**Шитиков Егор(3-2)**

**ЛАБОРАТОРНАЯ РАБОТА №1**

**1) Конфигурация –** настройка имени пользователя и электронной почты.

Проведем настройку:

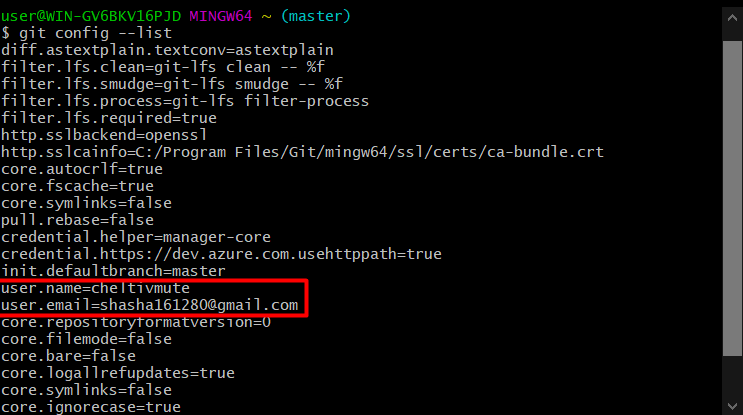
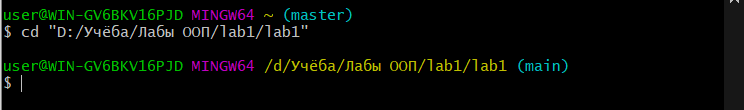


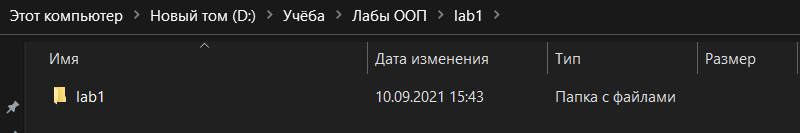
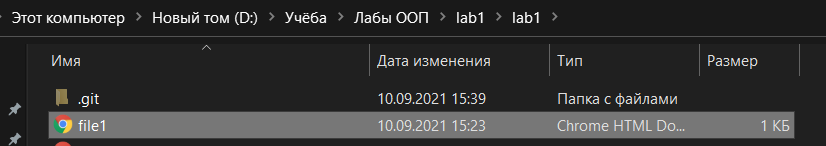
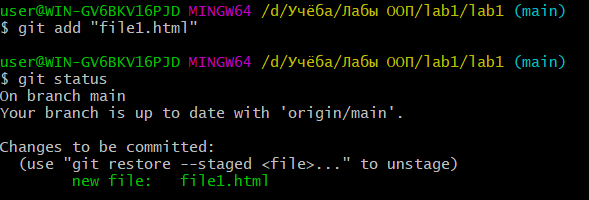
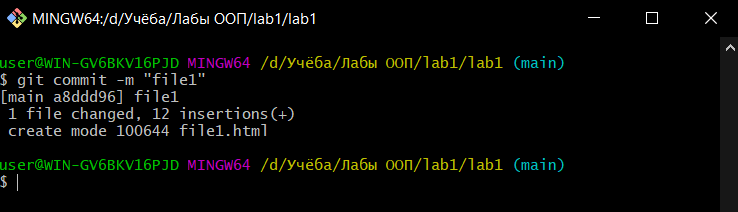
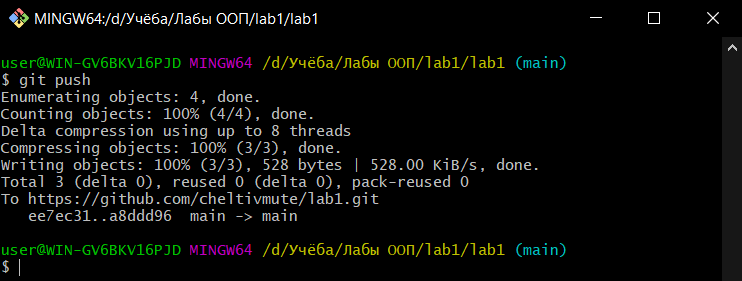


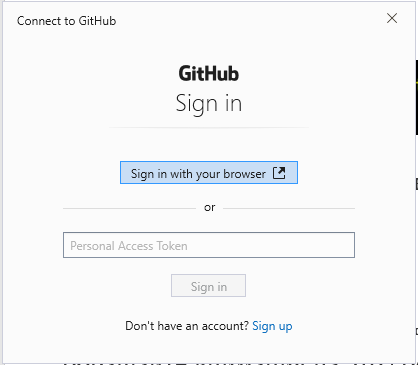
В состав Git входит утилита **git** **config**, которая позволяет просматривать и *настраивать параметры*, контролирующие все аспекты работы Git, а также его внешний вид.  Если указана **опция** **--global**, то эти настройки достаточно сделать **только один раз**, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов нужно указать другое имя или электронную почту, можно выполнить эту же команду без параметра --global в каталоге с нужным проектом.

**Ключи**

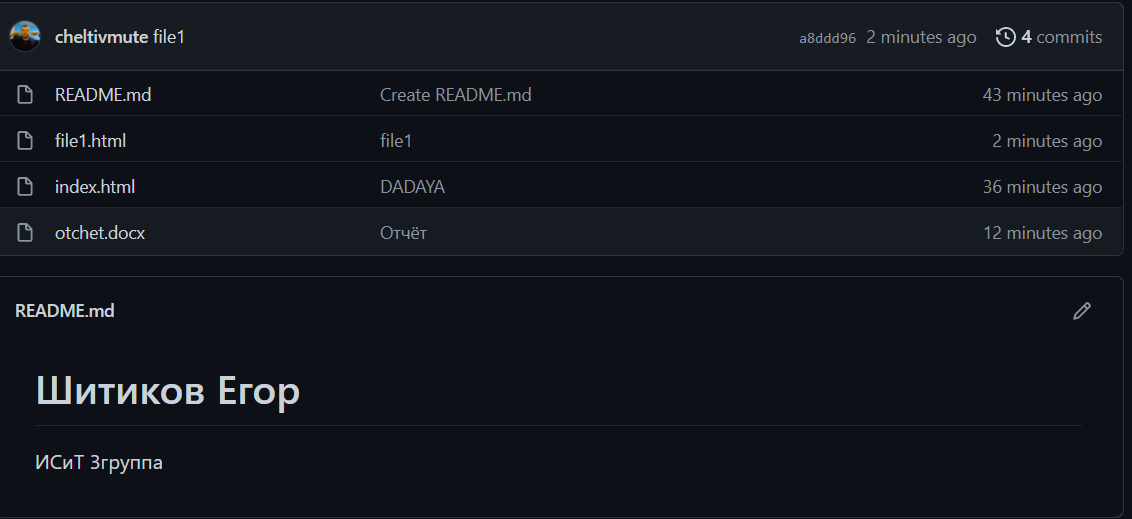
**--global**  
Изменение настроек на уровне пользователя. Без указания данного ключа настройки будут изменены только на уровне текущего репозитория.  
  
**--system**  
Изменение настроек на уровне системы (то есть сразу для всех пользователей).

Выполнение команды **git config** вместе с **опцией --list** отобразит список всех конфигураций, видимых в данный момент (в текущем каталоге).  
  
  
  
2) **Клонирование репозитория** (место хранения данных) **с** **GitHub и создание *локального* репозитория с помощью git init**Перейдем в конкретный каталог с помощью команды **cd**.  
Клонируемрепозиторий:  
git clone <ссылка на репозиторий> <название папки>То есть мы просим Git создать копию репозитория, который находится по ссылке (<ссылка на репозиторий>), и можем указать название новой папки, в которую Git скопирует репозиторий (<название папки>). Если его не указать, папка будет называться так же, как и сам репозиторий.

Создадимновый репозиторий с помощью команды **git** **init**. Эта команда создаёт в ***текущем каталоге*** новый подкаталог с именем .git, содержащий все необходимые файлы репозитория — структуру Git репозитория, **служебные файлы**.   
  
Загрузим наш проект на GitHub *из локального репозитория*.  
  
Для начала вручную скопируем его в папку локального репозитория:  
  
  
Сделали текущим каталог нашего репозитория и выполняем команду **git add .** – таким образом мы подготовим наш проект к коммиту (**проиндексируем**).   
  
  
  
Сделаем коммит:  
  
  
  
С помощью команды **git push** отправим наш проект на GitHub.  
  


Итак, в следующем окне выбираем первую опцию – авторизация через браузер:  
  
  
  
Ждем некоторое время, параллельно на сайте в нашем репозитории видим, что появился проект!  
  
Изображение выглядит как текст, снимок экрана, монитор

Автоматически созданное описание



**3) Добавление в репозитория нового файла. Информация о статусе файлов**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.

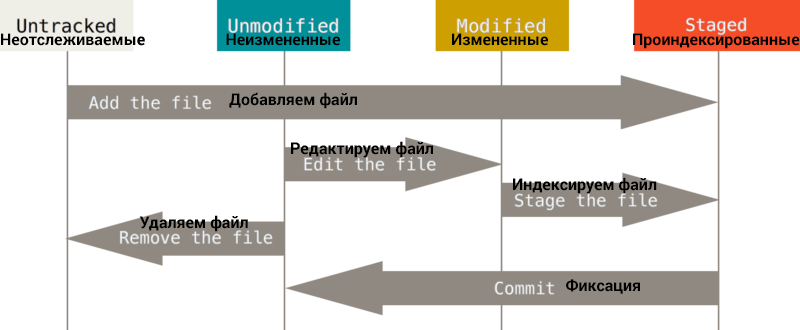
Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. **Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения,** а затем цикл повторяется.

**Файлы в GIT могут находиться в 3 состояниях:**

* Модифицированные (**modified**) - изменения есть, но в локальную БД не внесены.
* Индексированные (**staged**) - модифицированные файлы, при этом помечены как готовые для фиксации.
* Зафиксированные (**commited**) - данные сохранены в локальной БД.

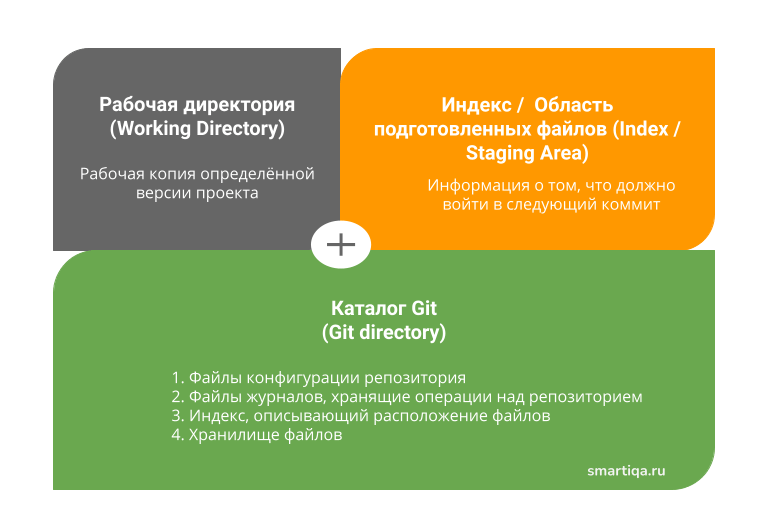
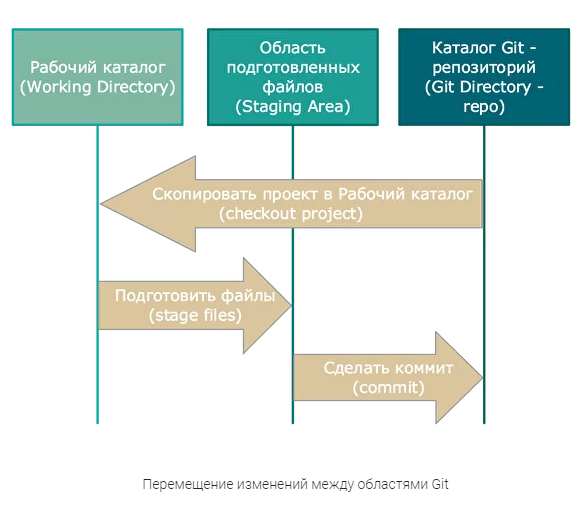
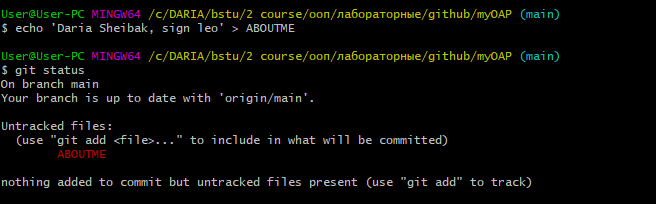
**Как строится процесс работы в Git**

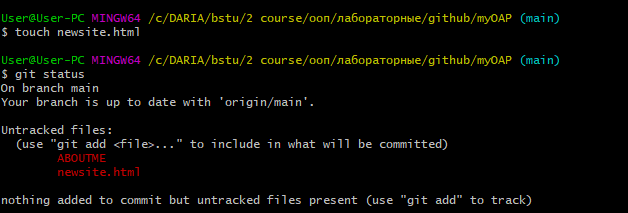
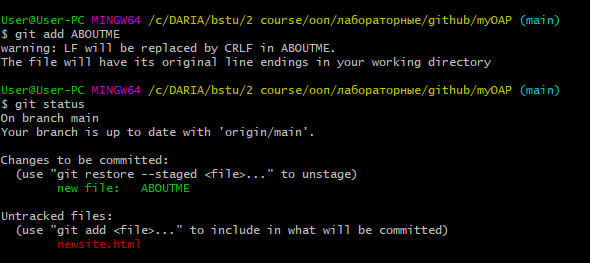
* 1. Редактируем файлы (модификация)
* 2. Индексируем файлы, при это снимки файлов добавляются в область индексирования (git add)
* 3. Фиксация. Из области индексирования файлы сохраняются в папке Git (git commit)



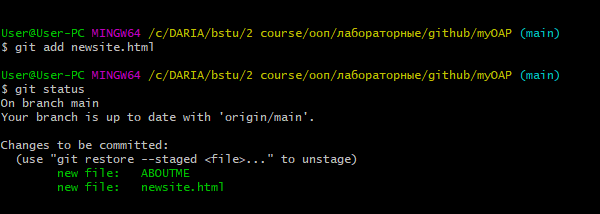
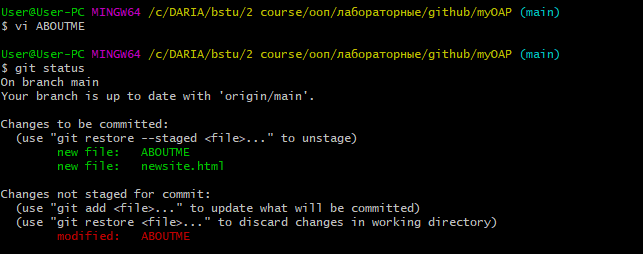
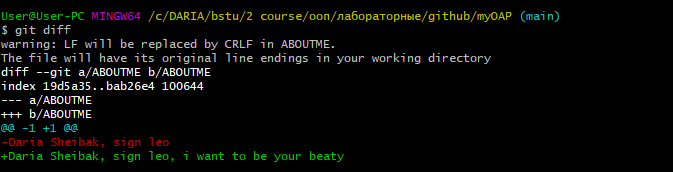
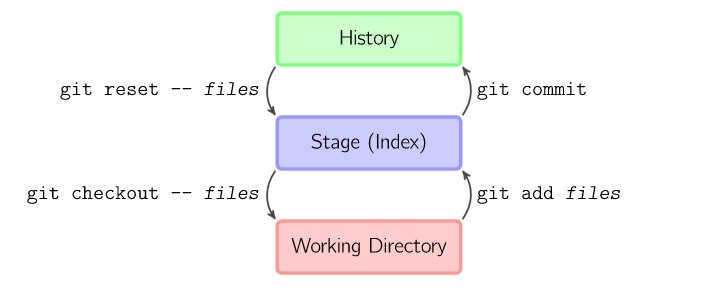
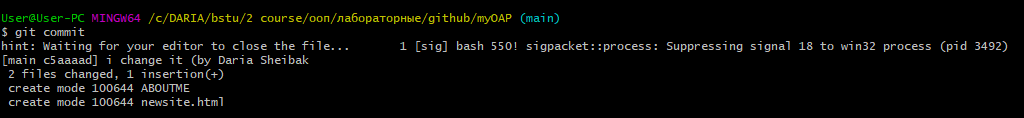
***ВАЖНО!* Репозиторий Git условно можно разделить на три составляющие:**

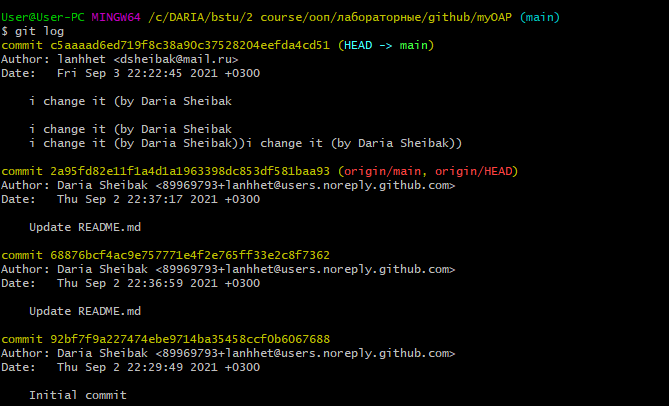
1. **Рабочая директория - Working directory**. Это файловая структура, с которой непосредственно работает пользователь в конкретный момент времени. Технически же - это копия определенной версии вашего проекта, которую вы извлекли из базы Git и в которую пытаетесь внести свои изменения.
2. **Индекс или Область подготовленных файлов - Index / Staging area**. Это область, где хранятся имена файлов и изменения в них, которые должны войти в следующий коммит. Технически индекс - это просто файл.
3. **Директория Git - Git Directory**. Папка, в которой Git хранит все версии вашего проекта и также свои служебные файлы. Данная папка носит название **.git**и располагается в корневой директории вашего проекта.

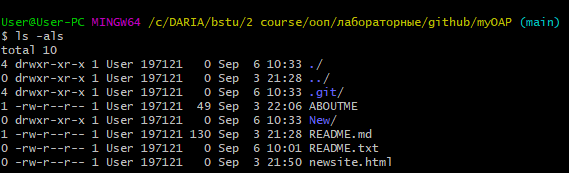
  
  
  
Таким образом, получается, что ваши файлы путешествуют между этими тремя областями. Файлы, с которыми вы напрямую работаете - это Working Directory. Что-то изменили в этих файлах - изменилось состояние Working Directory.  
  
Хотите зафиксировать эти изменения - скажите Git, какие именно из всех изменений, вы хотите сохранить. Для этого вы добавляете изменения в файлах во вторую область - Staging (он же Index). Это некое среднее состояние между Working Directory и Git Directory - изменения уже на пути к фиксации, но еще не сохранены в базе Git.  
  
Если вы уверены, что все изменения, которые вы добавили в Index / Staging, необходимо сохранить в базу Git, то вы делаете коммит, и они в сжатом виде помещаются в Git Directory. Теперь все надежно сохранено в папке .git.  
  
  
  
  
  
Добавим в репозиторий myOAP новый файл ABOUTME, который содержит в себе текст «Daria Sheibak, sign leo», с помощью команды **echo** и символа **перенаправления ввода >.**   
  
Либо создадим файл с помощью команды **touch**, которая в качестве параметра принимает имя файла.

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда **git status**.  Команда **git status** отображает **состояние рабочего каталога** и раздела проиндексированных файлов. С ее помощью можно проверить индексацию изменений и ***увидеть файлы, которые не отслеживаются Git.***   
  
Понять, что новые файлы **неотслеживаемые** можно по тому, что они 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 коммиты, пока мы его явно об этом не попросим**. Это предохранит от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять.  
  
Для того чтобы начать **отслеживать** (добавить под версионный контроль) новый файл, используется команда**git add.**Можно видеть, что теперь файл ABOUTME **проиндексирован**, так как он находится в секции «**Changes to be committed**». Если мы выполним коммит в этот момент, то версия файла, существовавшая на момент выполнения команды **git add**, будет добавлена в историю снимков состояния.

*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».*

Проиндексируем следующий файл:  
  
  
  
Теперь эти файлы **под версионным контролем**.  
  
  
Если мы модифицируем файл **ABOUTME** с помощью редактора vi (**команда vi**), то последующий **git status** об этом нам сообщит (напротив файла написано **modified**- "изменено"):  
  
  
  
  
Если результат работы команды **git status** недостаточно информативен для вас — вам **хочется знать, что конкретно поменялось, а не только какие файлы были изменены** — вы можете использовать команду **git diff**.   
  
  
  
**Git commit** - **это команда для записи индексированных изменений в репозиторий Git.**  
  
Прежде чем создавать очередной коммит, необходимо проиндексировать файлы в рабочей области с помощью команды [**git add**](https://ru.stackoverflow.com/questions/tagged/git-add). Новый коммит будет включать текущие состояния индексированных файлов плюс последние сохраненные состояния неиндексированных (но отслеживаемых) файлов. Обратите внимание: коммит включает в себя не изменения относительно предыдущего коммита, а **"снимок" (англ. shapshot)** текущего состояния рабочей области.  
  
  
  
  
  
**Коммиты** — **основные конструктивные элементы временной шкалы проекта Git**. **Их можно рассматривать как снимки состояния или контрольные точки на временной шкале проекта Git**. Коммиты создаются с помощью команды git commit, которая делает снимок состояния проекта на текущий момент времени. Коммиты снимков состояния Git всегда выполняются в локальный репозиторий.

С помощью команды **git log** мы можем просмотреть историю коммитов:  
  
  
Помимо **автора и даты**, у каждого коммита **есть идентификатор, который называется hash.** Одним из самых полезных аргументов является **-p или --patch,** который показывает разницу, внесенную в каждый коммит.   
  
Для просмотра всех файлов и каталогов в текущей папке используем команду **ls -als.**



**4) Настройте gitignore файл для вашего с# репозитория.** Перейдите по  
ссылке https://github.com/github/gitignore и изучите структуру нужного файла.

Изображение выглядит как текст

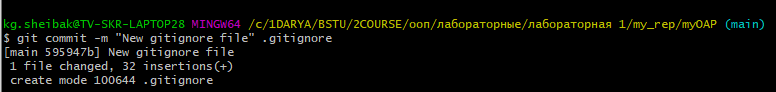
Автоматически созданное описание

Создадим вручную файл .gitignore и заполним его согласно информации по ссылке для С++ проекта:

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание  
  
После индексации изменений сделаем коммит:  
  
  
  
Отправим наш файл на GitHub с помощью команды **git push**:

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание  
  
Изображение выглядит как текст

Автоматически созданное описание

**5) Изучите команды удаления и перемещения файлов.**

**Удаление файла.**

Удалить файл из репозитория можно двумя способами: **первый** – удалить его из рабочей директории и уведомить об этом git; **второй** – воспользоваться средствами git.

**Первый способ** – удалить вручную файл из репозитория или командой ОС (rm – Windows), затем выполнить git add и после git commit.

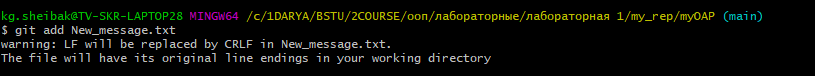
**Второй способ** — это сразу использовать команду git rm без предварительного удаления файла из директории.  
  
  
  
Изображение выглядит как текст

Автоматически созданное описание

*Поясним:*  
Создали новый файл в нашем репозитории, затем добавили его под версионный контроль (git add), то есть проиндексировали; после мы зафиксировали индекс (git commit), а далее – удалили с помощью команды git rm. Не стоит забывать также сделать коммит после удаления файла, чтобы сделать изменения постоянными.

**Перемещение файла.**

Формат команды:  
  
**git mv <Имя файла> <Директория/Имя файла>**

Создадим новый файл для дальнейшего перемещения:  
  
  
  
  
Создадим для него папку в рамках нашего репозитория:  
  
  
  
Проиндексируем наш новый файл (git add):  
  
  
  
Перемещаем:  
  
  
  
Результат (перемещение - это еще и переименование):  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
**5) Изучите команду git log – просмотр истории коммитов.**

Команда **git log** выводит в обратном хронологическом порядке список сохраненных в данный репозиторий версий. То есть первыми показываются самые свежие коммиты. Как видите, рядом с каждым коммитом указывается его контрольная сумма SHA-1, имя и электронная почта автора, дата создания и сообщение о  
фиксации.

Изображение выглядит как текст

Автоматически созданное описание

**По умолчанию (без аргументов) git log перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке — последние коммиты находятся вверху**. Из примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 контрольными суммами, именем и электронной почтой автора, датой создания и сообщением коммита.

**Команда git log имеет очень большое количество опций для поиска коммитов по разным критериям.**

Одним из самых полезных аргументов является -p или --patch, который показывает разницу (выводит патч), внесенную в каждый коммит. Так же вы можете ограничить количество записей в выводе команды; используйте параметр -2 для вывода только двух записей

6) **Сделайте отмену внесенных в файл изменений.**

**git checkout -- [file] — опасная команда**. Любые изменения соответствующего файла пропадают — вы просто копируете поверх него другой файл. Ни в коем случае не используйте эту  
команду, если вы не уверены, что файл вам не нужен. Все, что зафиксировано коммитом в Git, почти всегда можно восстановить. Но все, что вы потеряете, не сделав коммит, скорее всего, вам больше не увидеть.  
  
Изображение выглядит как текст

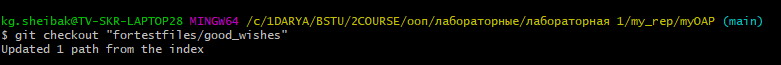
Автоматически созданное описание

Изменим файл и сохраним его:  
  
Изображение выглядит как текст

Автоматически созданное описание

Изменения не проиндексированы:  
  
Изображение выглядит как текст, снимок экрана, экран

Автоматически созданное описание

**Чтобы отменить все незакоммиченные изменения:**

Посмотрим результат (**восстановился файл в состоянии на момент последнего коммита**):  
  
Изображение выглядит как текст

Автоматически созданное описание

**6) Изучите команду git revert**

Что делать, если коммит уже сделан, но он нас по каким-то причинам не устраивает? Ситуаций может быть много:

* Забыли добавить в коммит нужные файлы
* Изменения нужно «откатить», чтобы доработать
* Изменения больше не актуальны, и их нужно удалить
* Изменения были сделаны по ошибке, и их нужно отменить

**Git revert - Откатить состояние репозитория к указанному коммиту**

Самая простая ситуация — отмена изменений. Фактически она сводится к созданию ещё одного коммита, который выполняет изменения противоположные тому коммиту, который отменяется. Таким образом откат будет виден в истории как очередной коммит.

Команда *git* revert "отменяет" не только последний коммит, но и любой другой коммит из истории проекта.

Команда **git rever**t – безопасный способ отменить операцию без потери истории коммитов. Команда отменяет действия прошлых [коммитов](https://proglib.io/p/git-commit), создавая новый, содержащий все отменённые изменения.

Создадим новый файл:  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
Проверим его статус и проиндексируем, а после сделаем коммит:  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
Предположим, что это изменение в репозитории больше неактуально. Вызовем команду git log, возьмем идентификатор последнего коммита и выполним относительно него команду **git revert**:

После этой команды откроется редактор, ожидающий ввода описания коммита. Обычно сообщение revert не меняют, поэтому достаточно просто закрыть редактор:

Изображение выглядит как текст

Автоматически созданное описание

Наш проект вернулся к тому состоянию, в котором был до выполненного коммита (так можно делать с любым другим коммитом из истории проекта), а «откат» теперь виден в истории как очередной коммит:

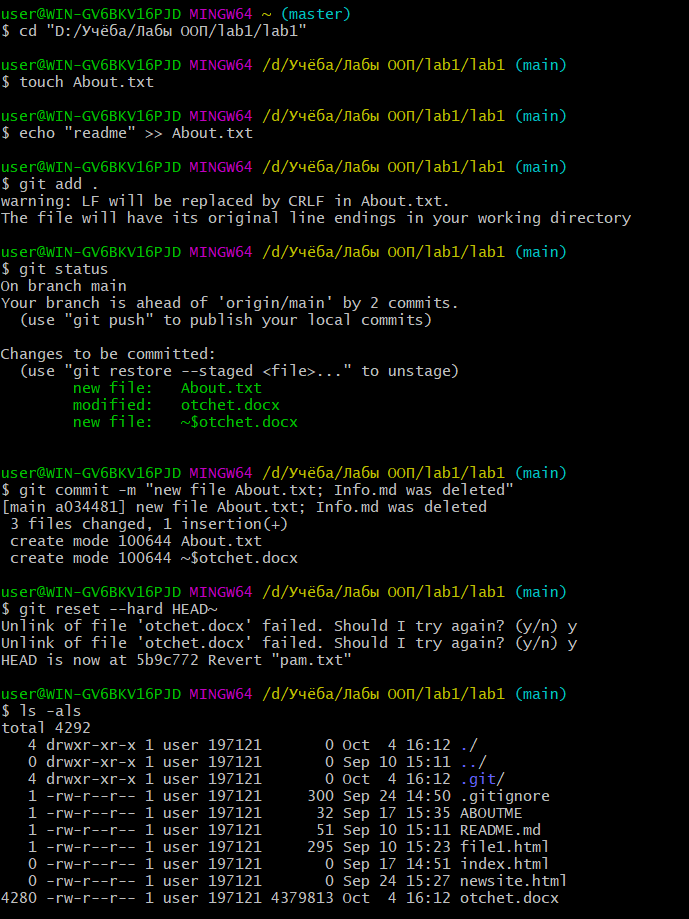
Изображение выглядит как текст

Автоматически созданное описание

**7) Изучите команду git reset**

Иногда удалить нужно только что сделанный по ошибке коммит. Конечно, и в этом случае подходит git revert, но так загрязняется история. Если этот коммит сделан был только сейчас и ещё не отправлялся на Github, то **лучше сделать так, как будто бы этого коммита не существовало в принципе.**

Git позволяет удалять коммиты. Это **опасная операция**, которую нужно делать только в том случае, если речь идет про новые коммиты, которых нет ни у кого, кроме вас.

Вручную удалим какой-нибудь файл из репозитория, в командной строке создадим новый и проиндексируем эти изменения. Выполним коммит:   
  
  
  
После этого вызовем команду:  
  
**git reset --hard HEAD~**

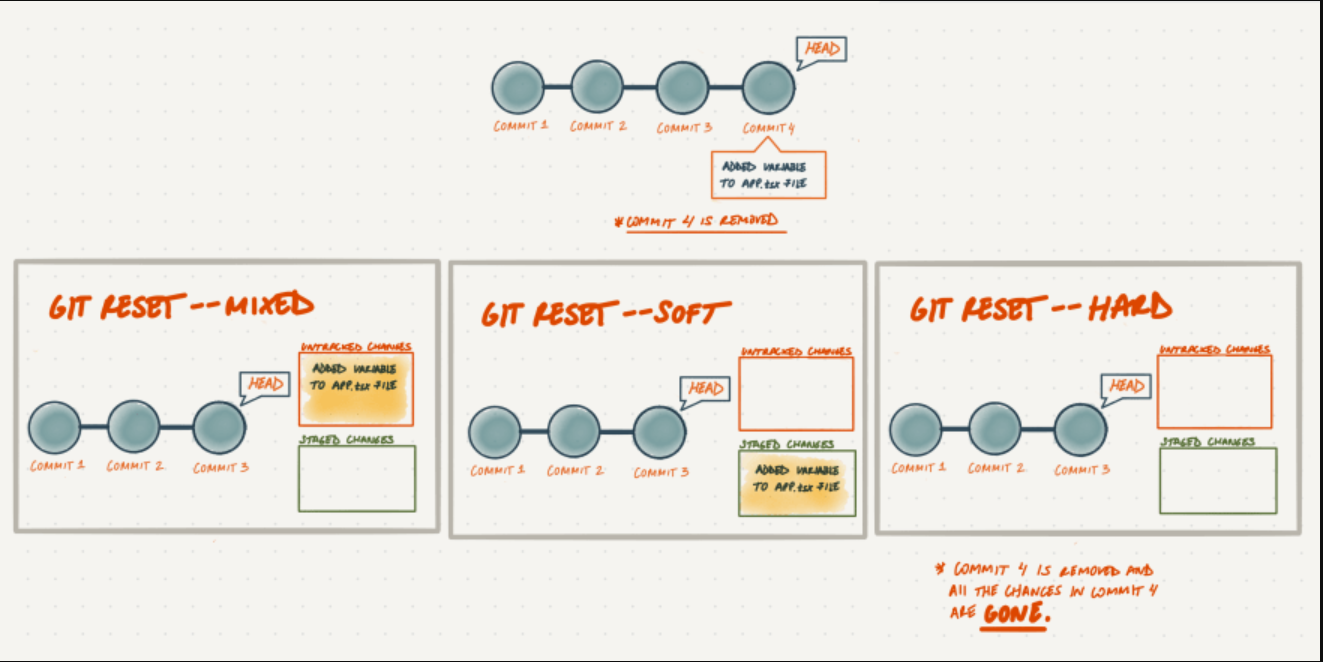
*HEAD (голова) — так обозначается последний сделанный коммит.*

**HEAD~ означает "один коммит от последнего коммита". Если бы мы хотели удалить два последних коммита, то могли бы написать HEAD~2.**

Последний коммит был удален и история о нем тоже.

- *В результате вернется наш удаленный файл и удалится только что созданный*.

Для этой команды существует 3 опции:  
**--mixed** (по умолчанию, в таком варианте *reset* отправляет изменения последнего коммита в рабочую директорию. Затем их можно исправить или отменить и выполнить новый коммит)  
**--soft** (HEAD переключается на последний коммит, однако, изменения, добавленные после этой фиксации, остаются с пометкой staged)  
**--hard** (полное удаление)



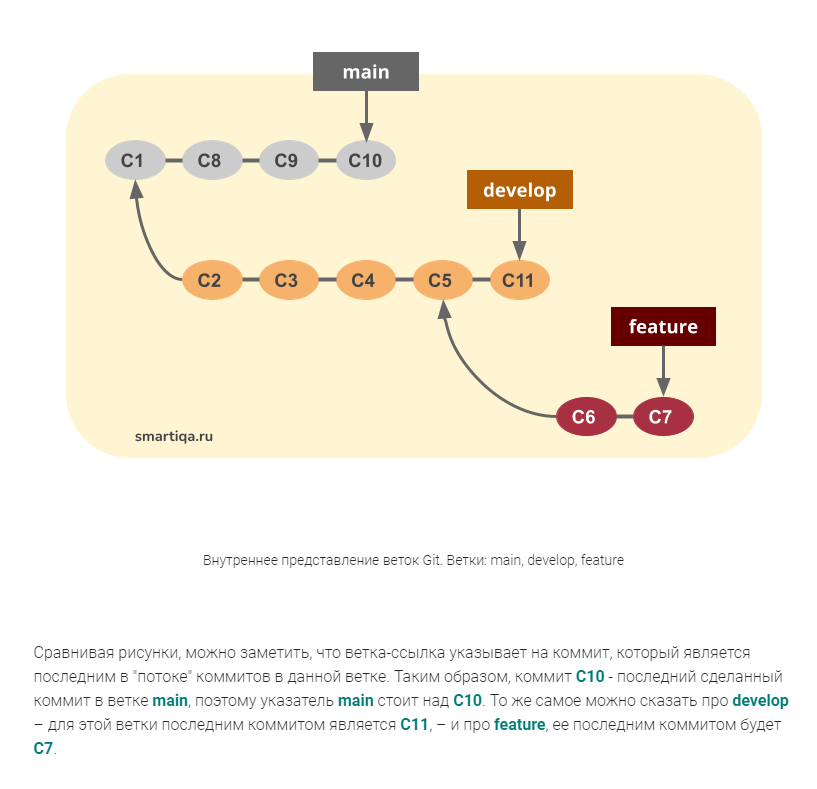
**8) Работа с ветками**

**Создайте (branch) новую ветку, например, test и переключитесь (checkout) в нее.**  
  
*Используя ветвление, вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию.*

*С точки зрения логики, ветка – это последовательность коммитов.*

Изображение выглядит как текст

Автоматически созданное описание  
  
*На физическом уровне, то есть с точки зрения внутренней реализации Git, ветка – это ссылка на последний коммит в этой ветке.   
  
Преимущество веток в их независимости. Вы можете вносить изменения в файлы в одной ветке, например, пробовать новую функцию, и они никак не скажутся на файлах в другой ветке. Каждый раз, когда мы создаем новый коммит, Git автоматически перемещает указатель****main****на последний коммит.*

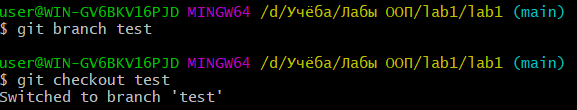
  
  
Итак, чаще всего ветки используются **в следующих случаях.**

1. Ветки нужны, чтобы несколько программистов могли **вести работу** над одним и тем же проектом или даже файлом **одновременно**, при этом не мешая друг другу.
2. Кроме того, ветки используются **для тестирования экспериментальных функций**: чтобы не повредить основному проекту, создается новая ветка специально для экспериментов.
3. Помимо прочего, ветки можно использовать **для**разных выходящих **параллельно релизов** одного проекта.

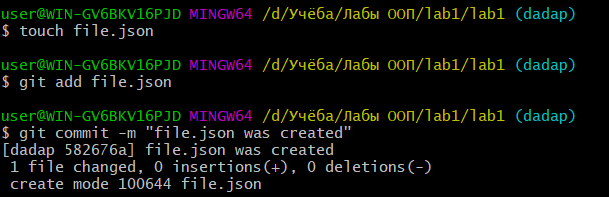
**Способ создания ветки git branch + git chekcout**

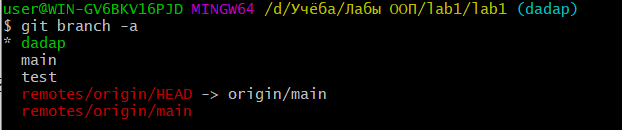
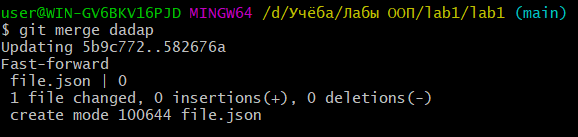
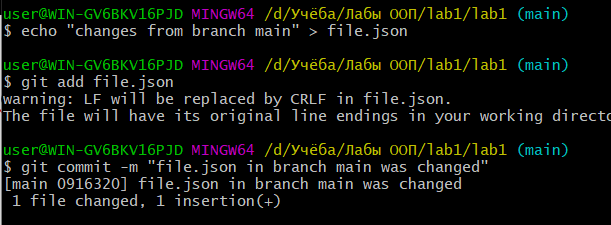
git branch <имя ветки>

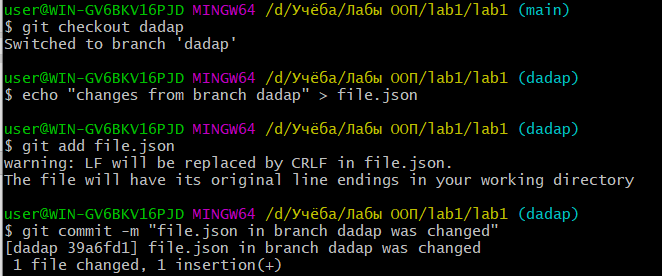
git checkout <имя ветки>

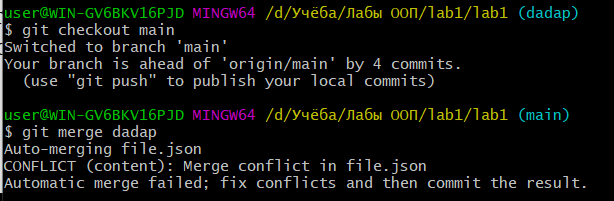
Создадим новую ветку с именем test и переключимся на нее:  
  
  
  
Или так. Мы создали ветку dadap и сразу на нее переключились:

git checkout -b <имя ветки>

  
  
  
Добавим в ветку dadap новый файл file.json, проиндексируем его и сделаем коммит:  
  
  
  
  
Когда вы работаете с большим количеством веток, можно легко забыть имя нужной, а без имени ветки переключится на нее не получится. Для таких ситуаций существует команда просмотра списка веток. На самом деле, это уже знакомая нам git branch, но с другими ключами.  
  
git branch <ключи>  
  
**-a значит все ветки  
-r значит удаленные ветки**

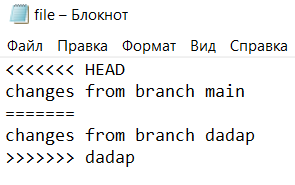
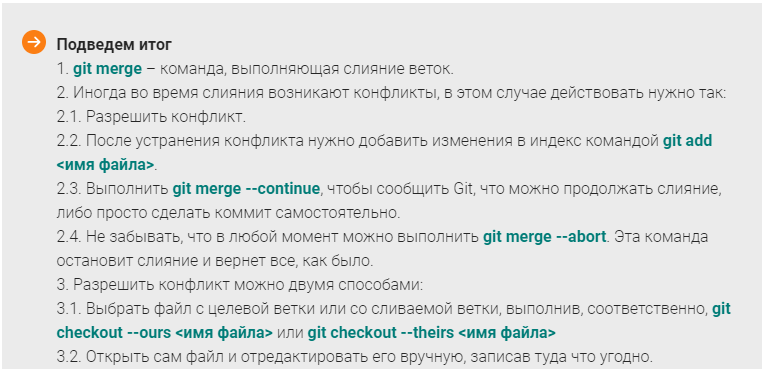
  
  
В примере выше можно увидеть, что перед веткой **dadap**стоит звездочка. Такая запись означает, что сейчас указатель **HEAD**находится на ветке **dadap**.  
  
**Выполните слияние веток.**  
Общий **ход**нашей **работы**выглядит следующим образом:  
1. *Решили добавить новую функцию – создали отдельную ветку*. Дальше работаем в новой ветке.  
2. Написали функцию, протестировали ее работу, внесли все необходимые исправления, еще раз протестировали и *убедились, что функция работает исправно* и не привнесла ошибок в остальной код.  
3. Теперь нужно как-то перенести изменения с тестовой ветки на основную. Тут нам на помощь и приходит **слияние**: мы просто **сливаем (т.е. переносим) изменения с нашей тестовой ветки в основную.**  
**Сливаемая ветка** – та ветка, с которой мы берем изменения, чтобы влить их в целевую. **Целевая ветка** – та ветка, в которую мы сливаем наши изменения.  
  
**Слияние веток** – это перенос изменений с одной ветки на другую. При этом слияние не затрагивает сливаемую ветку, то есть она остается в том же состоянии, что позволяет нам потом продолжить работу с ней.  
  
  
  
  
  
  
Формат команды:  
  
git merge <сливаемая ветка>  
**Поочереди отредактируйте файл file.json в ветках main и dadap.**  
Изменения в ветке main:  


Изменения в ветке dadap:  
  


Очень часто во время слияния веток оказывается, что ваши изменения удаляют или переписывают информацию в уже существующих файлах. Такая ситуация называется файловым конфликтом. Git останавливает выполнение слияния, пока вы не разрешите конфликт.  
  
  
Сделаем слияние веток:   
  
  
  
*По сути, Git сталкивается с проблемой*: у него есть два файла с одним и тем же именем, и он не знает, какой из них взять. Поэтому обращается к нам за помощью.

Разрешим ситуацию конфликта слияния:  
  
Общий подход к **разрешению конфликтов** такой:

1. Непосредственно разрешить конфликт одним из двух рассмотренных немного ниже способов. Либо, если возникновение конфликта стало неожиданным для вас, можно выполнить **git merge --abort**. Эта команда прервет слияние и вернет все, как было.
2. Сообщить Git, что мы разрешили конфликт, добавив все файлы с разрешенными конфликтами в индекс. Сделать это можно уже знакомой командой **git add <конфликтный файл>** для каждого конфликтного файла.
3. Продолжить слияние, выполнив **git merge --continue**. Либо вручную создать **merge-коммит** уже знакомой командой **git commit**.

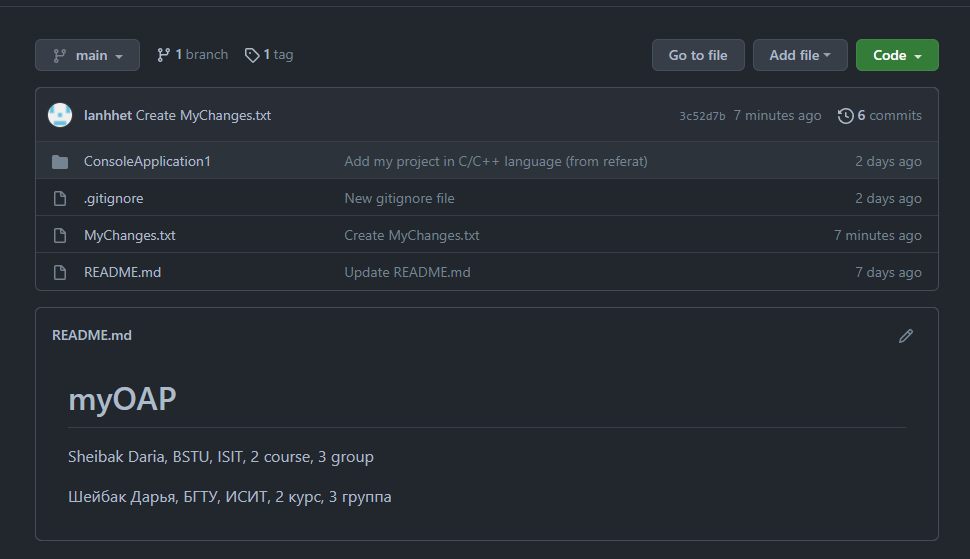
  
  
  
  
  
  
  
  
  
**Удаление ветки:**

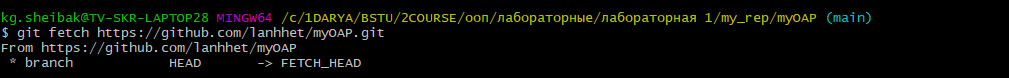
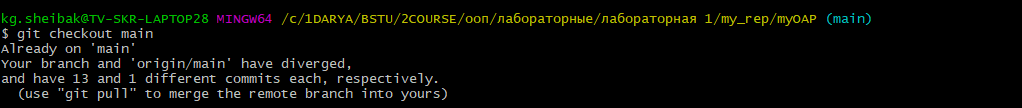
**git branch -d branch\_name**

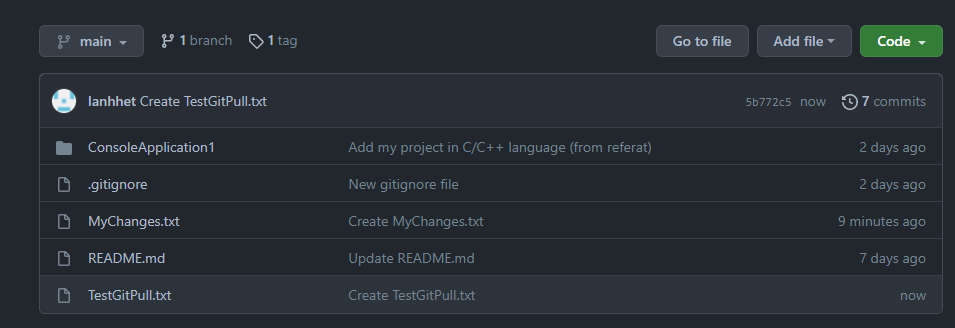
**git branch -D branch\_name**

Параметр -d означает --delete, который удалит локальную ветвь, только в случае, если вы смерджили её с какой-то из веток.

Опция -D обозначает --delete --force, которая удаляет ветку независимо от ее статуса push или merge, так что будьте осторожны при её использовании

**9) Работа с удаленными репозиториями (расположенными удаленно)**Команду загрузки **git push** см выше.  
  
Команда **git push** используется для установления связи с удалённым репозиторием, вычисления локальных изменений отсутствующих в нём, и собственно их передачи в вышеупомянутый репозиторий. Команда **git push чаще всего используется для публикации выгружаемых локальных изменений в центральном репозитории.** Для того чтобы **поделиться изменениями, внесенными в локальный репозиторий**, с удаленными участниками команды, необходимо выполнить команду push.  
  
**Выполним команды выгрузки git pull и git fetch.**1. Команда **git fetch** связывается с удалённым репозиторием и забирает из него все изменения, которых у вас пока нет и сохраняет их локально. Она сливает все данные с проекта, которые находятся в нашем *удаленном репозитории*. Все данные, которых у нас нет, она сливает, **но не применяет в наши ветки.**  


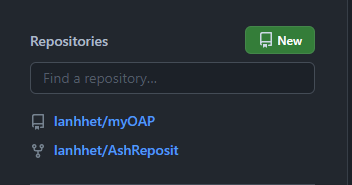
  
  
  
  
Проверим ветку (git checkout <имя ветки>):  
  
  
  
Выполним git merge, чтобы залить в ветку то, что мы взяли с помощью git fetch с удаленного репозитория (GitHub):  
  
Изображение выглядит как текст, внутренний, снимок экрана, экран

Автоматически созданное описание  
  
2. Команда **git pull** работает как комбинация команд **git fetch и git merge**, т. е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой.  
  
  
  
Изображение выглядит как текст

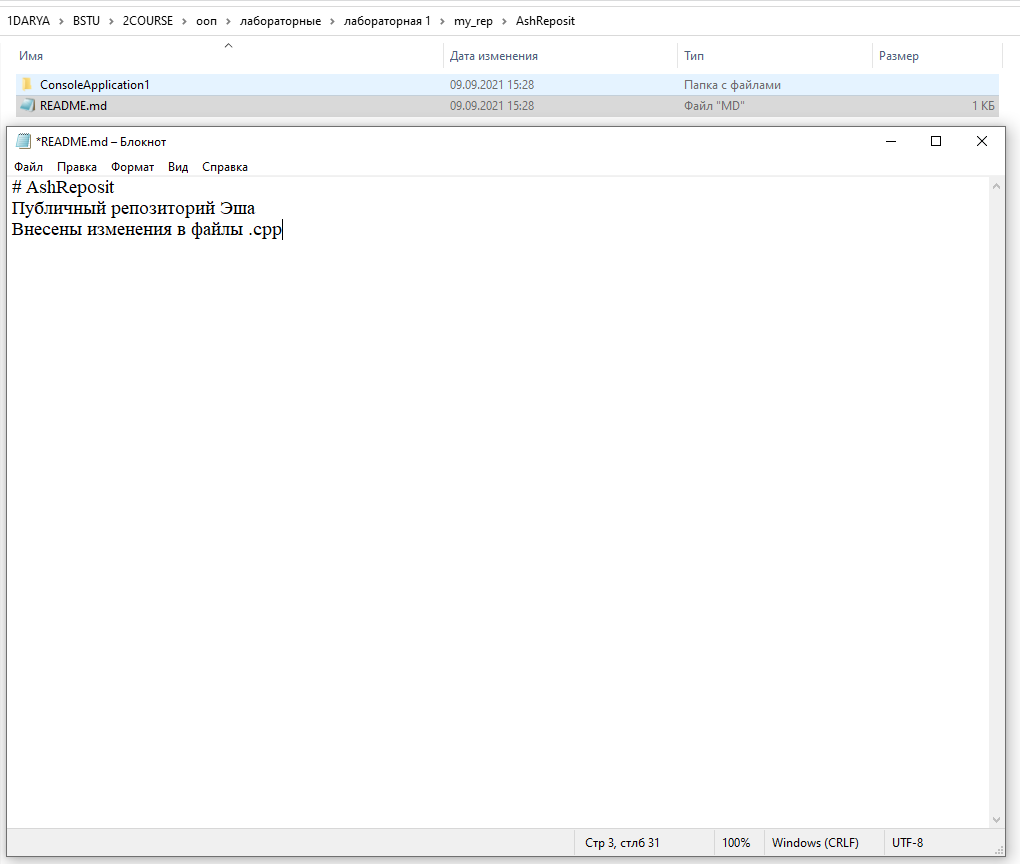
Автоматически созданное описание

**10) Cовместная работа**

1. У вас есть репозиторий, в который будут приходить pull request и фиксироваться владельцем репозитория. Найдите репозиторий коллеги, над которым вы хотели бы вести совместную работу. Найдите кнопку Fork и нажмите ее. После этого создается  
копия этого репозитория, но уже в вашем аккаунте.  
  
Изображение выглядит как текст, электроника

Автоматически созданное описание  
  
  
  
**Pull Request** — это запрос на вливание изменений из вашей ветки в основную ветку исходного репозитория.   
  
**Пул реквест** (pull request) -- это *запрос* к управляющему каким-либо репозиторием (человеку, группе людей или вообще роботу) *на применение изменений* (из вашего репозитория и/или указанной вами ветки).  
  
  
2. И теперь этот уже свой репозиторий клонируем себе на локальную  
машину. Внесите изменения в файл. Сделайте commit.  
  
Клонируем этот репозиторий себе локально:  
  
Изображение выглядит как текст

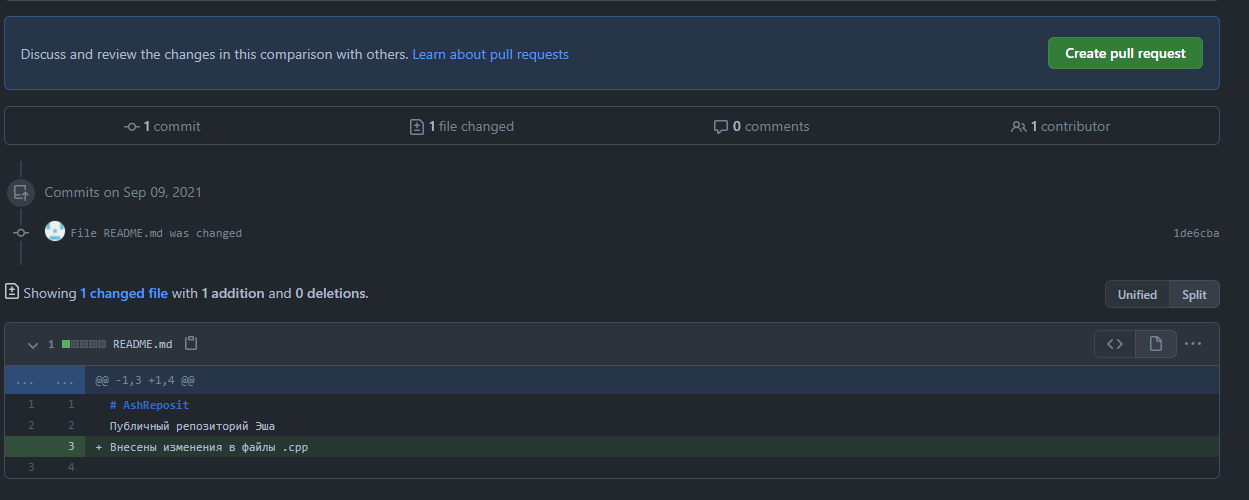
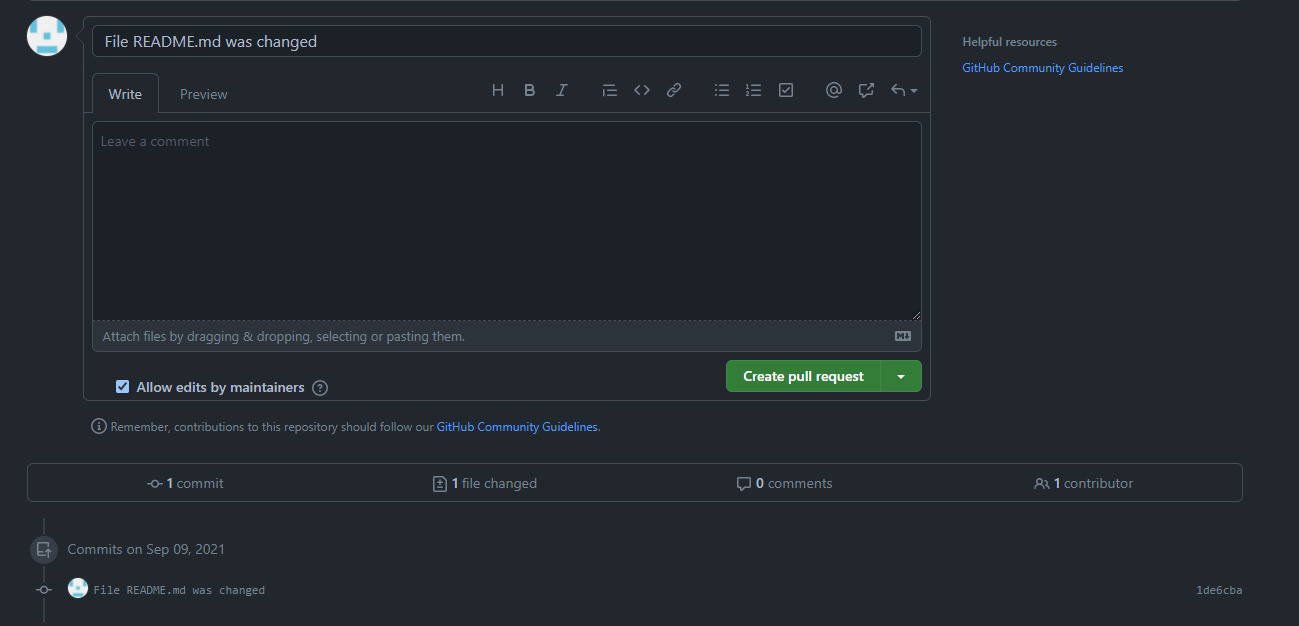
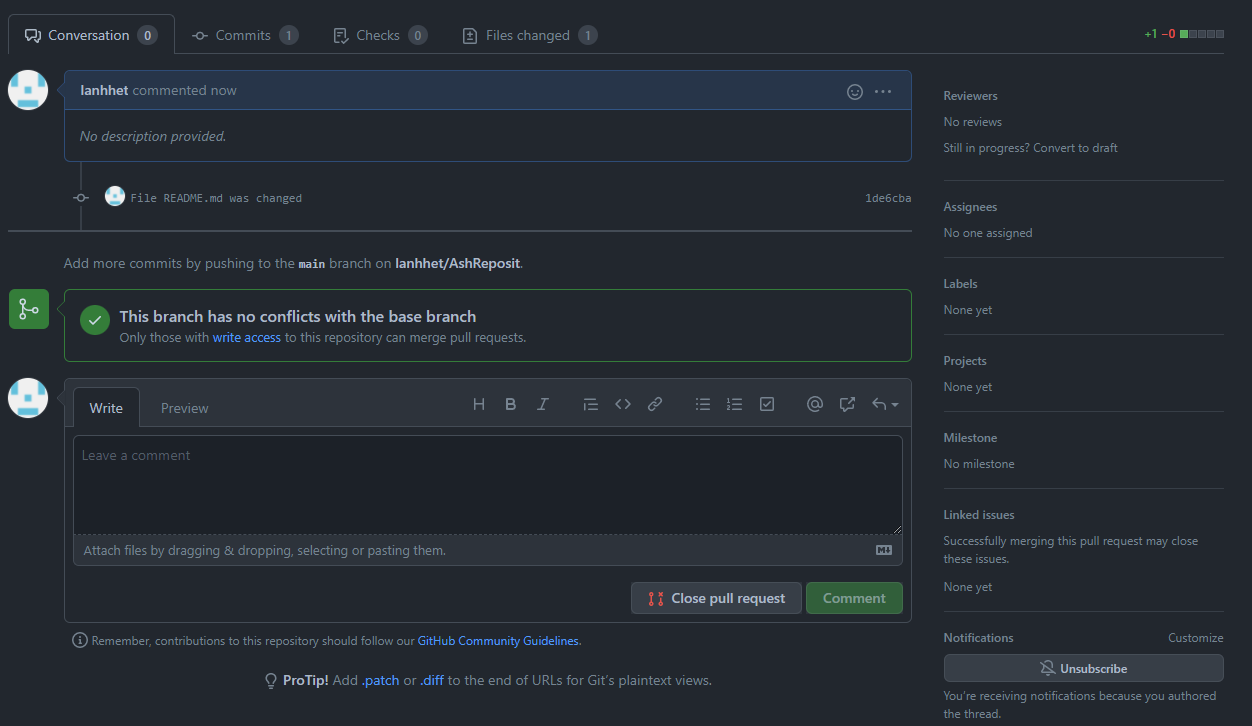
Автоматически созданное описание  
Изображение выглядит как текст

Автоматически созданное описание  
  
Внесем изменения вручную в файл данного репозитория:  
  
  
  
Проиндексируем и сделаем коммит:  
  
Изображение выглядит как текст

Автоматически созданное описание  
3. Отправьте изменения в свой удаленный облачный репозиторий.  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
Изображение выглядит как текст, снимок экрана, монитор, внутренний

Автоматически созданное описание  
  
4. Создаем Pull requests через интерфейс Github.   
  
Для этого есть кнопка, которая так и называется. Нажимаем кнопку **Create pull request**. После этого переходим на страницу репозитория, с которым мы хотим объединиться. Видно, кто и какие изменения предлагается внести. Ваш коллега должен зафиксировать изменения.  
  
Изображение выглядит как текст, снимок экрана, монитор, экран

Автоматически созданное описание  
  
  
  


У коллеги видим следующее:  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
Одобрим:  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
  
  
Изображение выглядит как текст

Автоматически созданное описание

5. Теперь владелец репозитория внесет изменения и нам тоже надо их

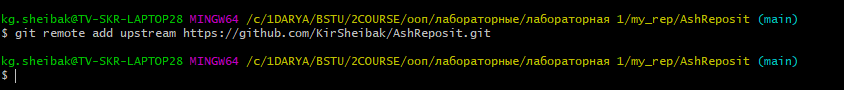
получить. Зафиксируем изменения, сделанные в совместном проекте у себя.

Коллега внес изменения – добавил файл:

Изображение выглядит как текст

Автоматически созданное описание

Выполним команду **git remote add upstream** [**<ссылка**](https://github.com/gilamran/tsc-watch.git) **на репозиторий коллеги, откуда мы будем брать изменения>** *(добавим удаленный сервер в качестве вышестоящего):*



Далеекоманда **git fetch upstream** *(извлечь изменения):*

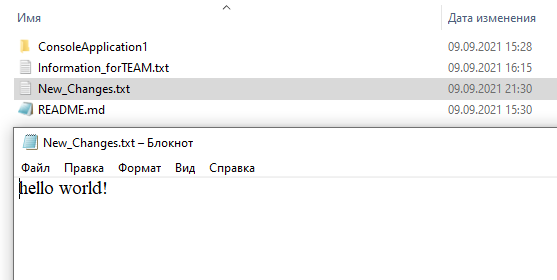
Изображение выглядит как текст

Автоматически созданное описание

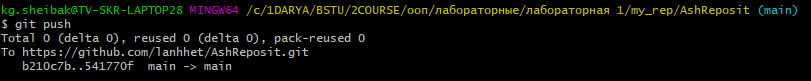
После выполняем **git merge upstream/main:**

Изображение выглядит как текст

Автоматически созданное описание

Файл появился в локальном репозитории:  
  


Теперь отправим эти изменения на GitHub в наш fork репозитория.

Выполним **git push**:  
  
  
  
  
Изображение выглядит как текст, снимок экрана, монитор, экран

Автоматически созданное описание  
  
  
**11) Интеграция со средой разработки**Используя инструменты интеграции с git выполниите предыдущие  
задания, но из среды разработки (Visual Studio 2012).

Для командной работы с Git в Visual Studio 2012 нужно установить дополнительно пакет Microsoft.TeamFoundation.Git.Provider.msi.   
  
Используем опцию VIEW –> Team Explorer:  
  
Изображение выглядит как текст

Автоматически созданное описание

Зададим настройки Git:  
  
Изображение выглядит как текст

Автоматически созданное описание  
  
Внесем изменения в любой из файлов проекта:  
  
Изображение выглядит как текст

Автоматически созданное описание

Сделаем коммит изменений:  
  
Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**12) Изучите команды**

**git remote**

Команда git remote позволяет создавать, просматривать и удалять подключения к другим репозиториям. По сути, команда git remote — это интерфейс для управления списком записей об удаленных подключениях, которые хранятся в файле /.git/config репозитория.   
  
Просмотрим список удаленных подключений к другим репозиториям:

**1. git remote -v**  
Изображение выглядит как текст

Автоматически созданное описание  
  
Команда git remote также предоставляет удобный способ изменения файла /.git/config репозитория. Перечисленные ниже команды позволяют управлять подключениями к другим репозиториям и изменять файл /.git/config репозитория. 

**2. git remote add <name> <url>**   
  
- Создание нового подключения к удаленному репозиторию. После добавления удаленного репозитория имя ＜name＞ можно использовать в качестве удобного ярлыка для адреса ＜url＞ в других командах Git.  
  
**3. git remote rm <name>**

- Удаление подключения к удаленному репозиторию с именем ＜name＞

**4. git remote rename <old-name> <new-name>**

- Переименование удаленного подключения с имени ＜old-name＞ на ＜new-name＞.

**git stash**

Команда **git stash** позволяет на время «сдать в архив» **(или *отложить*) изменения, сделанные в рабочей копии, чтобы вы могли применить их позже**. Откладывание изменений полезно, если вам необходимо переключить контекст и вы пока не готовы к созданию коммита.

Команда **git stash save** выполняет то же самое, что и git stash, но имеет несколько полезных опций. Например, можно сохранить изменения и добавить сообщение — подписать изменения, чтобы потом вспомнить, что именно было спрятано. В качестве сообщения, например, можно написать о том, какие именно изменения выполнены в файлах.

Формат:

**git stash save "Какое-нибудь сообщение"**

Команда **git stash list** выводит список всех ваших прятаний.

Формат:

**git stash list**

Команда git stash apply берет самое свежее прятанье (stash@{0}) и применяет его к репозиторию. То есть изменения, которые находятся в этом прятанье, применяются к текущему репозиторию. Это похоже на то, как вы применяете патч, только в качестве патча выступает ваше прятанье.

Формат:

**git stash apply**

Команда git stash pop выполняет все тоже самое, что и команда git stash apply, но удаляет прятанье, которое она применяет к репозиторию.

Формат:

**git stash pop**

**git rebase**

Вместо слияния можно выполнить перебазирование функциональной ветки на главную ветку main с помощью этой команды.

В результате вся функциональная ветка окажется поверх главной ветки main, включая в себя все новые коммиты в ветке main. Если вместо команды merge при коммитах используется rebase, эта команда *перезаписывает* историю проекта, создавая новые коммиты для каждого коммита в исходной ветке.

**Главное преимущество rebase — более чистая история проекта.**  
Например:

Вся история ветки test перебазируется в историю ветки main. И сами данные тоже.  
  
Изображение выглядит как текст

Автоматически созданное описание

**git tag – пометка коммита в истории**

**Теги — это ссылки, указывающие на определенные точки в истории Git.** Команда git tag обычно используется для захвата некой точки в истории, которая используется для релиза нумерованной версии.  
  
Для создания нового тега выполните следующую команду:

**git tag <tagname>**

Просмотреть список имеющихся тегов в Git можно очень просто. Достаточно набрать команду git tag (параметры -l и --list опциональны).  
  
  
  
Данная команда перечисляет теги в алфавитном порядке; порядок их отображения не имеет существенного значения.

Тег используется для обозначения и пометки конкретного **коммита** в истории. Обычно он используется для обозначения точек выпуска.  
  
Изображение выглядит как текст

Автоматически созданное описание

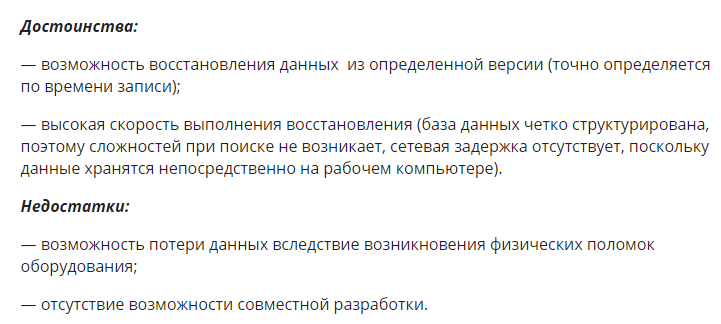
Теги бывают **аннотированные и облегченные.**

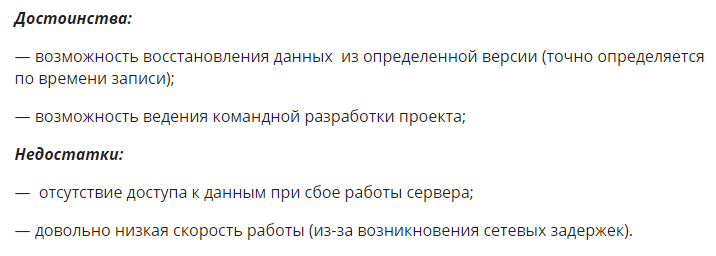
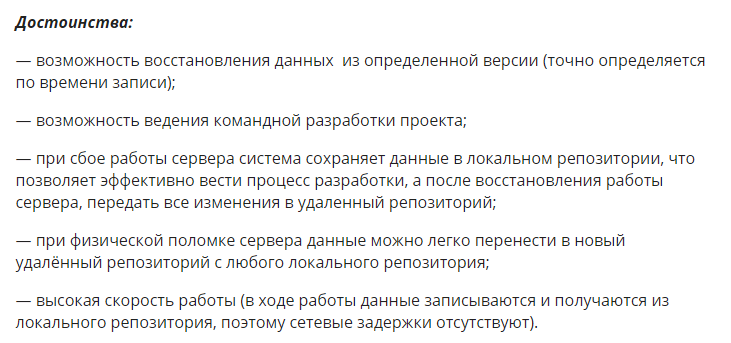
**Контрольные вопросы.**1. Что такое система контроля версий, для чего ее используют?  
  
**Система контроля версий** — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии.   
  
Она позволяет вернуть файлы к состоянию, в котором они были до изменений, **вернуть проект к исходному состоянию**, **увидеть изменения**, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу. Такие системы наиболее широко используются при разработке программного обеспечения **для   
хранения исходных версий** разрабатываемой программы.   
  
Система контроля версий — **программное обеспечение, которое обеспечивает командную работу в рамках одного или нескольких проектов**. Команда разработчиков взаимодействует с консольным или браузерным инструментом для выгрузки кода на сервер, скачивания его на рабочий компьютер и изменения структуры. **Она хранит все версии проекта и обеспечивает к ним доступ. Любой член команды может взаимодействовать с основной «веткой» проекта или создавать новые.  
  
Какие задачи решает система контроля версий:**

* 1. **Защищает исходный код от потери**. Данные хранятся на удалённом сервере, даже если разработчики удалят файлы с локального компьютера, они останутся в репозитории.
  2. **Обеспечивает командную работу**. Программисту не надо использовать инструменты для командной работы и платить за них. Каждый может работать на своём компьютере и обновлять файлы по мере необходимости.
  3. **Помогает отменить изменения**. В любой момент можно вернуться к контрольной точке, сравнить исходный код с текущим и обновить главную ветку после ревью.
  4. **Распределённая работа**. Необязательно работать с проектом «наживую». Плагин может функционировать на сайте, а программисты будут спокойно создавать новую версию.

**RCS, CVS,** **Subversion**, Aegis, Monoton, **Git**, Bazaar, Arch, Perforce, **Mercurial** – самые распространенные системы контроля версий.

**Виды СКВ:**  
1) Локальная система контроля версий

**Локальные  СКВ** обычно хранят на компьютере список изменений, внесенных в файлы. Основываясь на этих данных, система контроля версий воссоздает нужную версию файла (актуальную на определенный момент времени).  
  


2) **Централизованная система контроля версий**Централизованные  системы контроля версий предполагают сохранение версий проектов на  общий сервер, с которого потом получают нужные версии клиенты.  
  
3) **Распределенная система контроля версий**  
  
Для устранения единой точки отказа используются распределенные системы контроля версий. Они подразумевают, что клиент выкачает себе весь репозиторий целиком вместо выкачки конкретных интересующих клиента файлов. Если умрет любая копия репозитория, то это не приведет к потере кодовой базы, поскольку она может быть восстановлена с компьютера любого разработчика. Каждая копия является полным бэкапом данных.  
  
Все копии являются равноправным и могут синхронизироваться между собой.  
  


2. Какой принцип хранения файлов использует Git?  
  
**Принцип хранения** - многие VCS хранят свои данные в виде списка изменений, **Git же хранит полные копии файлов, только заменяя неизмененные файлы на ссылки.**

Таким образом **Git является своего рода небольшой файловой системой.**

Такой подход предоставляет ряд преимуществ при восстановлении данных, работе с коммитами и т.д., но требует больше места.

**Git включает в себя для каждой фиксации(коммита) полную копию всех файлов, за исключением того, что для содержимого, уже присутствующего в Git repo, моментальный снимок будет просто указывать на указанное содержимое**, а не дублировать его.  
  
  
3. В чем отличие git от других систем контроля версий?

Основное отличие Git от любой другой СКВ - **это подход к работе со своими данными**. Концептуально, большинство других систем хранят информацию в виде **списка изменений в файлах**. Эти системы (CVS, Subversion, Perforce, Bazaar и др) представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени.   
  
Git не хранит и не обрабатывает данные таким способом. Вместо этого, подход Git к хранению данных больше похож **на набор снимков миниатюрной файловой системы**. **Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок**. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков**.

**Для работы большинства операций в Git достаточно локальных файлов и ресурсов** — в основном, системе не нужна никакая информация с других компьютеров в вашей сети.

**В Git для всего вычисляется хеш-сумма**, и только потом происходит сохранение. В дальнейшем обращение к сохранённым объектам происходит по этой хеш-сумме.

4. В каких трех основных состояниях файлы могут находиться в Git ?

**Файлы в GIT могут находиться в 3 состояниях:**

* Модифицированные (**modified**) - изменения есть, но в локальную БД не внесены.
* Индексированные (**staged**) - модифицированные файлы, при этом помечены как готовые для фиксации.
* Зафиксированные (**commited**) - данные сохранены в локальной БД.

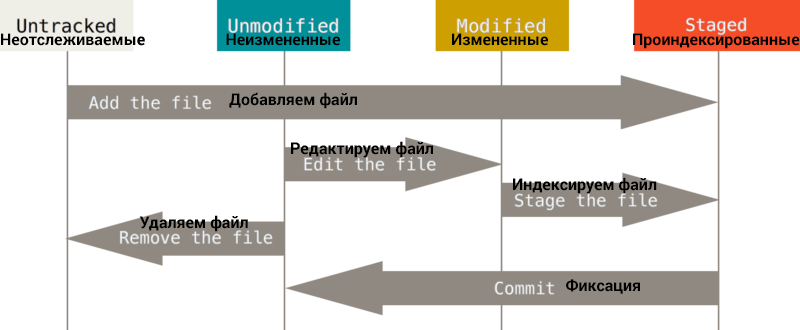
**К модифицированным (измененным) относятся файлы, которые поменялись, но ещё не были зафиксированы.**

**Индексированный** — это изменённый файл в его текущей версии, отмеченный для включения в следующий коммит.

**Зафиксированный** значит, что файл уже сохранён в вашей локальной базе.

**Как строится процесс работы в Git**

* 1. Редактируем файлы
* 2. Индексируем файлы, при это снимки файлов добавляются в область индексирования (git add)
* 3. Фиксация. Из области индексирования файлы сохраняются в папке Git (git commit)



5. Что такое индексация файла? Какой командой она выполняется?

**Чтобы проиндексировать его, необходимо выполнить команду 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».

**Индексирование** (также: indexing, staging) — это процесс добавления текущего содержимого (изменённого) файла в индекс (также: index area, staging area).

Что при этом происходит с технической точки зрения?

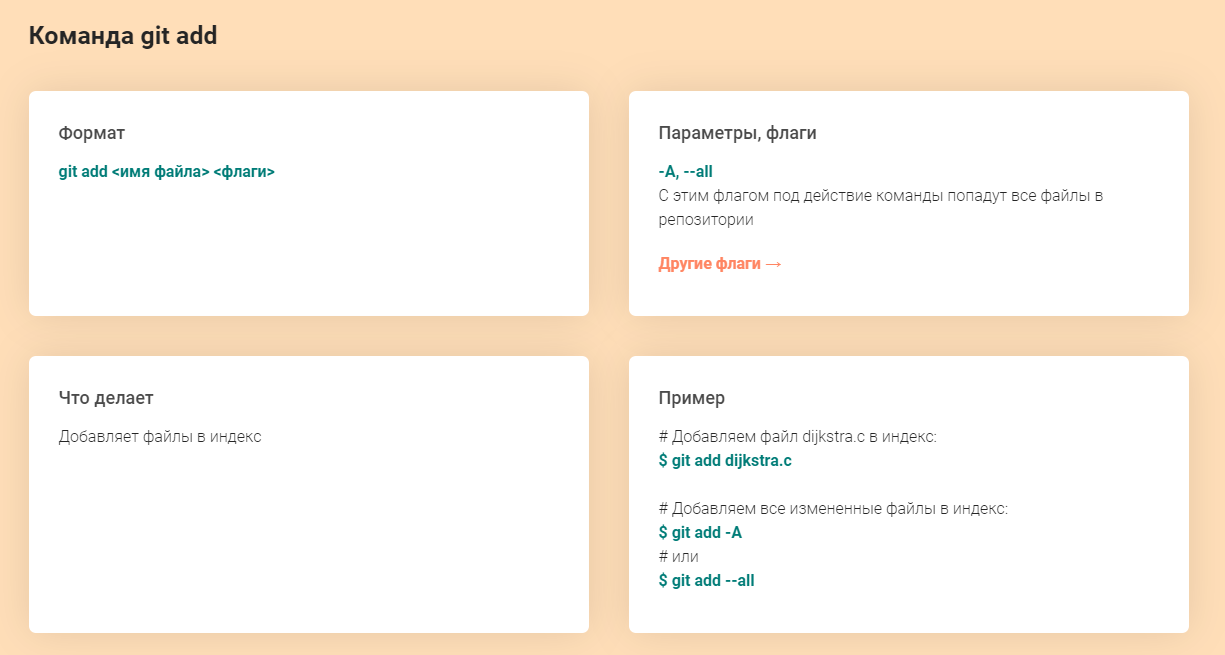
Берётся содержимое файла, впереди к нему дописывается немного служебной информации, высчитывается хэш-сумма, затем всё это сжимается и сохраняется в файле в каталоге .git/objects

Имя файла уникально, т.к. формируется на основании полученного хэша. Затем в файл  .git/index дописывается ещё одна запись о добавленном файле (сохраняется имя файла, а также тот самый хэш, и ещё некоторое количество служебной информации)

**При выполнении команды git commit именно индекс, а не текущее состояние рабочей копии становится новым коммитом. git add помещает текущее состояние файла в индекс. Это и есть индексация.**

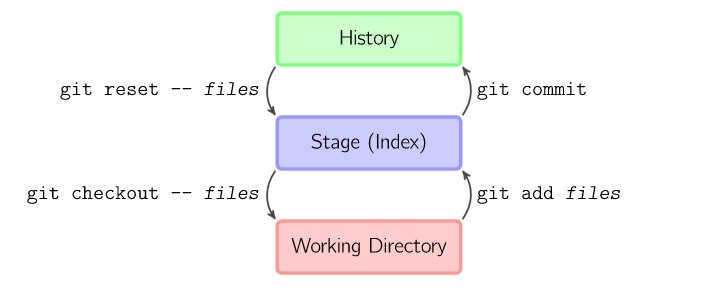
git status нам показывает два списка файлов:

1. Changes to be committed: (выводятся зеленым) — это разница между последним коммитом и индексом, то, что попадет в коммит.
2. Changes not staged for commit: (выводятся красным) — это разница между индексом и рабочей копией, то, что в коммит не попадет.



6. Что такое фиксация файла? Какой командой она выполняется?

**Коммиты** — основные конструктивные элементы временной шкалы проекта Git. Их можно рассматривать как снимки состояния или контрольные точки на временной шкале проекта Git. Коммиты создаются с помощью команды git commit , которая делает снимок состояния проекта на текущий момент времени.

**Git commit - это команда для записи индексированных изменений в репозиторий Git, фиксация файла.**  
  
**Прежде чем создавать очередной коммит, необходимо проиндексировать файлы в рабочей области с помощью команды [git add](https://ru.stackoverflow.com/questions/tagged/git-add" \o "показать вопросы с меткой \"git-add\")**. Новый коммит будет включать текущие состояния индексированных файлов плюс последние сохраненные состояния неиндексированных (но отслеживаемых) файлов. Обратите внимание: **коммит** **включает** в себя не изменения (дельты, патчи) относительно предыдущего коммита, а **"снимок" (англ. shapshot) текущего состояния рабочей области.**  
  
**Чтобы включить в комментарий к коммиту информацию о том какие именно были сделаны изменения в каких файлах надо дать команду:**$ git commit –v  
  
Коммит в git репозитории хранит снимок всех файлов в репозитории. Почти как огромная копия, только эффективнее. **Git пытается быть лёгким и быстрым, так что он не просто слепо копирует весь проект каждый раз, а ужимает коммит в набор изменений или «дельту» между текущей версией и предыдущей**. Это позволяет занимать меньше места. Также Git хранит всю историю о том, когда какой коммит был сделан и кем.

**Из чего состоит коммит**

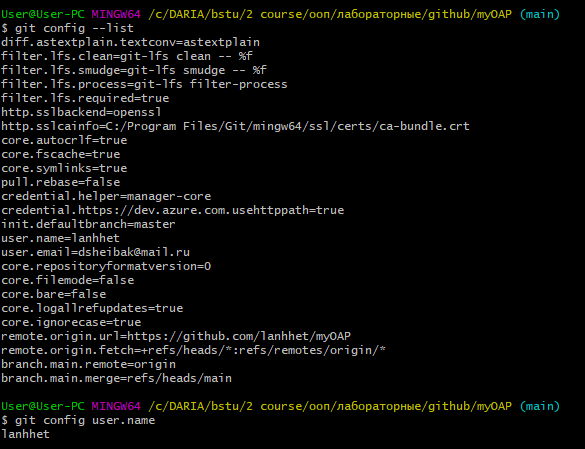
Каждый коммит имеет:

* хэш (уникальный id)
* commit message (сообщение)
* список измененных файлов
* изменения по каждому файлу

Все коммиты можно посмотреть в истории коммитов. История хранит все данные обо всех коммитах проекта. **Показывается история командой $ git log**

7. Продемонстрируйте команду проверки выбранных настроек.

$ git config –list  
$ git config user.name  
$ git config user.email





8. Как инициализировать репозиторий в существующей папке?

[**Репозиторий Git**](https://bitbucket.org/product/ru/code-repository)— это виртуальное хранилище проекта. В нем можно хранить версии кода для доступа по мере необходимости. Для создания нового репозитория используется команда **git init**. **Команду git init выполняют только один раз для первоначальной настройки нового репозитория**. Выполнение команды приведет к созданию нового подкаталога .git в вашем рабочем каталоге. Кроме того, будет создана новая главная ветка.  
  
**git init** используется для начала использования git в проекте, **который не находится под git**. Для проектов, которые уже находятся под git, вы используете git clone.  
  
**Команда git init создает новый репозиторий Git**. С ее помощью можно преобразовать существующий проект без управления версиями в репозиторий Git или **инициализировать новый пустой репозиторий**. Большинство остальных команд Git невозможно использовать без инициализации репозитория, поэтому данная команда обычно выполняется первой в рамках нового проекта.

При выполнении команды git init в текущем рабочем каталоге создается подкаталог .git со всеми необходимыми метаданными Git для нового репозитория. Метаданные включают подкаталоги для объектов, ссылок и файлов шаблонов.   
  
**git init <directory>**

**Создание пустого репозитория Git в указанном каталоге**. При выполнении этой команды будет создан новый подкаталог — он будет содержать только подкаталог .git.

**Если команда git init уже выполнялась по отношению к каталогу проекта** и в нем есть подкаталог .git, команду git init **можно безопасно выполнить** для этого каталога повторно.

Краткое замечание: git init и git clone легко спутать, поскольку **обе команды можно использовать для инициализации нового репозитория** Git. Однако команда git clone зависит от git init и **предназначена для создания копии существующего репозитория.** Выполнение git clone сначала вызывает git init, чтобы создать новый репозиторий, затем копирует данные из существующего репозитория и извлекает новый набор рабочих файлов.

Итак, предположим, что вы уже начали работать над проектом, то есть рабочая папка на локальном компьютере у вас есть. Чтобы инициализировать репозиторий в рабочей папке, надо зайти в нее и выполнить команду:

**git init**

В рабочей папке появится папка .git, что означает, что репозиторий инициализирован. Теперь наша папка не просто папка проекта, но и git-репозиторий.

9. Как указать файлы, за которыми должна следить система?

Для того чтобы начать отслеживать (**добавить под версионный контроль**) новый файл, используется команда **git add**. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

$ git add README

Если вы снова выполните команду status, то увидите, что файл README **теперь отслеживаемый и добавлен в индекс.**Индекс в Git**— это специальная промежуточная область, в которой хранятся изменения файлов на пути от рабочей директории до репозитория. При выполнении коммита в него попадают только те изменения, которые были добавлены в индекс.**

10.Как выполнить фиксацию изменений?

Фиксация изменений выполняется с помощью команды **git commit.** Вам нужно указать что было изменено с помощью небольшого комментария, буквально в несколько предложений. Хорошая практика выполнять фиксацию перед каждым серьезным изменением.

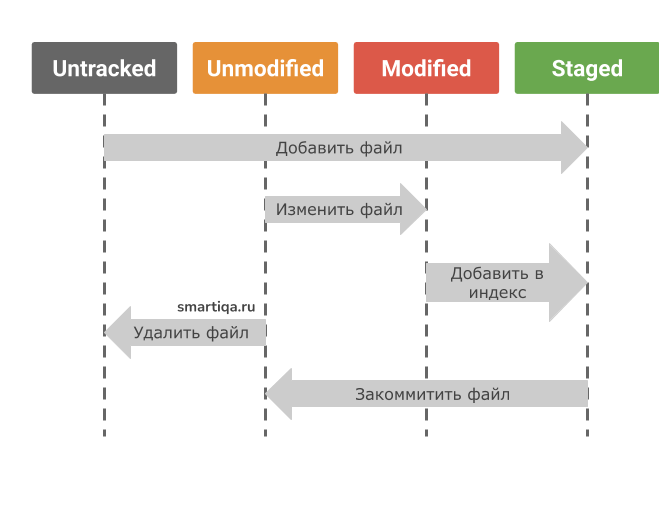
Таким образом, вы будете хранить все версии проекта, от самой первой и до текущей, а также сможете знать, что, когда и где было изменено. Чтобы создать свой первый коммит выполните:

**git commit -m "Initial Commit" -a**  
Команде необходимо передать два параметра, первый - **это -m, ваш комментарий**, второй **-a, означает, что нужно применить действие ко всем измененным файлам**.

**Коммит**– фиксация изменений, внесенных в индекс. Другими словами, коммит – это единица изменений в вашем проекте. **Коммит хранит измененные 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 обратиться.  
  
11. Какой командой определить состояния файлов?

А теперь давайте более подробно разберемся с тем, в каких состояниях могут быть файлы с точки зрения Git. Каждый файл может находится только в одном из двух состояний:  
  
**1.** **Отслеживаемый.** Об этих файлах Git знает и отслеживает изменения в них. Отслеживаемые файлы в свою очередь могут находится в следующих состояниях:

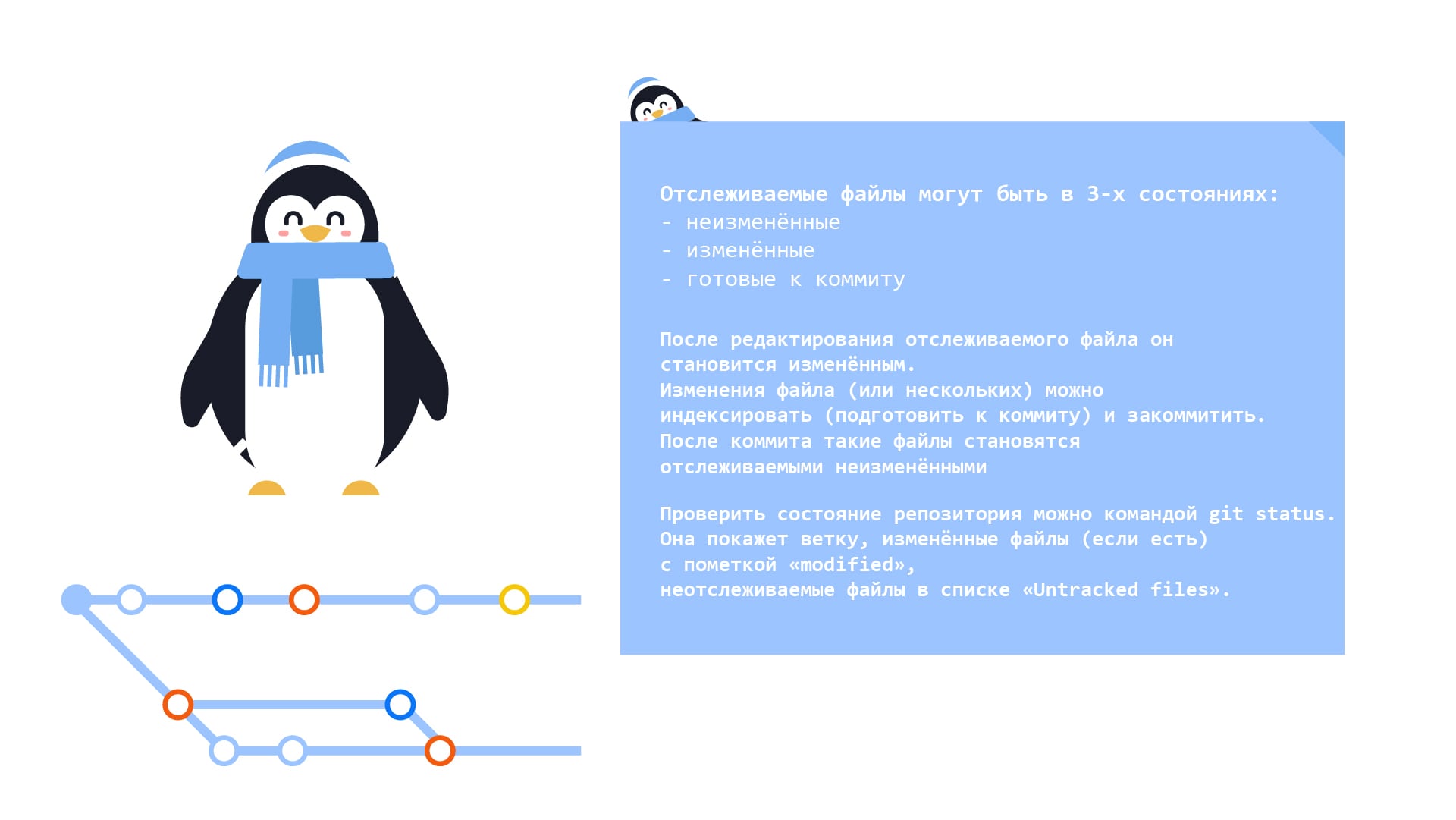
1. **Неизмененный.**То есть с момента последнего коммита в файле не было никаких изменений
2. **Измененный.**То есть с последнего коммита в файле были произведены какие-то изменения.
3. **Подготовленный к коммиту.**Это значит, что вы внесли изменения в этот файл и затем проиндексировали их, и эти изменения будут добавлены в следующий коммит.

**2. Неотслеживаемый.**О неотслеживаемых файлах Git не знает, поэтому изменения в них не будут добавлены в коммит. Это любые файлы в вашем рабочем каталоге, которые не входили в последний коммит и не подготовлены к текущему коммиту.  
  


**Команда git status выводит информацию о статусе файлов, находящихся в репозитории.**

Команда **git status**показывает **состояния файлов в рабочем каталоге и индексе: какие файлы изменены, но не добавлены в индекс; какие ожидают коммита в индексе**.

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда **git status**.  Команда **git status** отображает **состояние рабочего каталога** и раздела проиндексированных файлов. С ее помощью можно проверить индексацию изменений и ***увидеть файлы, которые не отслеживаются Git.***   
  
Понять, что новые файлы **неотслеживаемые** можно по тому, что они 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 коммиты, пока мы его явно об этом не попросим. Это предохранит от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять.



12.Для чего создается файл .gitignore? Поясните его структуру.

Игнорируемые файлы обычно представляют собой **файлы, специфичные для платформы, или автоматически созданные из сборочных систем.** Ниже приводятся некоторые общие примеры:

* Runtime файлы, такие как log, lock, cache или временные файлы (tmp).
* Файлы с конфиденциальной информацией, такие как пароли или ключи API.
* Каталоги зависимостей, такие как /vendor или /node\_modules.
* Build каталоги, такие как /public или /dist.
* Системные файлы
* Конфигурационные файлы IDE или текстового редактора

.gitignore нужен для скрытия файлов и папок от системы контроля версий [Git](http://tyapk.ru/blog/post/learning-git). Это **текстовый файл, в каждой строке которого содержится шаблон файла или каталога, который необходимо проигнорировать.**

Основной синтаксис:

* Каждая строка - отдельный шаблон
* Пустые строки игнорируются
* Строки, начинающиеся с # являются комментариями
* Символ слеша "/" в начале строки указывает на текущую папку (где лежит .gitignore)
* Звёздочка (\*) заменяет любое количество символов
* Две звёздочки (\*\*) используются для указания всех подпапок.
* Восклицательный знак(!) в начале строки инвертирует шаблон (используется для исключений)
* Для экранирования спецсимволов используется обратный слэш "\". Для игнорирования всей директории, правило должно оканчиваться на слэш(/), в противном случае правило считается именем файла
* Вопросительный знак (?) соответствует любому отдельному символу

**Комментарии.   
  
Строки, начинающиеся со знака хэша (#), являются комментариями и игнорируются**. Пустые строки могут быть использованы для улучшения читабельности файла и группировки связанных строк шаблонов.   
 **Слэш.**

**Символ косой черты (/) представляет собой разделитель каталогов**. Наклонная **черта в начале шаблона относится к директории, в которой находится файл .gitignore**. Если шаблон начинается со слеша, то он соответствует файлам и каталогам только в корне хранилища.

**Если шаблон не начинается со слэша, он соответствует файлам и каталогам в любом каталоге или подкаталоге**.

Если шаблон заканчивается косой чертой, то он соответствует только каталогам. Когда каталог игнорируется, все его файлы и подкаталоги также игнорируются.

**Имена файлов.**

Самый простой шаблон - это просто имя файла без каких-либо специальных символов.

Изображение выглядит как стол

Автоматически созданное описание

**Подстановка.**Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как стол

Автоматически созданное описание **Квадратные скобки.**

**[…] - Совпадает с любыми символами, заключенными в квадратные скобки. Когда два символа разделены дефисом - обозначает диапазон символов**. Диапазон включает в себя все символы, находящиеся между этими двумя символами. ***Диапазон может быть алфавитным или цифровым.***

Если первый символ после [ является восклицательным знаком (!), то образец соответствует любому символу, кроме символов из указанного набора.

Изображение выглядит как стол

Автоматически созданное описание

**Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.**

**Файл игнорирования .gitignore позволяет исключить файлы из проверки в репозитории**. Файл содержит шаблоны, которые описывают, какие файлы и каталоги следует игнорировать.

Пример:

# Игнорировать файл foo.txt.

foo.txt

# Игнорировать html файлы

\*.html

# Но конкретно foo.html не игнорировать

!foo.html

# Игнорировать rar файлы в корне проекта

# Допустим файл /temp/main.rar не будет проигнорирован т.к. он не в корне

/\*.rar

# Игнорировать css файлы из папки bar не включая подпапки

# Допустим файл /bar/temp/main.css не будет проигнорирован т.к. он в подпапке temp

/bar/\*.css

# Игнорировать js файлы из папки bar и подпапок, если таковые будут

/bar/\*\*.\*.js

13.Как используется команда git diff?

1) **Сравнение с последним коммитом.**  
  
Для вывода изменений в файлах по сравнению с последним коммитом, используется git diff без параметров:

**git diff**

**Команда выводит изменения в файлах, которые еще не были добавлены в индекс**. Сравнение происходит с последним коммитом.

**2) Сравнение с последним коммитом, включая файлы в индексе.**

Если вы изменили какие-нибудь файлы в вашем рабочем каталоге и добавили один или несколько из них в индекс (с помощью git add), то команда git diff не покажет изменения в этих файлах. **Чтобы показать изменения в файлах, включая файлы, добавленные в индекс, используется ключ --cached:**

**git diff --cached**

**3) Сравнение коммитов.  
  
Команда git diff позволяет сравнивать два различных коммита**. Сначала нужно определить хеш (ID) коммитов, которые требуется сравнивать. Можно воспользоваться командой git log, чтобы вывести список коммитов и их идентификаторы:

git log

Теперь сравним два коммита. Для этого в качестве первого аргумента команде git diff указывается хеш первого коммита, а вторым аргументом хеш второго коммита, например:

**git diff 4612297 5e356cf  
  
4) Сравнение двух веток.**  
  
Для вывода всех изменений между концами двух веток, необходимо для git diff указать имена веток:

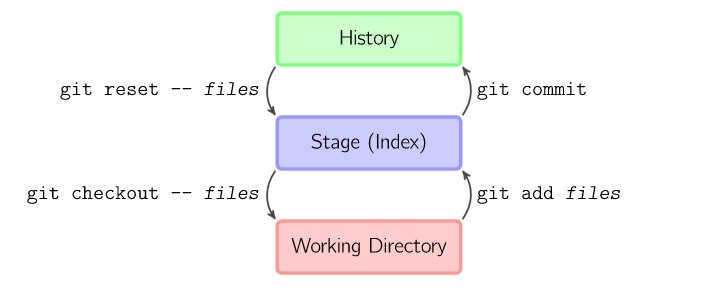
**git diff branch1 branch2**

**5) Сравнение файлов между двумя ветками.**  
  
Чтобы сравнить конкретные файлы между двумя ветками используется команда:

git diff branch1 branch2 ./myfile.cpp

Вместо branch1, branch2 нужно указать название веток, а вместо myfile.cpp путь до сравниваемого файла. Через пробел можно добавить еще файлы для сравнения

14.Как используется команда git commit?

**Git commit - это команда для записи индексированных изменений в репозиторий Git.**  
  
Прежде чем создавать очередной коммит, необходимо проиндексировать файлы в рабочей области с помощью команды [**git add**](https://ru.stackoverflow.com/questions/tagged/git-add). Новый коммит будет включать текущие состояния индексированных файлов плюс последние сохраненные состояния неиндексированных (но отслеживаемых) файлов. Обратите внимание: коммит включает в себя не изменения относительно предыдущего коммита, а **"снимок" (англ. shapshot) текущего состояния рабочей области.**  
  
  
  
**Коммиты** — **основные конструктивные элементы временной шкалы проекта Git**. **Их можно рассматривать как снимки состояния или контрольные точки на временной шкале проекта Git**. Коммиты создаются с помощью команды git commit, которая делает снимок состояния проекта на текущий момент времени. Коммиты снимков состояния Git всегда выполняются в локальный репозиторий.

Флаг -m позволяет задать сообщение фиксации.

Флаг -a позволяет автоматически индексировать все отслеживаемые файлы перед их фиксацией.

15.Как используется команда git log? Какие у нее есть параметры?

Одним из основных и наиболее мощных инструментов для этого является команда **git log**. По умолчанию (без аргументов) git log **перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке — последние коммиты находятся вверху.**

**Одним из самых полезных аргументов является -p или --patch, который показывает разницу (выводит патч), внесенную в каждый коммит.** Так же вы можете ограничить количество записей в выводе команды; используйте параметр -2 для вывода только двух записей.  
  
Например, если вы хотите увидеть сокращенную статистику для каждого коммита, вы можете использовать опцию **--stat.**

Следующей действительно полезной опцией является **--pretty**. Эта опция меняет формат вывода. Существует несколько встроенных вариантов отображения. Опция **oneline** выводит каждый коммит в одну строку, что может быть очень удобным если вы просматриваете большое количество коммитов. К тому же, опции **short, full и fuller** делают вывод приблизительно в том же формате, но с меньшим или большим количеством информации соответственно

Изображение выглядит как текст, монитор, снимок экрана, экран

Автоматически созданное описание  
  
Наиболее интересной опцией является **format**, которая позволяет указать формат для вывода информации.

16.Как используется команда git commit --amend?

Эта команда откроет в вашем текстовом редакторе сообщение вашего последнего коммита для того, чтобы вы могли его исправить. Когда вы сохраните его и закроете редактор, будет создан новый коммит, содержащий это сообщение, который теперь и будет вашим последним коммитом.

*Крайне часто разработчики делают коммит и сразу же понимают, что забыли добавить часть файлов через git add.* **Оставшуюся часть изменений можно дослать следующим коммитом либо, если изменения ещё не были отправлены во внешнюю систему, можно добавить изменения в текущий коммит**. Для этого во время коммита добавляется флаг –amend.

В реальности **--amend**не добавляет изменения в существующий коммит, этот флаг приводит к откату коммита (через reset) и выполнению нового коммита с новыми данными. Поэтому мы и видим ровно один коммит, хотя команда git commit выполнялась два раза (первый раз — когда сделали ошибочный коммит).

17.Как отобразить удаленные репозитории?

Git позволяет работать с несколькими удаленными репозиториями. **Для просмотра списка удаленных репозиториев, которые сконфигурированы в данный момент для данного локального репозитория, используется команда:**

**git remote -v**

Ключ -v означает verbose и используется, чтобы выводить подробную информацию.

18.Как извлечь данные из удаленного репозитория?

1. Команда **git fetch** связывается с удалённым репозиторием и забирает из него все изменения, которых у вас пока нет и сохраняет их локально. Она сливает все данные с проекта, которые находятся в нашем *удаленном репозитории*. Все данные, которых у нас нет, она сливает, **но не применяет в наши ветки.**  
  
2. Команда **git pull** работает как комбинация команд **git fetch и git merge**, т. е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой.

19.Как отправить данные в удаленный репозитроий?

С помощью команды **git push**. Это отправка данных на сервер, в удаленный репозиторий, на github. **Данные - это коммиты и ветки.**

20.Какая команда позволяет отобразить удаленные репозитории,  
связанные с текущим локальным?

Git позволяет работать с несколькими удаленными репозиториями. **Для просмотра списка удаленных репозиториев, которые сконфигурированы в данный момент для данного локального репозитория, используется команда:**

**git remote -v**

Ключ -v означает verbose и используется, чтобы выводить подробную информацию.

21. Каким образом можно получить изменения из удаленного  
репозитория в локальный?

1) Выполним команду git remote add upstream [<ссылка](https://github.com/gilamran/tsc-watch.git) на репозиторий коллеги, откуда мы будем брать изменения> (добавим удаленный сервер в качестве вышестоящего):

2) Далее команда git fetch upstream (извлечь изменения)

3) После выполняем git merge upstream/main

4) Файл появился в локальном репозитории

22.Для чего используется команда fetch? В чем отличие команды fetch  
от pull?

1. Команда **git fetch** связывается с удалённым репозиторием и забирает из него все изменения, которых у вас пока нет и сохраняет их локально. Она сливает все данные с проекта, которые находятся в нашем *удаленном репозитории*. Все данные, которых у нас нет, она сливает, **но не применяет в наши ветки.**  
  
2. Команда **git pull** работает как комбинация команд **git fetch и git merge**, т. е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой.

23.Для чего используется команда merge? В чем отличие merge от  
rebase?

**Git merge**

**Слияние** — обычная практика для разработчиков, использующих системы контроля версий. Независимо от того, созданы ли ветки для тестирования, исправления ошибок или по другим причинам, слияние фиксирует изменения в другом месте. Слияние принимает содержимое ветки источника и объединяет их с целевой веткой. В этом процессе изменяется только целевая ветка. История исходных веток остается неизменной.

+: простота, сохраняет полную историю и хронологический порядок

-: история коммитов может быть заполнена (загрязнена) множеством коммитов

**Git rebase**Вместо слияния можно выполнить перебазирование функциональной ветки на главную ветку main с помощью этой команды.

В результате вся функциональная ветка окажется поверх главной ветки main, включая в себя все новые коммиты в ветке main. Если вместо команды merge при коммитах используется rebase, эта команда *перезаписывает* историю проекта, создавая новые коммиты для каждого коммита в исходной ветке.

+: чистая история, упрощение потенциально сложной истории, очищает промежуточные коммиты, делая их одним коммитом

-: сжатие промежуточных коммитов до одного может скрыть контекст

**Главное преимущество rebase — более чистая история проекта.**

**Merge** - это средство интеграции изменений из двух (и даже более) последовательностей изменений. Когда вы хотите сказать - я беру изменения из своей ветки и из другой, и совмещаю их друг с другом - вам нужно делать merge. Слияние не изменяет имеющуюся историю изменений, а наоборот, продолжает её. Создавая мёрж-коммит вы как бы говорите всем - "я совместил изменения из этих двух веток, все кому нужны изменения из этих обоих веток ОДНОВРЕМЕННО (например, из ветки feature\_buy\_button и ветки master) - можете взять этот коммит".  
  
**Rebase**-  **Rebase сжимает все изменения в один «патч». Затем он интегрирует патч в целевую ветку.** Если вместо команды merge при коммитах используется rebase, эта команда *перезаписывает* историю проекта, создавая новые коммиты для каждого коммита в исходной ветке.

24. Что такое pull request?

**Pull Request** — это запрос на вливание изменений из вашей ветки в основную ветку исходного репозитория.   
  
**Пул реквест** (pull request) -- это *запрос* к управляющему каким-либо репозиторием (человеку, группе людей или вообще роботу) *на применение изменений* (из вашего репозитория и/или указанной вами ветки).  
  
25. Какая команда позволяет отобразить историю репозитория?

По умолчанию (без аргументов) **git log** перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке — последние коммиты находятся вверху.

26.Продемонстируйте создание новых веток в Git? Что такое  
ветвление? Что такое указатель HEAD?

**Ветка**. Дочерняя версия основного репозитория. Она входит в его состав, но не влияет на работу. После того, как разработчики закончат работу над новой функцией или исправят все баги, можно совместить дочерний и родительский репозитории.

*Используя ветвление, вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию.*

**HEAD** – так называемый курсор Git. Главное назначение HEAD - определять, в каком состоянии находится рабочая копия (напомним, что рабочая копия – это все файлы репозитория, за исключением директории .git/). **На какой коммит указывает HEAD – из того коммита и загружаются файлы в рабочую директорию.**

**HEAD — это указатель на текущую ветку, которая, в свою очередь, является указателем на последний коммит, сделанный в этой ветке. Это значит, что HEAD будет родителем следующего созданного коммита. Как правило, самое простое считать HEAD снимком**вашего последнего коммита**.**

### Преимущество веток в их независимости. Вы можете вносить изменения в файлы в одной ветке, например, пробовать новую функцию, и они никак не скажутся на файлах в другой ветке.  Изображение выглядит как текст Автоматически созданное описание **Создание новых веток.** **Способ 1. Команды git branch + git checkout**

### **Способ 2. Команда git checkout -b**

27.Как используется команда git checkout?

Изображение выглядит как стол

Автоматически созданное описание  
  
Как и git branch, git checkout – очень многофункциональная команда. Главное ее назначение – перемещать указатель HEAD.

28. Как выполнить включение изменений из одной ветки в другую?

Чтобы выполнить мердж (от англ. **merge**– слияние), в Git предусмотрена команда **git merge**.  
  
**git merge <сливаемая ветка>**Вместо слияния можно выполнить перебазирование функциональной ветки на главную ветку main с помощью **git rebase.**

29.Какие проблемы могут быть при слиянии и как они разрешаются?

Очень часто во время слияния веток оказывается, что ваши изменения удаляют или переписывают информацию в уже существующих файлах. Такая ситуация называется **файловым конфликтом**. Git останавливает выполнение слияния, пока вы не разрешите конфликт.  
  
**По сути, Git сталкивается с проблемой: у него есть два файла с одним и тем же именем, и он не знает, какой из них взять.** Поэтому обращается к нам за помощью.  
  
Общий подход к **разрешению конфликтов** такой:

1. Непосредственно разрешить конфликт одним из двух рассмотренных немного ниже способов. Либо, если возникновение конфликта стало неожиданным для вас, можно выполнить **git merge --abort**. Эта команда прервет слияние и вернет все, как было.
2. Сообщить Git, что мы разрешили конфликт, добавив все файлы с разрешенными конфликтами в индекс. Сделать это можно уже знакомой командой **git add <конфликтный файл>** для каждого конфликтного файла.
3. Продолжить слияние, выполнив **git merge --continue**. Либо вручную создать **merge-коммит** уже знакомой командой **git commit**.

Выше мы уже сказали, что существует **два способа разрешать конфликты**, вот они:

1. Первый способ. Разрешить конфликт вручную. Тогда мы можем самостоятельно изменить конфликтные файлы, сделав их такими, какими мы хотим их видеть.
2. Второй способ. Просто выбрать один из двух файлов.

30.Что такое GitLab?  
  
**GitLab** — веб-приложение и система управления репозиториями программного кода для Git.

**GitLab предлагает решение для хранения кода и совместной разработки масштабных программных проектов.** Репозиторий включает в себя систему контроля версий для размещения различных цепочек разработки и веток, позволяя разработчикам проверять код и откатываться к стабильной версии софта в случае непредвиденных проблем.

GitLab является конкурентом GitHub, в котором среди многих других проектов размещается разработка ядра Linux Линуса Торвальдса. *Поскольку GitLab разрабатывается на той же основе управления версиями (Git)*, принцип их работы схож. GitLab поддерживает как публичные, так и неограниченное количество частных ветвей разработки.

  
  
GitLab — это отличный инструмент для разработчиков, который предоставляет следующие возможности:

— управление публичными и приватными git-репозиториями;   
— управление пользователями и группами, правами доступа к git-репозиториям;   
— отслеживание ошибок, деплой (*процесс «разворачивания» веб-сервиса, например, сайта, в рабочем окружении. рабочее окружение — место, где сайт запускается и доступен для запросов, это может быть как готовый хостинг, так и своя собственная серверная инфраструктура*), анализ кода;   
— интеграция с разными CI-системами CI – программами контроля ресурсов  
  
**При разработке ПО, самым важным является контроль версий, на что как раз и сделали основной упор при разработке GitLab.**

Система контроля версий, является компонентом, который управляет программами, которые в свою очередь отслеживают любые изменения, вносимые в любую информацию, код программы и т.д. В серьезные разработки, обычно привлечены несколько разработчиков, поэтому важно иметь контроль версий, потому что в последствии становится нереально следить за версиями кода и внесенными изменениями. **Итого, получается, что имея инструмент, такой как GitLab, разработка будет успешной, и гарантировано без конфликтов и совпадений фрагментов кода.**

Изображение выглядит как текст

Автоматически созданное описание