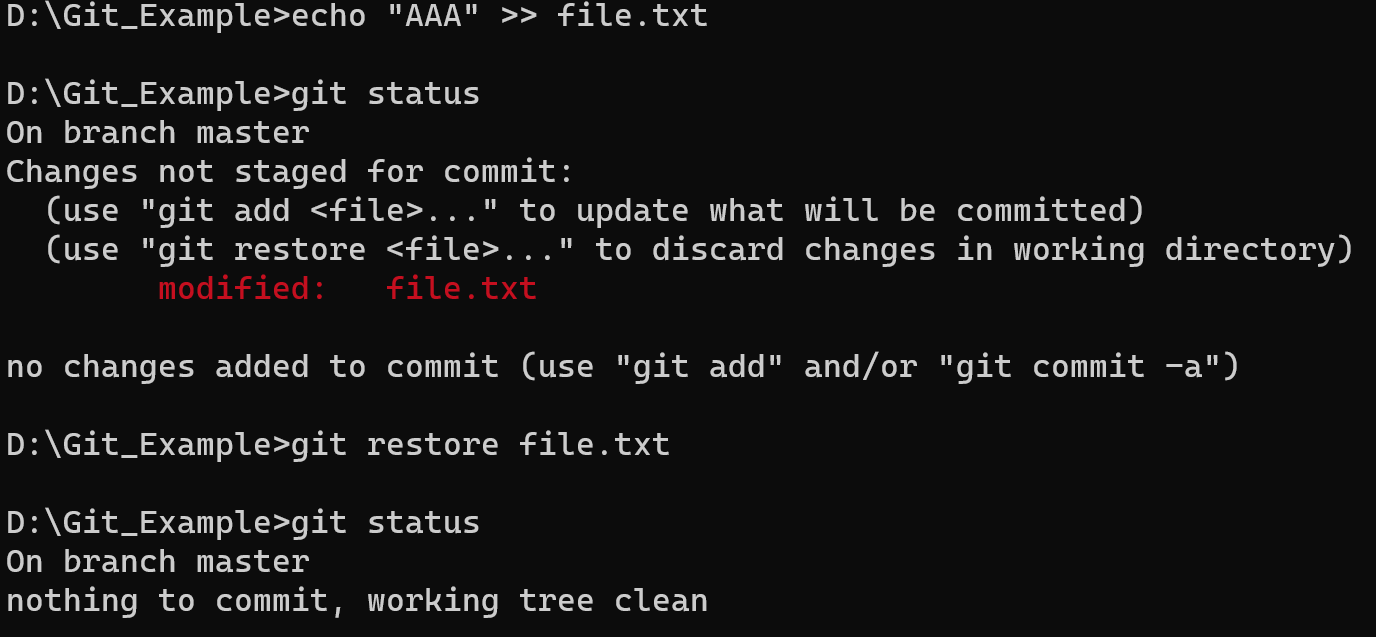
вызвав “git status”  Как видите, нам пишет, что рабочее дерево пустое.

Git add – это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если вы будете думать об этом как «добавить этот контент в следующий коммит», а не как «добавить этот файл в проект»

**Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. Добавление параметра -a в команду git commit заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без git add:**

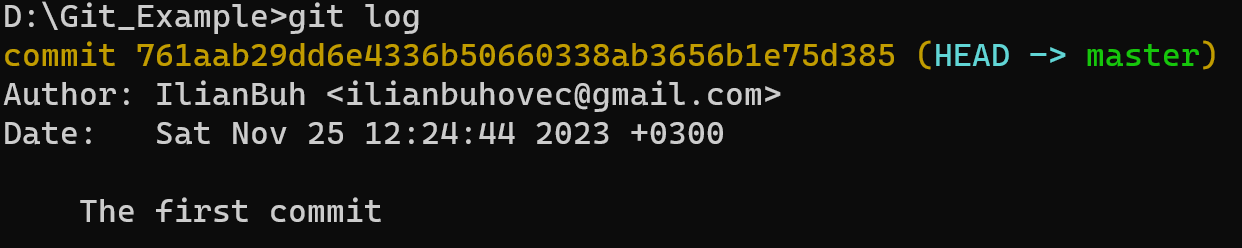
**Это удобно, но будьте осторожны: флаг -a может включить в коммит нежелательные изменения. (нужно показать)**

1. Если вы внесли измения и решили, что они лишние то в таком случае применяется команда “git restore”



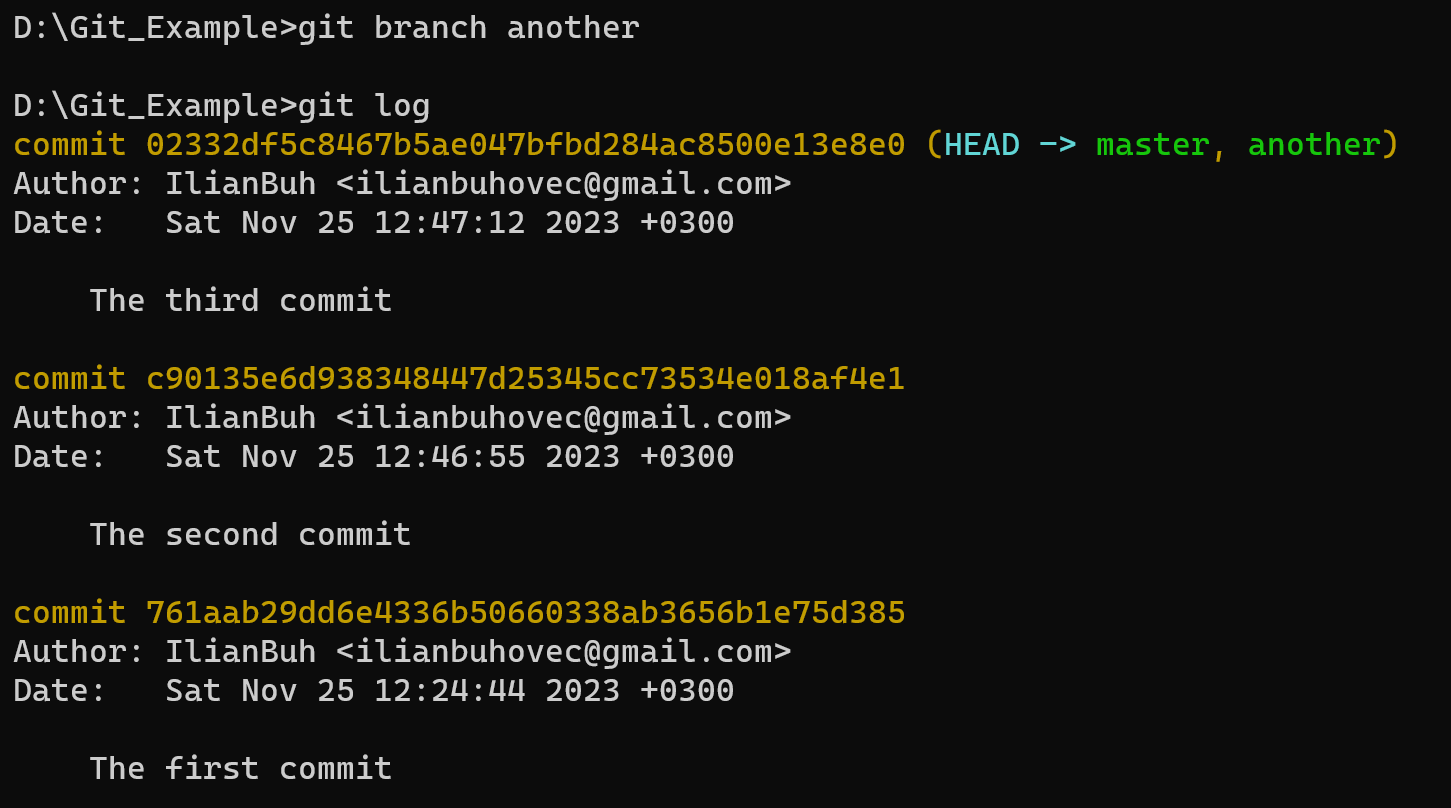
После этого все изменения в файле будут сброшены до состояния последнего коммита.

1. Следующая функция, которую мы продемонстрируем, это “git log”. Она показывает нам историю нашего репозитория



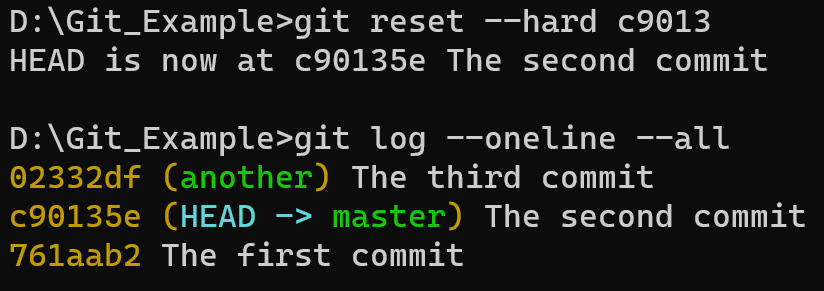
Здесь мы можем обнаружить хэш коммита, по которому мы можем указывать гиту, какую версию нам надо включить; автора коммита; дату, когда он был сделан и название коммита.

1. В гите возможны параллельно развивающиеся ветки изменения проекта, и для создания веток применяется команда “git branch Name”, а для удаления какой-то ветки применяется атрибут –d “git branch –d Name”



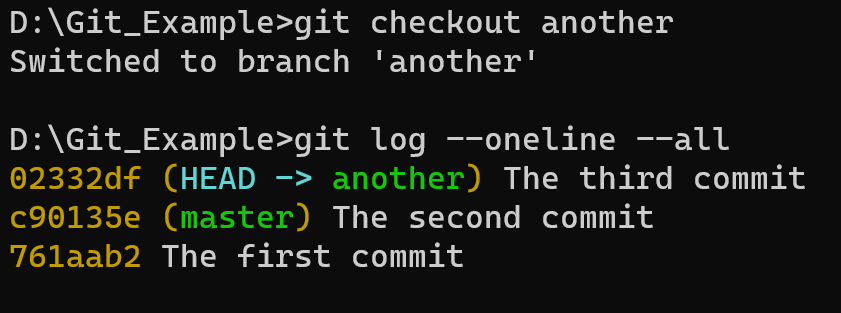
Здесь мы добавили к новую ветку в наш репозиторий, и теперь у нас появятся два пути по ветке “master” и “another”

1. Сейчас мы находимся в ветке “master”. Давайте представим, что нам надо вернуться ко второму коммиту. Это можно сделать с помощью команды “git reset –hard “хэш коммита””. Как хэш коммита достаточно ввести первые пять символов, необязательно всю последовательность символов, и гит поймет, что именно вы хотите получить



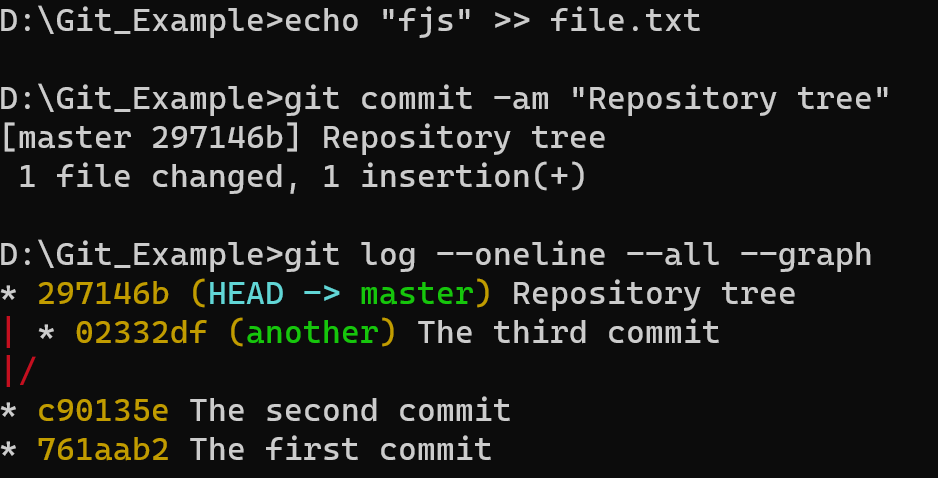
Таким образом для ветки “master” мы вернулись к предыдущей версии, но на ветке “another” мы все еще находимся на последнем сделанном нами ранее коммите.

1. Чтобы переключиться к ветке “another” можно использовать команду “git checkout another”



Теперь файлы нашего проекта имеют свою самую последнюю версию

Давайте теперь посмотрим как выглядит разветвление репозитория. Вернемся на ветку “master” и создадим новый коммит.



Вот будет выглядеть разделение путей репозитория. Теперь у нас есть две ветви, которые будут изменяться независимо друг от друга

Сам гит предоставляет еще множество различных команд таких как “git diff”, “git merge”, “git cherry-pick” и т.д., однако мы не сможем их всех вам показать, так как на это уйдет не 10 минут и даже не один час. Это тема, которая заслуживает долгового и кропотливого изучения. А мы перейдем к нашей последней теме на сегодня. И это совместимость с таким сервером как GitHub

Можно добавить про игнор, диф (про рм точно нужно, может это даже важнее веток, ещё про переммещение файлов)

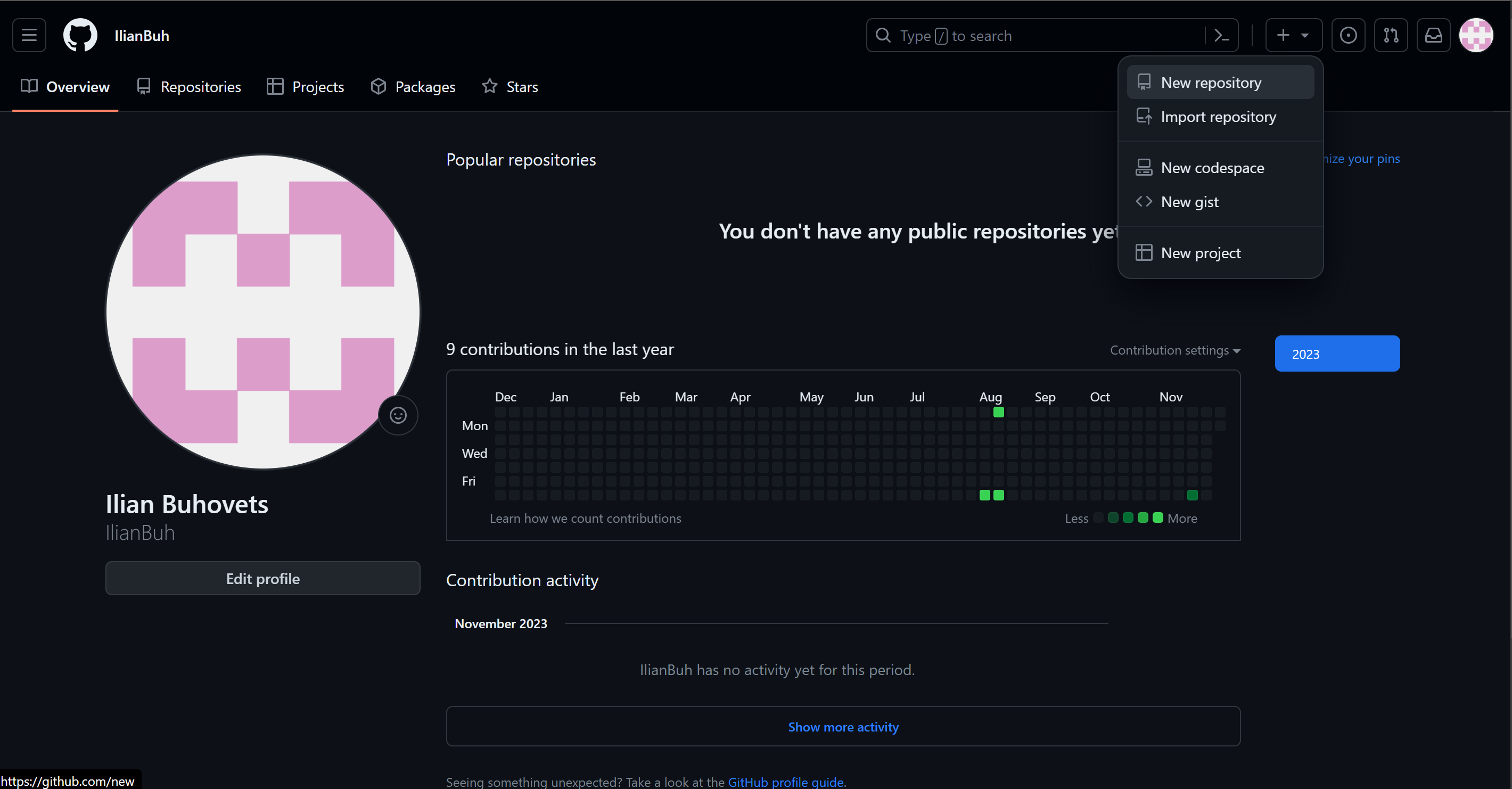
**Работа с GitHub**

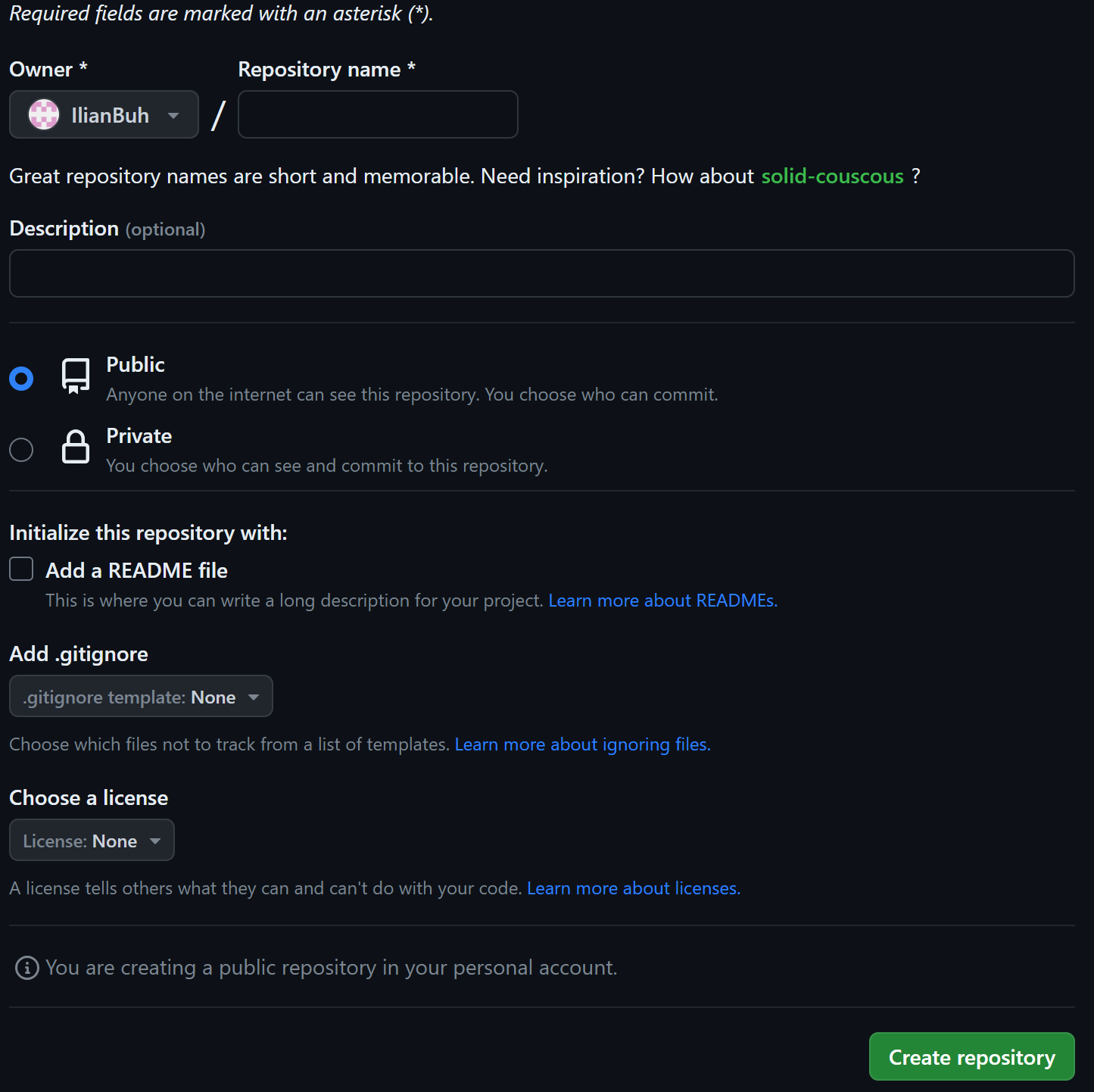
GitHub — это крупнейшее хранилище Git репозиториев, а так же центр сотрудничества для миллионов разработчиков и проектов. Огромный процент всех репозиториев хранится на GitHub, а многие проекты с открытым исходным кодом используют его ради Git хостинга, баг-трекера, рецензирования кода и других вещей. Так что, пока всё это не часть открытого Git проекта, наверняка вы захотите, или вам придётся взаимодействовать с GitHub при профессиональном использовании Git.

Смысл работы с GitHub заключается в том, что у нас имеется репозиторий на сервере, который мы клонируем к себе на компьютер, и по мере завершенности необходимых правок перезаписываем репозиторий на сервере на локальный.

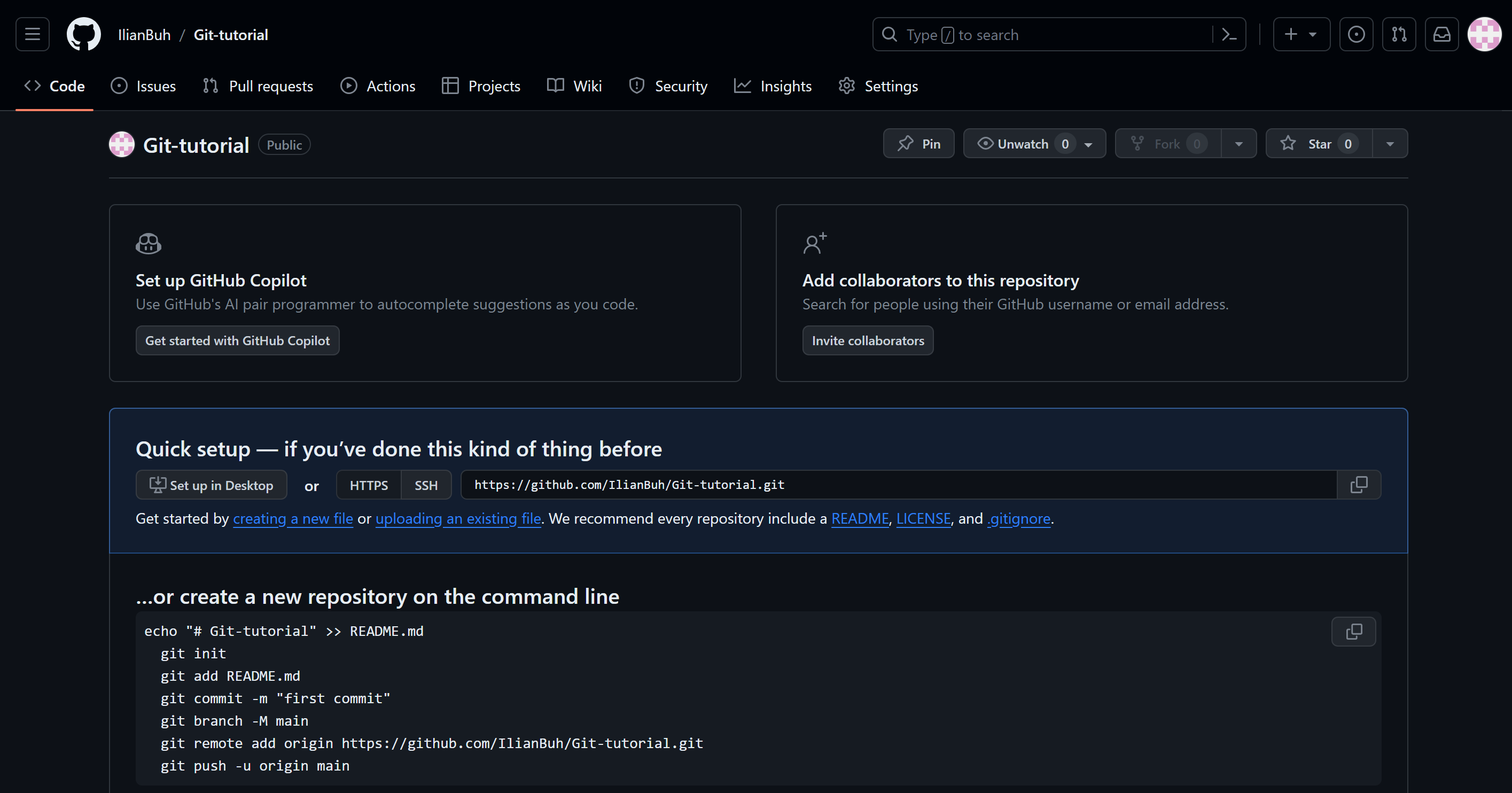
**Пример создания нового репозитория на GitHub**

Вызвать окно создания нового репозитория можно разными способами. Однако мы рассмотри вариант, как это можно сделать через домашнюю страницу. Вызовем необходимое окно через страницу аккаунта, нажав сначала на плюсик справа сверху, а затем выберем пункт «New repository».



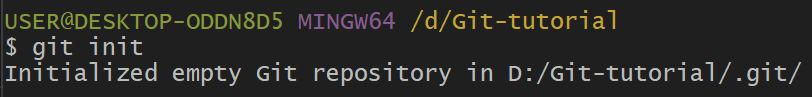


В всплывшем окне нас могут заинтересовать некоторые пункты. Первый пункт “Repository name”. В нем соответственно указывается имя нового репозитория. Далее следует«». Здесь мы можем описать цель создания данного репозитория. Потом мы можем сделать его публичным или частным. Публичные будут видеть все пользователи гитхаба, в то время как частный смогут увидеть только те, кому вы позволите, стоит уточнить, что частный репозиторий доступен только по платной подписке. Теперь у вас может возникнуть вопрос: что это за файл такой README, который гитхаб предлагает нам добавить сразу с созданием нового репозитория? На самом деле это обычный текстовый документ, в котором вы можете оставлять свои записи для описания чего-то. Теперь небольшое отступление насчет .gitignore. Дело в том, что не всегда нам требуется, чтобы файлы попадали в коммиты, это за частую артефакты сборки и файлы, генерируемые машиной из исходных файлов в вашем репозитории, либо файлы, которые по какой-либо иной причине не должны попадать в коммиты. И для того, чтобы гит игнорировал эти файлы, они помечаются в файле .gitignore. Надеюсь, что мы пролили свет на этот загадочный пункт достаточно и можем двигаться дальше. То, за что отвечает последний пункт, останется вам как домашнее задание, потому что это не является тем, что обязательно к просмотру на данном этапе. Мы разобрали все пункты и теперб можем создавать новый репозиторий, нажав на зеленую кнопку справа снизу.

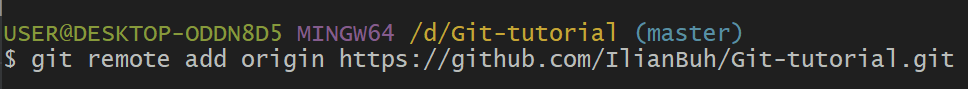


В новом окне мы можем добавить тех, кто сможет так же изменять репозиторий, нажав на кнопку справа сверху “Invite collaborators”. Еще одна из самых важных деталей, это ссылки на наш репозиторий, с помощью которых мы сможем скопировать его на наш компьютер.

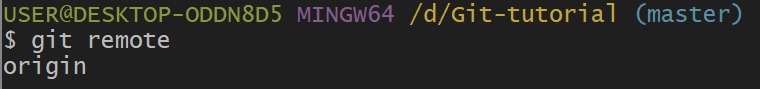
Итак, создадим теперь новый репозиторий у нас на компьютере через команду “git init”.



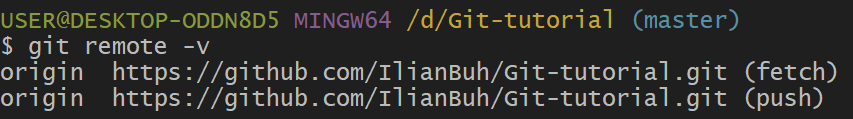
Для того чтобы работать с глобальным репозиторием нам надо, как мы уже упоминали, клонировать его на наш компьютер. Это делается за счет команды “git remote add origin “URL of the repo””(origin это название удаленного репозитория для нашего локального, оно может быть любым, но для большего удобство обычно предпочитают такой выбор



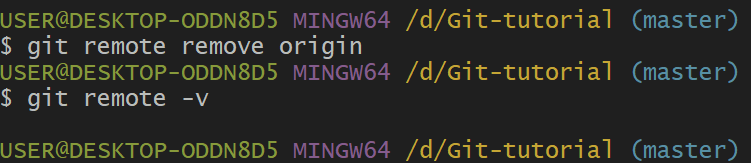
Итак, гит нам ничего не написал. Давайте посмотрим, получилось ли у нас соединить наши репозитории. Воспользуемся командой “git remote”



Так же для просмотра информации об удаленном репозитории используется атрибут -v после команды remote



Если вам понадобится отключить репозиторий, то воспользуйтесь командой “git remote remove Name”. В нашем случае Name принимает значение origin.

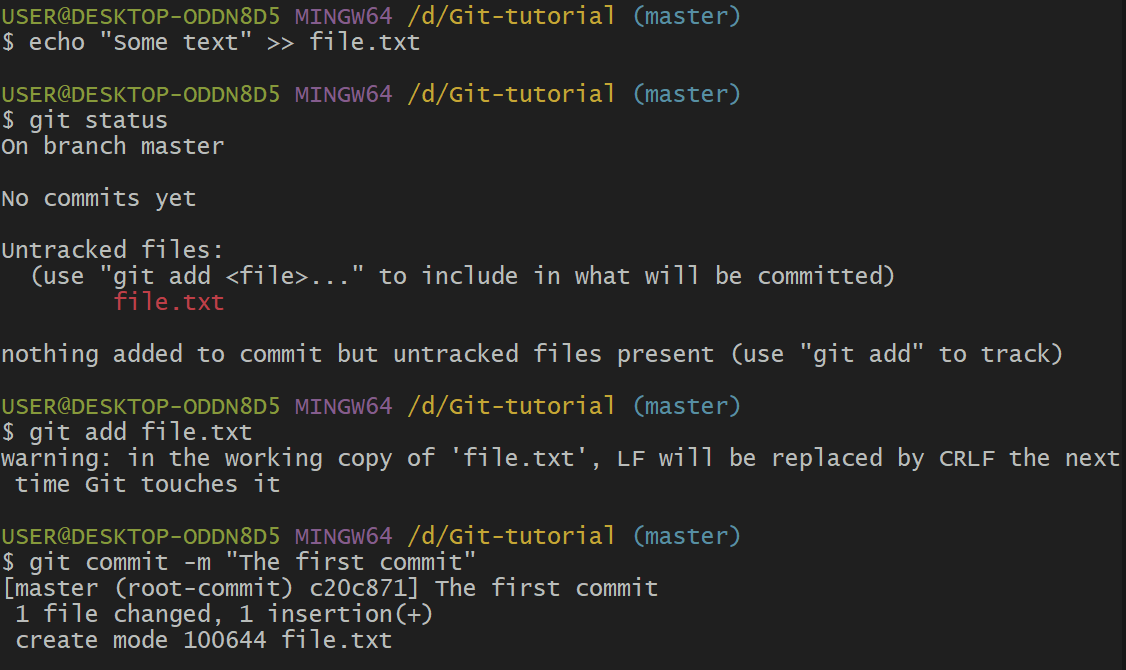


Нам высвечивается указанное нами имя, а значит все получилось.

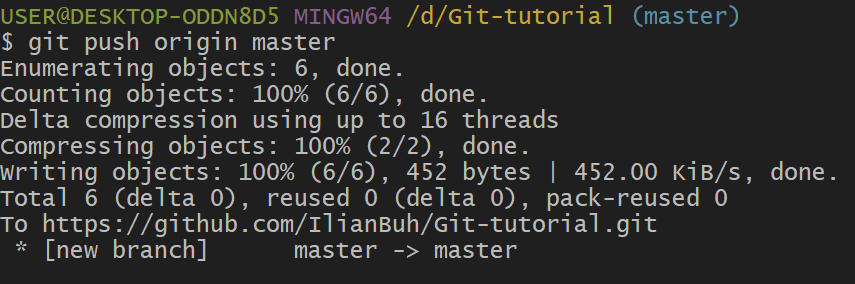
После клонирования репозитория работа с ним никак не меняется, так как теперь у вас по сути появляется дополнительное локальное хранилище. Однако, как же синхронизировать изменения, которые мы внесли, с репозиторием на GitHub?

Это можно сделать с помощью команд “git push” для загрузки с локального репозиторий в удаленный и “git pull” для обратного процесса.

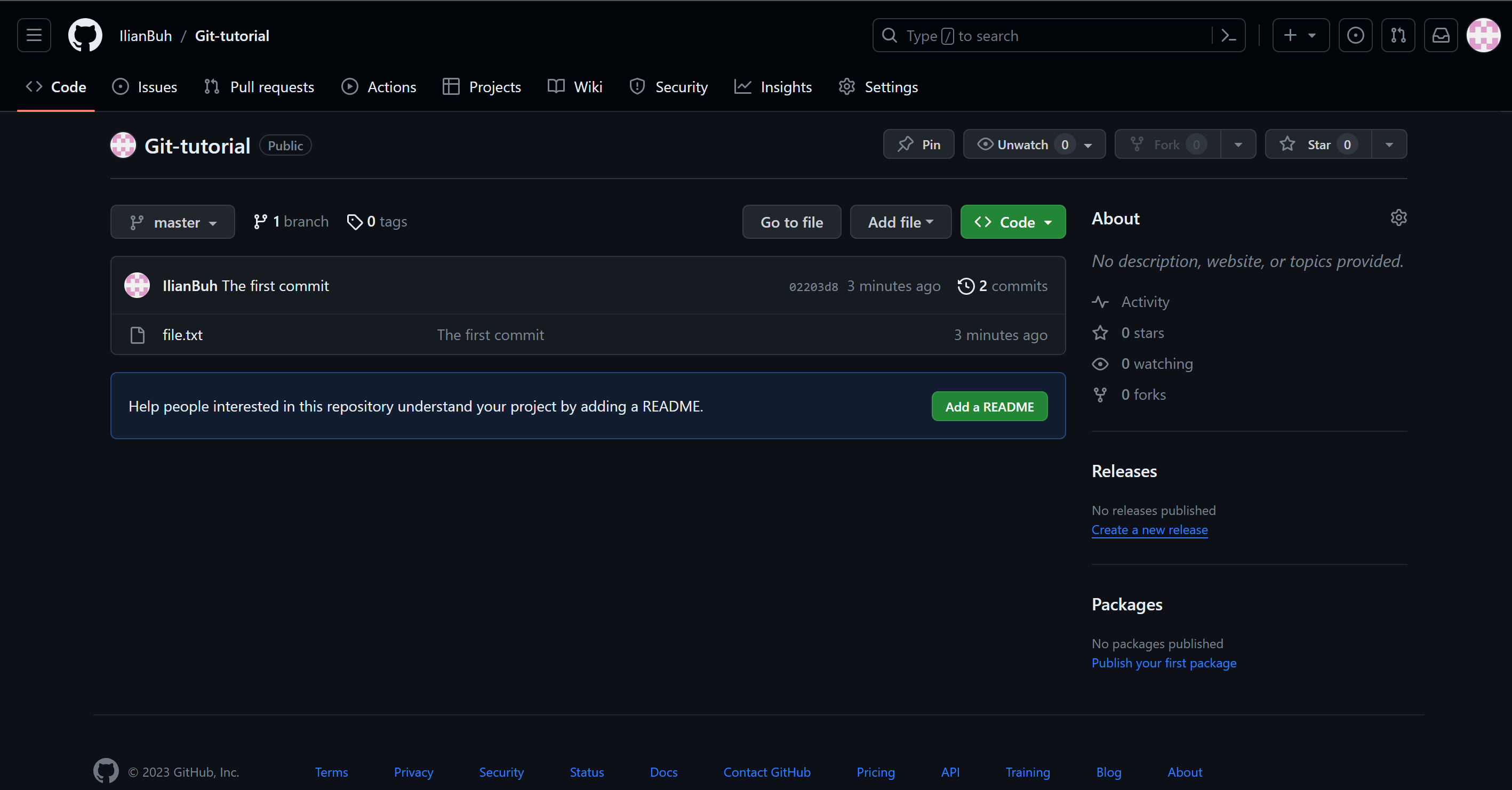
Давайте добавим некоторые изменения в наш репозиторий и попробуем отправить их на гитхаб.



Воспользуемся командой “git push”. Только помимо самой команды нам так же надо указать сначала название репозитория, на который мы хотим отправить наши коммиты(origin в нашем случае), а затем имя ветки, в которой мы их делали.

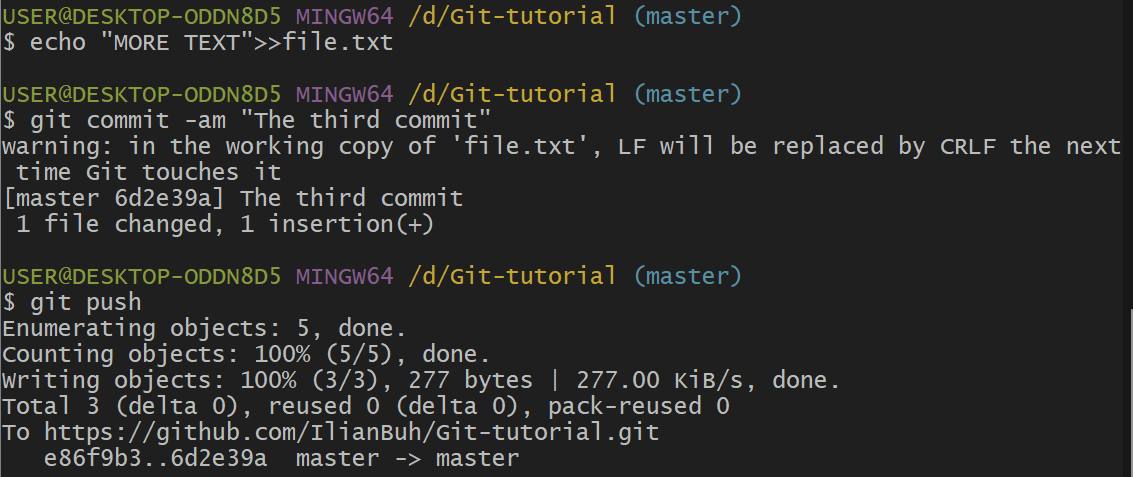
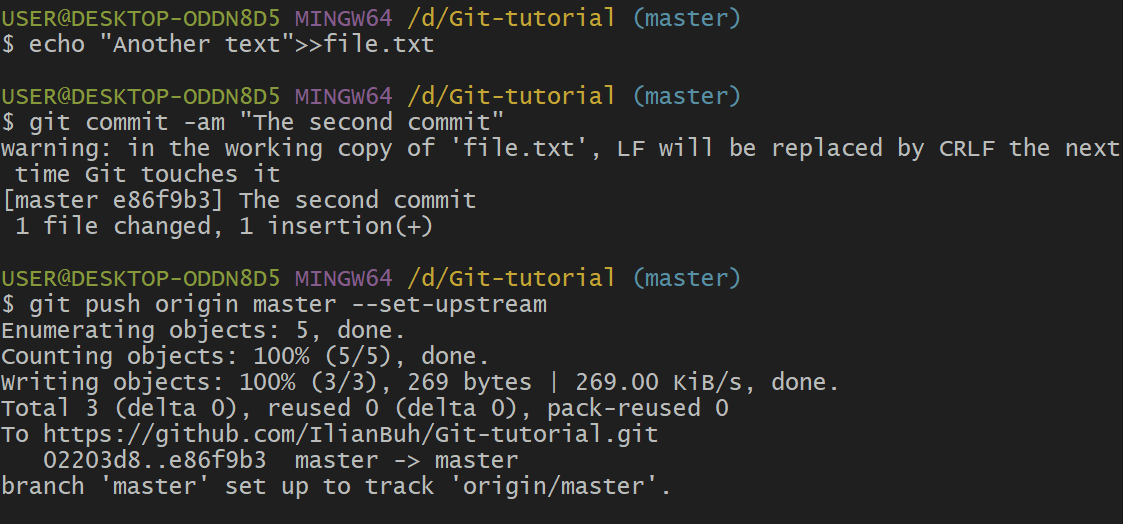


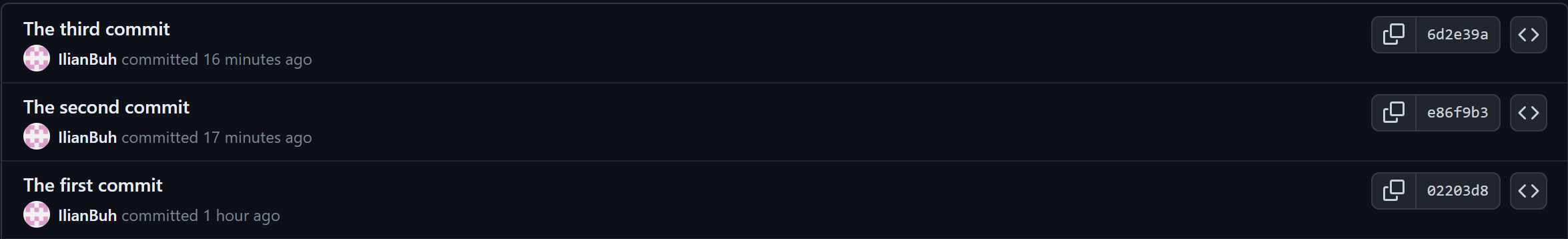
Давайте теперь посмотрим, как выглядит наш репозиторий на гитхабе



У нас добавился все содержимое рабочего каталога в локальном репозитории и все новые коммиты.

Если вам надо часто прописывать одну и ту же пару названий репозиторий-ветка(origin master в нашем случае), то вы можете прописать «git push origin master --set-upstream» и если дальше вы будете писать просто «git push», то будет подразумеваться полная форма«git push origin master».





Однако если у нас в локальном хранилище мы сделали разветвление и хотим отправить определенную ветку, то мы должны дополнить предыдущую команду до “git push -u RemoteBranch OurBranch”. Таким образом мы говорим Git загрузить нашу ветку с названием “OurBranch” (и фиксаторы внутри нее) в ветку “RemoteBranch” в удаленном репозитории(мы можем использовать “origin” вместо , в таком случае в удаленном хранилище изменения будут вноситься в главную там ветку); с атрибутом -u, мы устанавливаем связь между удаленными и локальными коммитами.

Для обратного процесса, т.е. для загрузки новых коммитов из репозитория применяется команда “git pull RemoteBranch OurBranch ”, где названия означают то же самое, что и до этого, только теперь у нас будут записываться коммиты из удаленной ветки “RemoteBranch” в локальную “OurBranch”(вместо можно использовать название “origin”, это будет означать дефолтную ветку на удаленном репозитории).

На самом деле здесь имеется еще куча подводных камней и это все невозможно объяснить вот так наспех. Поэтому мы посоветуем вам ознакомиться с дальнейшим материалом по работе с Git и GitHub уже самостоятельно, а мы на этом закончим наше выступление.

Если будем чёт делать завтра или в пн, то нужно наделать скринов.

**Заключение**

Подытоживая хочется упомянуть, что Git является очень мощным инструментом в арсенале профессиональных разработчиков программных продуктов. Поэтому всем, кто хочет стать высококвалифицированным специалистом чуть ли необязательно требуется изучить работу с Gitом. Желаем напастись терпения для преодоления этого длинного пути и удачи в покорении этого зверя. Спасибо за внимание!