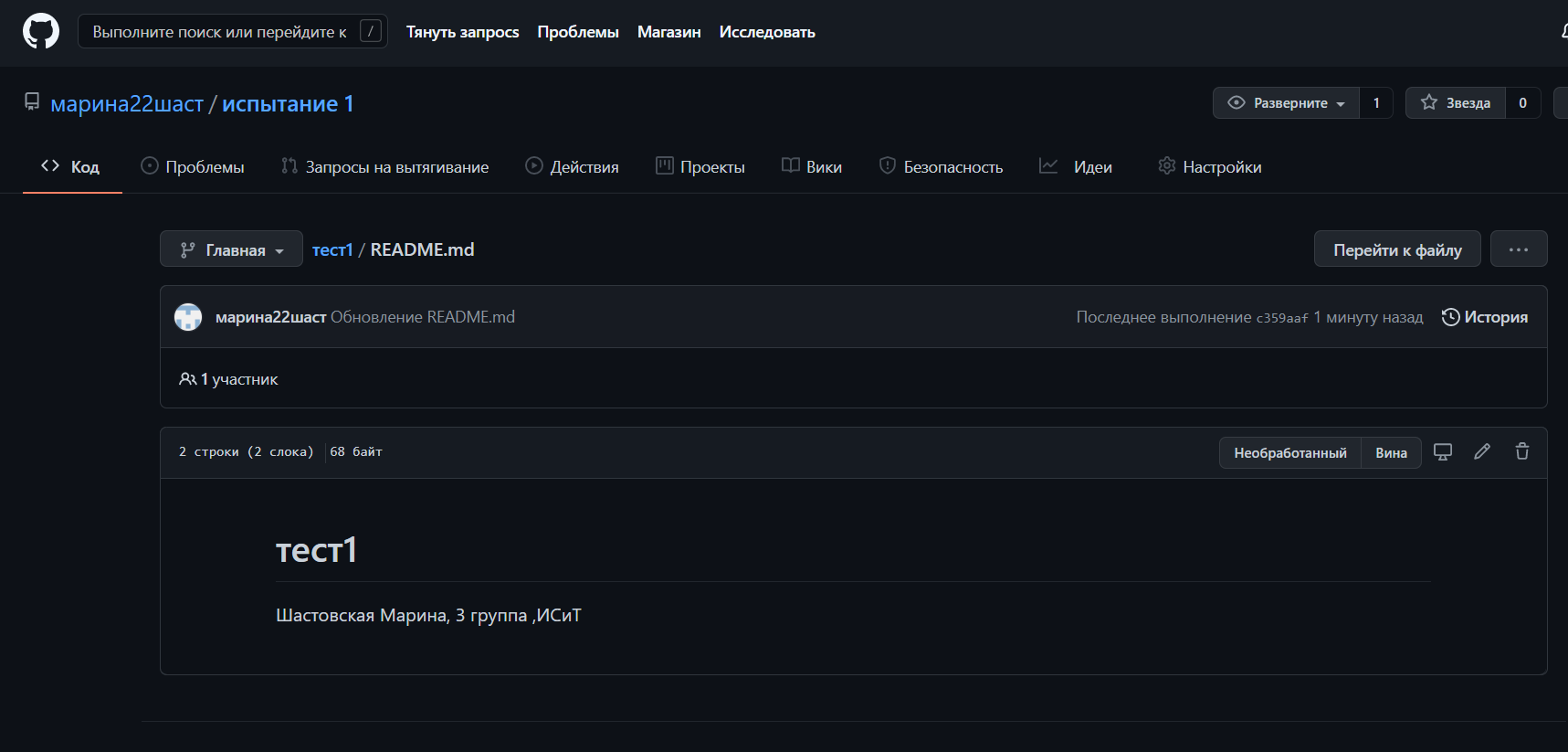
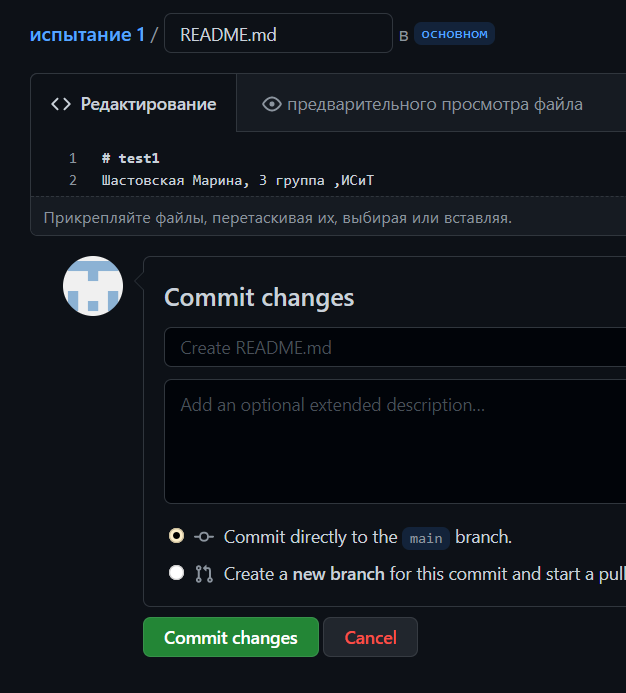
**Отчёт по лабораторной работе №1**

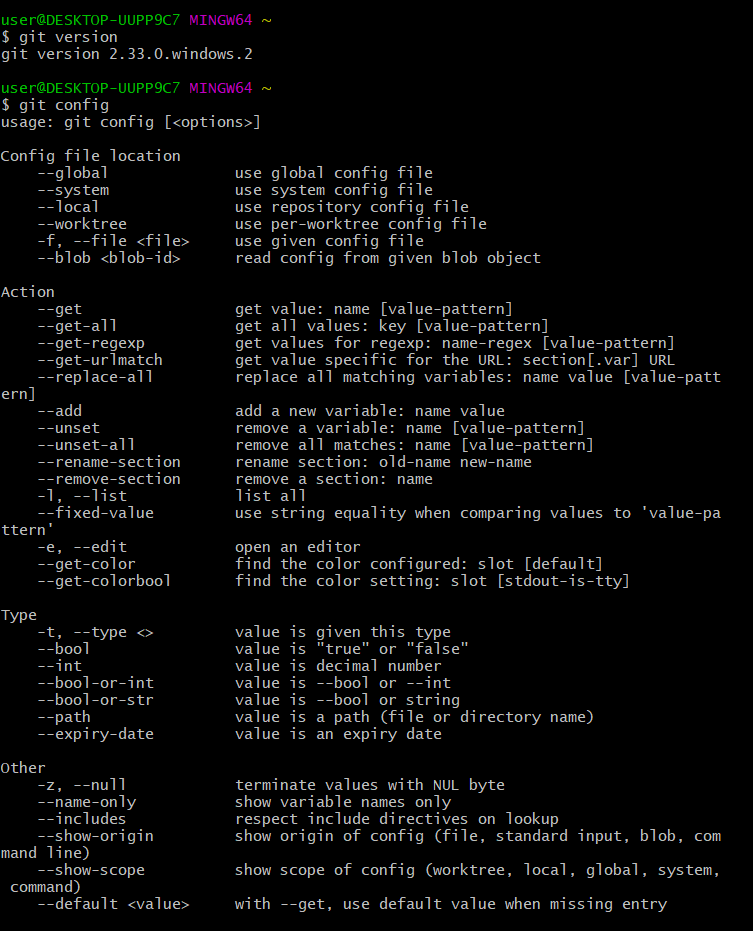
**“Системы контроля версий”**

1) Создайте репозиторий для вашего первого проекта. Модифицируйте файл README.md так, чтобы он содержал ваше имя, номер группы и специальность.

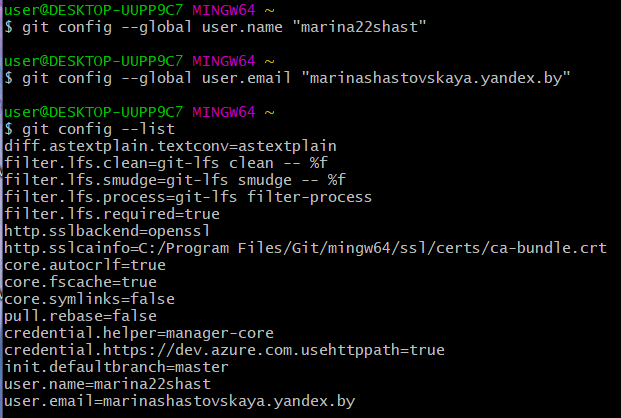
****

**2) Базовый уровень. Работа с клиентом GitBush.**

Проверьте версию. Выпоните конфигурацию (git config).



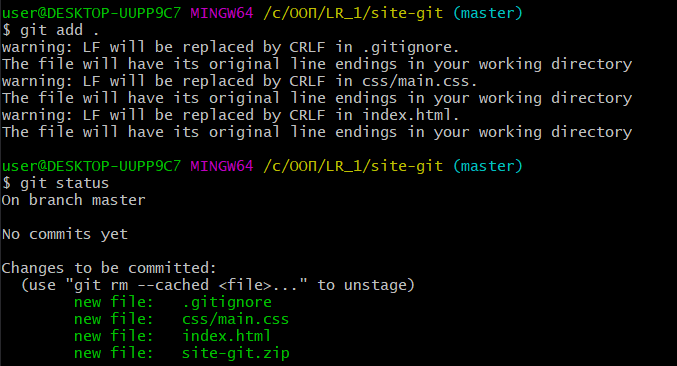
Настраиваем имя пользователя. Через команду git config.Проверка выбранных настроек через git config –list, выводящая список всех обнаруженных в текущий момент параметров.



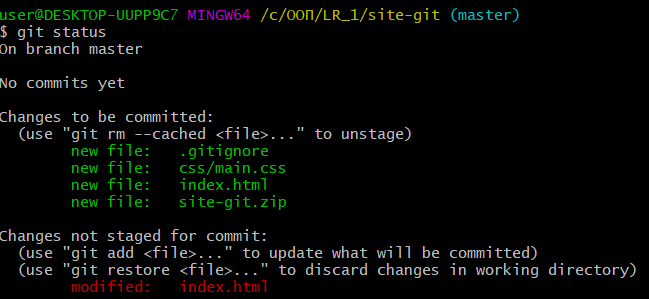
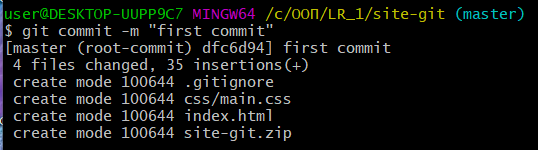
## Научитесь клонировать репозитории (clone).

## 

## git add .-Команда добавляет новые или измененные файлы в ваш рабочий каталог в промежуточную область Git.

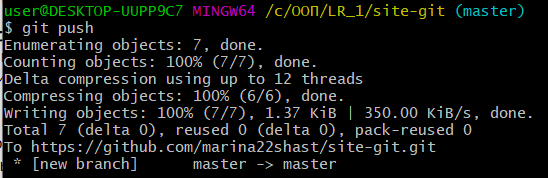


Команда **$ git status** позволяет отследить состояние репозитория. Эта команда позволяет узнать, какие изменения необходимо зарегистрировать git (при необходимости, отменить).

git commit (фиксируем изменения)

## git push (проталкиваем изменения)

Это отправка данных на сервер, в удаленный репозиторий, на github. Данные - это коммиты и ветки.

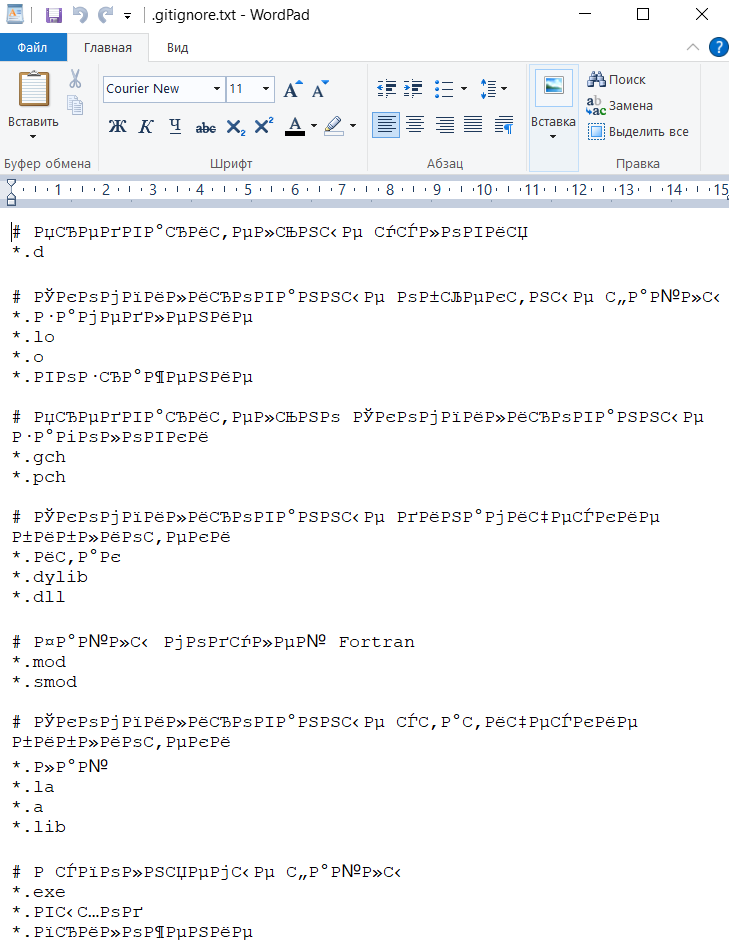


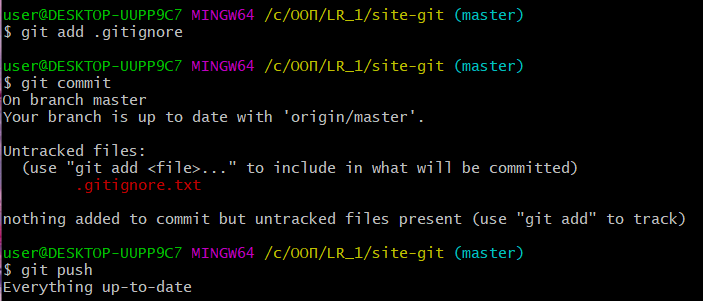
**Настройте gitignore файл для вашего с# репозитория.**

# **Коллекция шаблонов.gitignore**-Это коллекция на GitHub. Мы используем этот список для заполнения шаблонов, доступных в интерфейсе GitHub.com при создании новых репозиториев и файлов..gitignore.

Шаблон должен содержать набор правил, помогающих репозиториям Git работать с определенным языком программирования, фреймворком, инструментом или средой.

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





**Изучите команды удаления и пермещения файлов.**

## Удаление файлов

## 

## Перемещение файлов

## 

## 

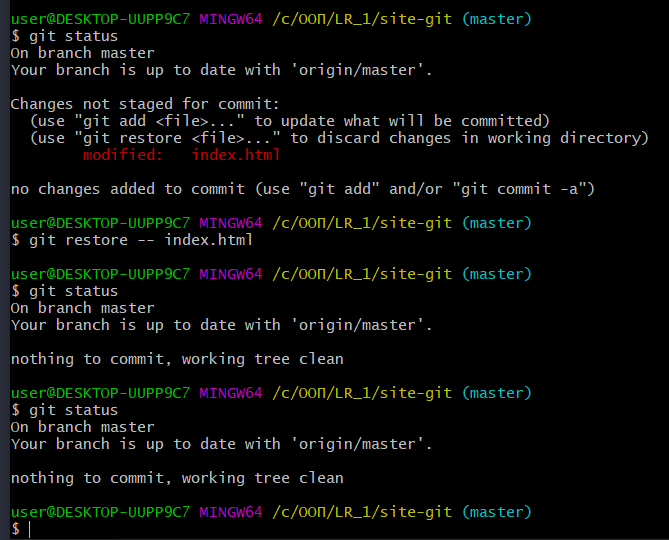
## git log. выводит в обратном хронологическом порядке список сохраненных в данный репозиторий версий

## 

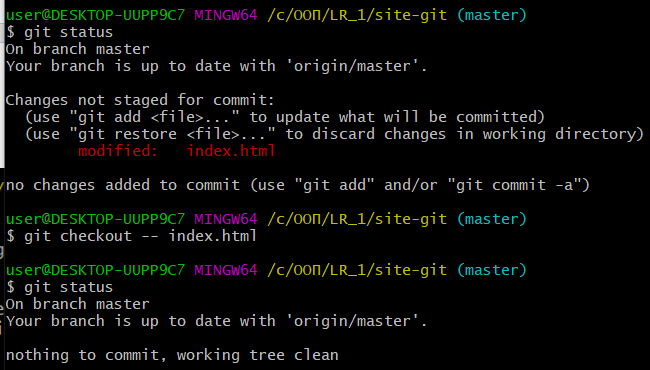
Сделайте отмену внесенных в файл изменений

***Git restore; git checkout***

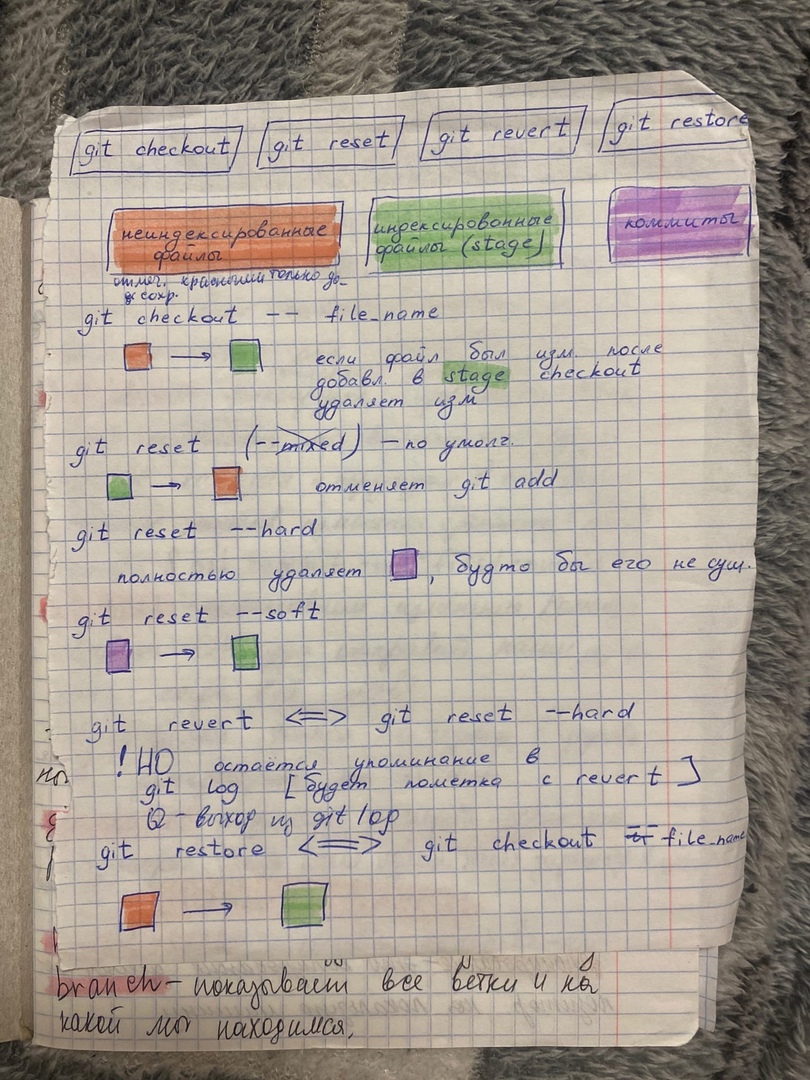
**git-restore** — восстановит файлы рабочего дерева.



[**git checkout**](https://git-scm.com/docs/git-checkout)**-** выполняет две вещи: переключает ветви и восстанавливает файлы до определенного состояния.



****



**3) Работа с ветками**

**Branch+(имя)-**создаёт новую ветку.

**Branch**-показывает все ветки и на какой мы находимся.

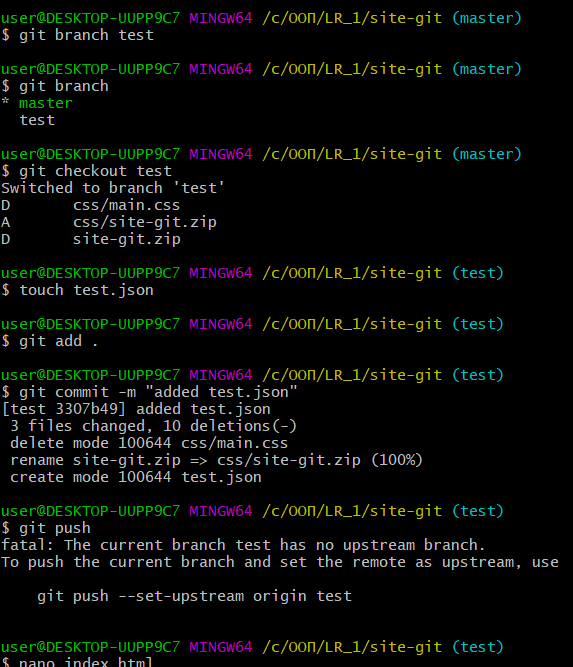
**Checkout-** переключает на ветку.

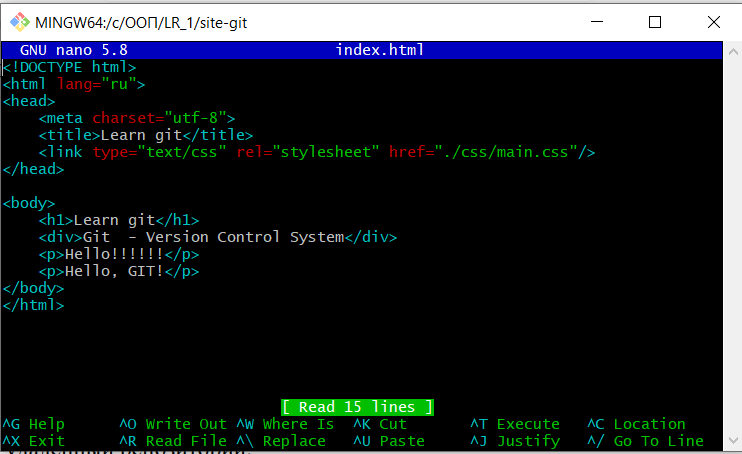
**Touch**-создаёт файл.

**Nano**-редактор файлов в консольной строке.

**Merge-**слияние веток

**Mergetool**-разрешает конфликтную ситуацию.



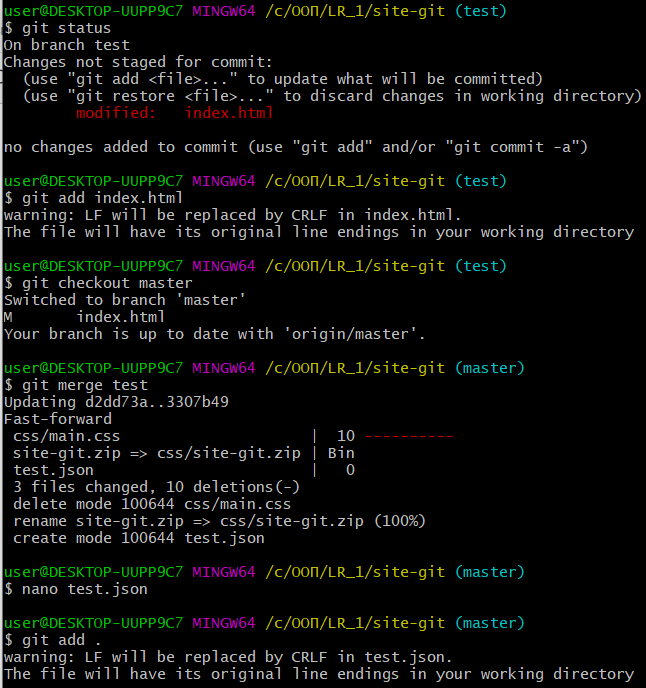


В конце нажимаем Ctrl S+ Ctrl X

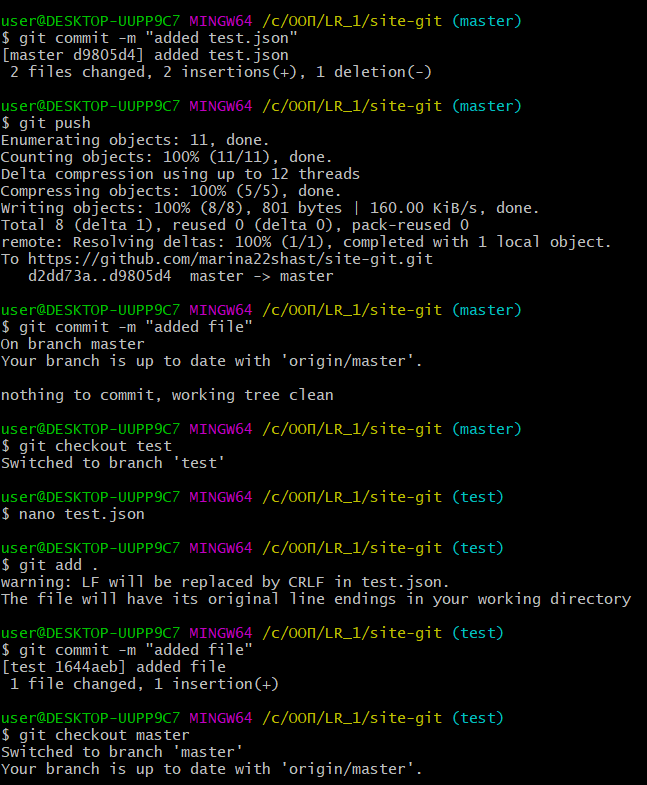
**Checkout-** переключает на ветку. **Merge-**слияние веток

Выполните слияние (merge) ветвей.

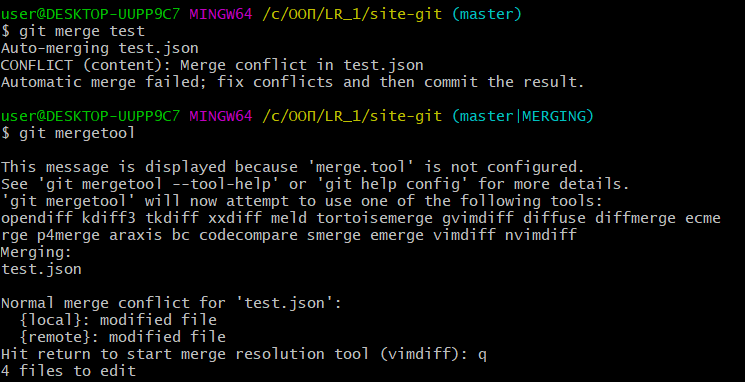
По очереди отредактируйте файл test.json в ветках master и test. Выполните commit. Выполните слияние. Разрешите ситуацию конфликта слияний.

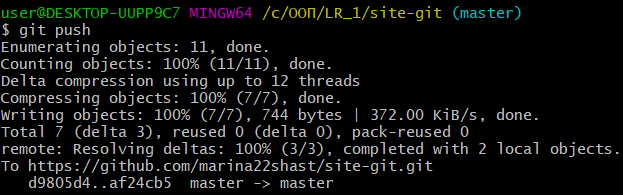


**Mergetool**-разрешает конфликтную ситуацию.



**Merge-**слияние веток

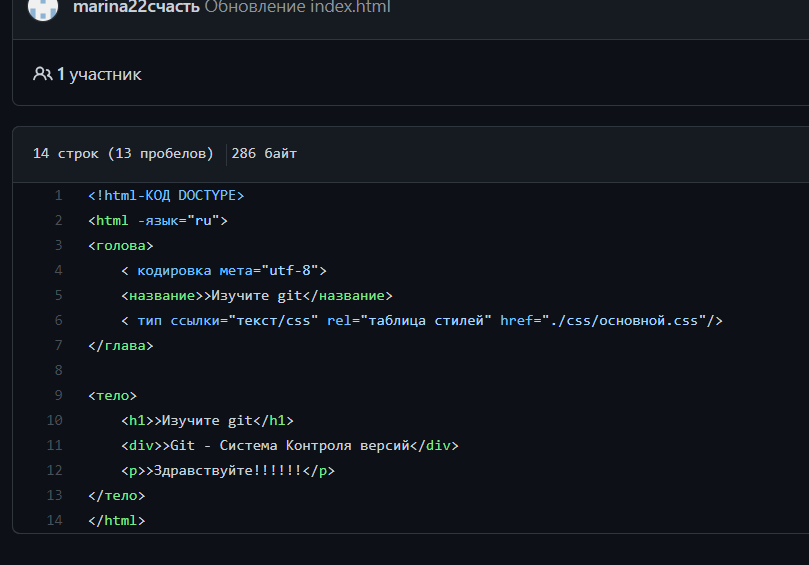


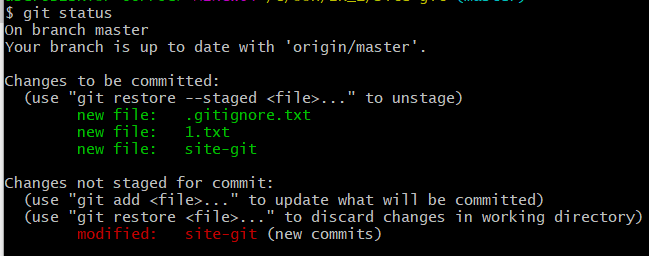


**4) Работа с удаленными репозиториями**

Выполните команды загрузки (push) и выгрузки (pull, fetch) в удаленный репозиторий.

Загрузить содержимое из удаленного репозитория можно с помощью двух команд: git pull и git fetch.

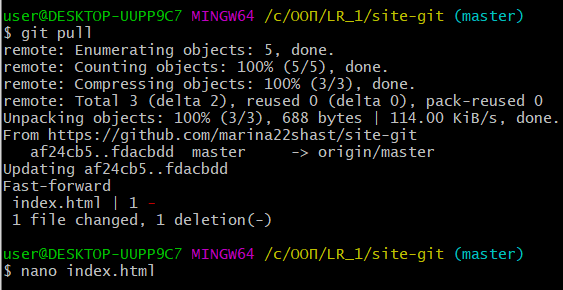


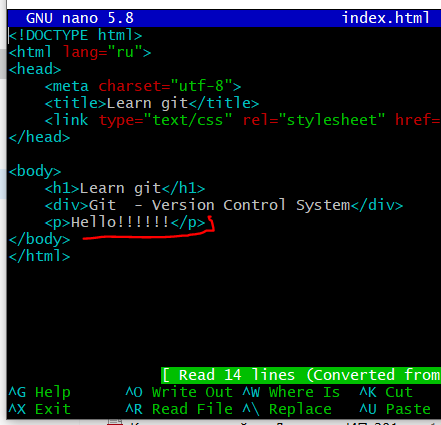


git pull- забирает изменения из удаленного репозитория и интегрирует их с изменениями в локальном репозитории.

git pull действует более агрессивно: она загружает удаленное содержимое для активной локальной ветки и сразу выполняет команду git merge, создавая коммит слияния для нового удаленного содержимого.

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.

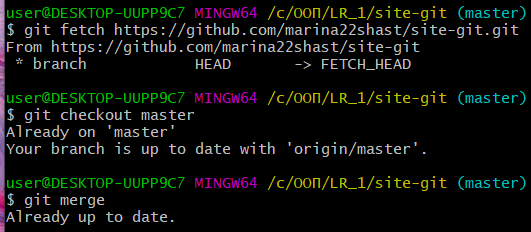




git fetch загружает из удаленного репозитория в ваш локальный репозиторий.

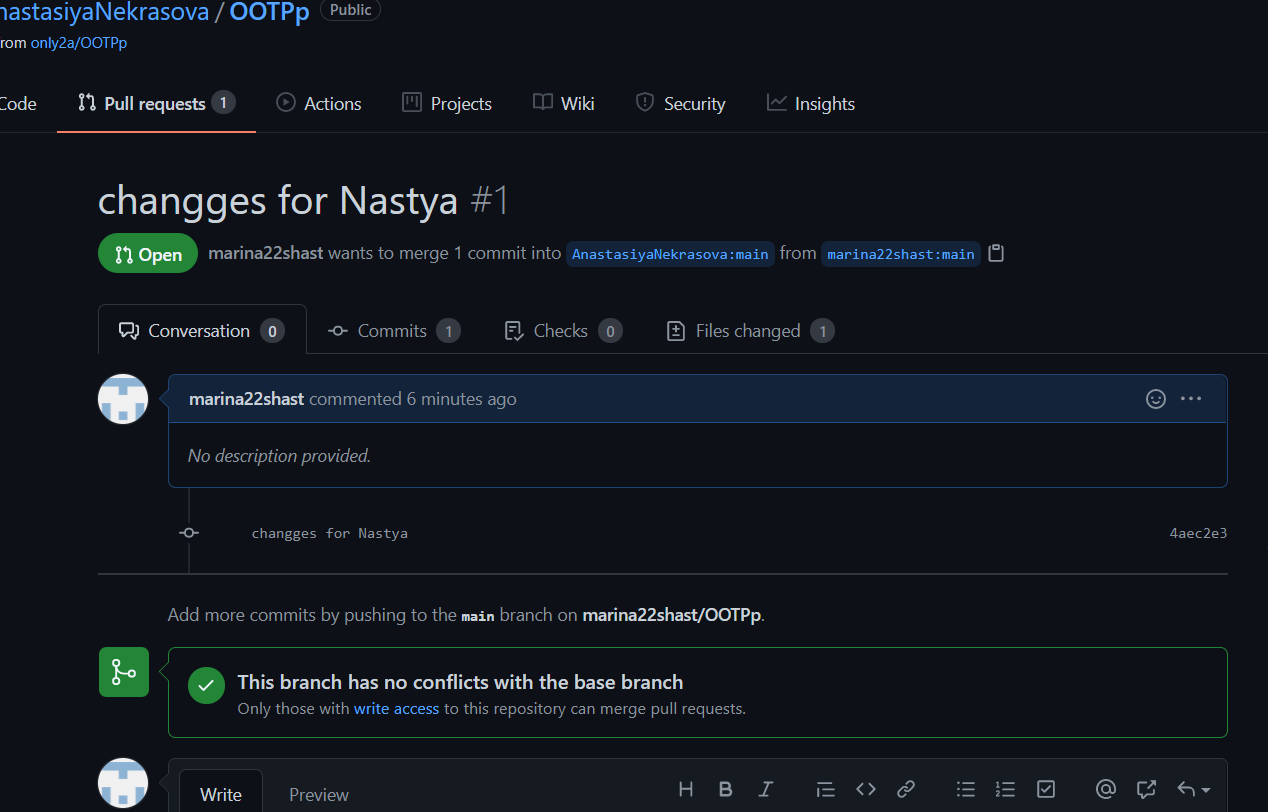
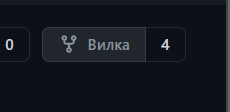
git fetch - «безопасным» вариантом. Она загружает удаленное содержимое, но не обновляет рабочее состояние локального репозитория, оставляя текущую работу нетронутой.

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.**



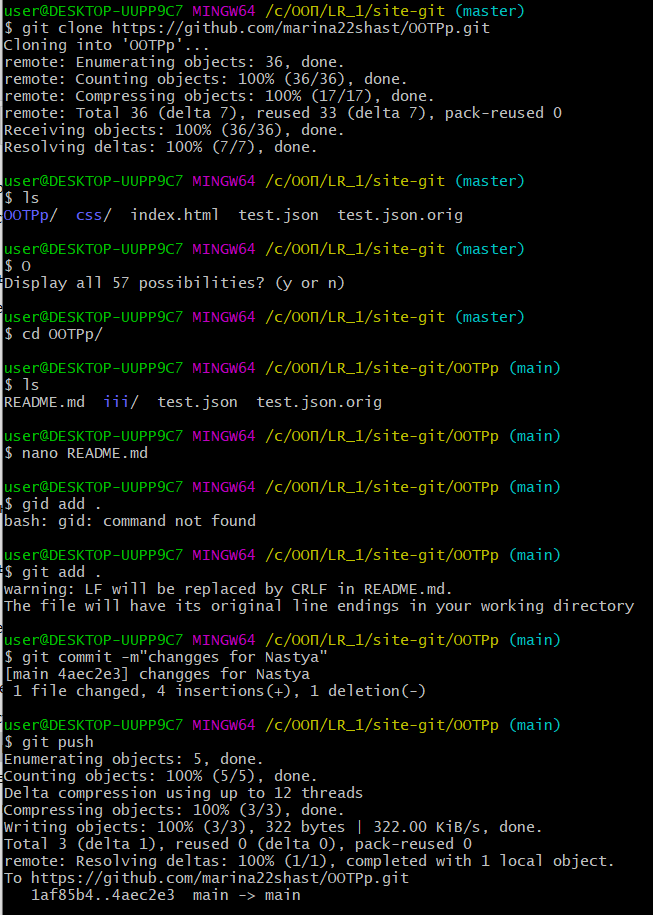
**5) Совместная работа**

Найдите копку Fork и нажмите ее. После этого создается копия этого репозитория, но уже в вашем аккаунте.



И теперь этот уже свой репозиторий клонируем себе на локальную машину. Внесите изменения в файл. Сделайте commit. Отправьте изменения в свой удаленный облачный репозиторий.

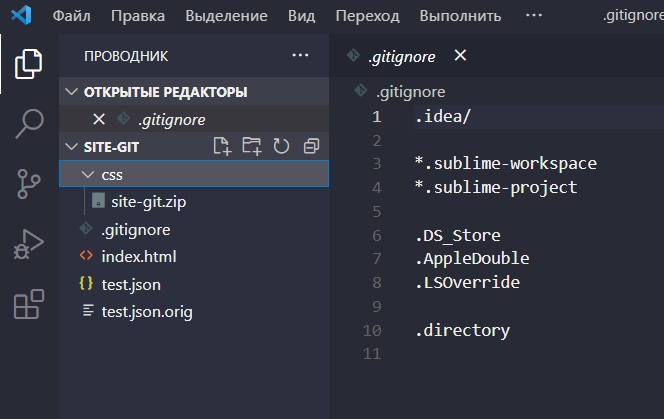
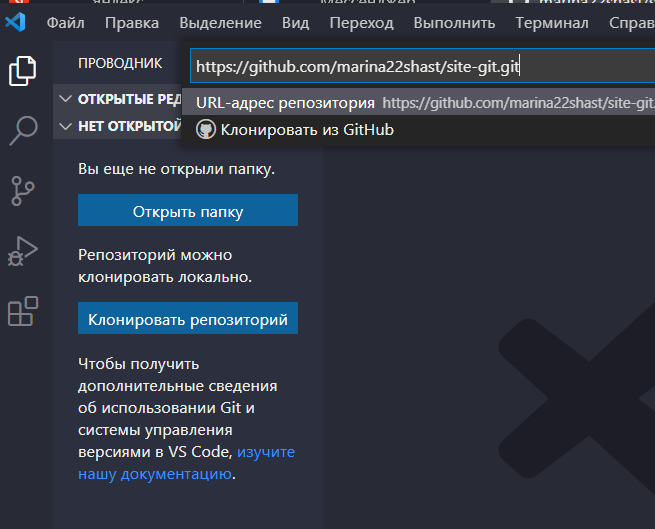
Ls-просмотр всех файлов



**После того как запушили кликаем.**

Создаем Pull requests через интерфейс Github. 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

6) Интеграция со средой разработки

**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.   
  
2. Какой принцип хранения файлов использует Git?  
  
Git хранит полные копии файлов, только заменяя неизмененные файлы на ссылки.

Таким образом Git является своего рода небольшой файловой системой.  
3. В чем отличие git от других систем контроля версий?

Основное отличие Git от любой другой СКВ - это подход к работе со своими данными.

Большинство других систем хранят информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени.   
  
Git не хранит и не обрабатывает данные таким способом. 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 ?

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

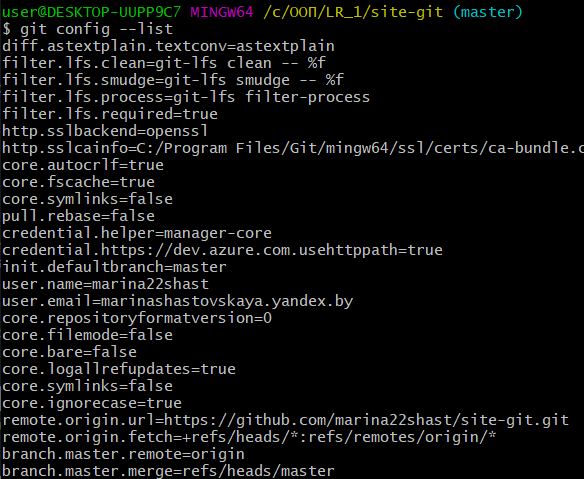
Чтобы проиндексировать его, необходимо выполнить команду git add. Индексирование (также: indexing, staging) — это процесс добавления текущего содержимого (изменённого) файла в индекс (также: index area, staging area).  
6. Что такое фиксация файла? Какой командой она выполняется?

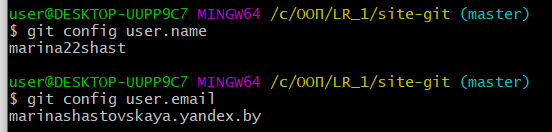
Коммиты создаются с помощью команды git commit , которая делает снимок состояния проекта на текущий момент времени.

Git commit - это команда для записи индексированных изменений в репозиторий Git, фиксация файла.

7. Продемонстрируйте команду проверки выбранных настроек.

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



  
8. Как инициализировать репозиторий в существующей папке?  
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.

При выполнении команды git init в текущем рабочем каталоге создается подкаталог .git со всеми необходимыми метаданными Git для нового репозитория.   
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.

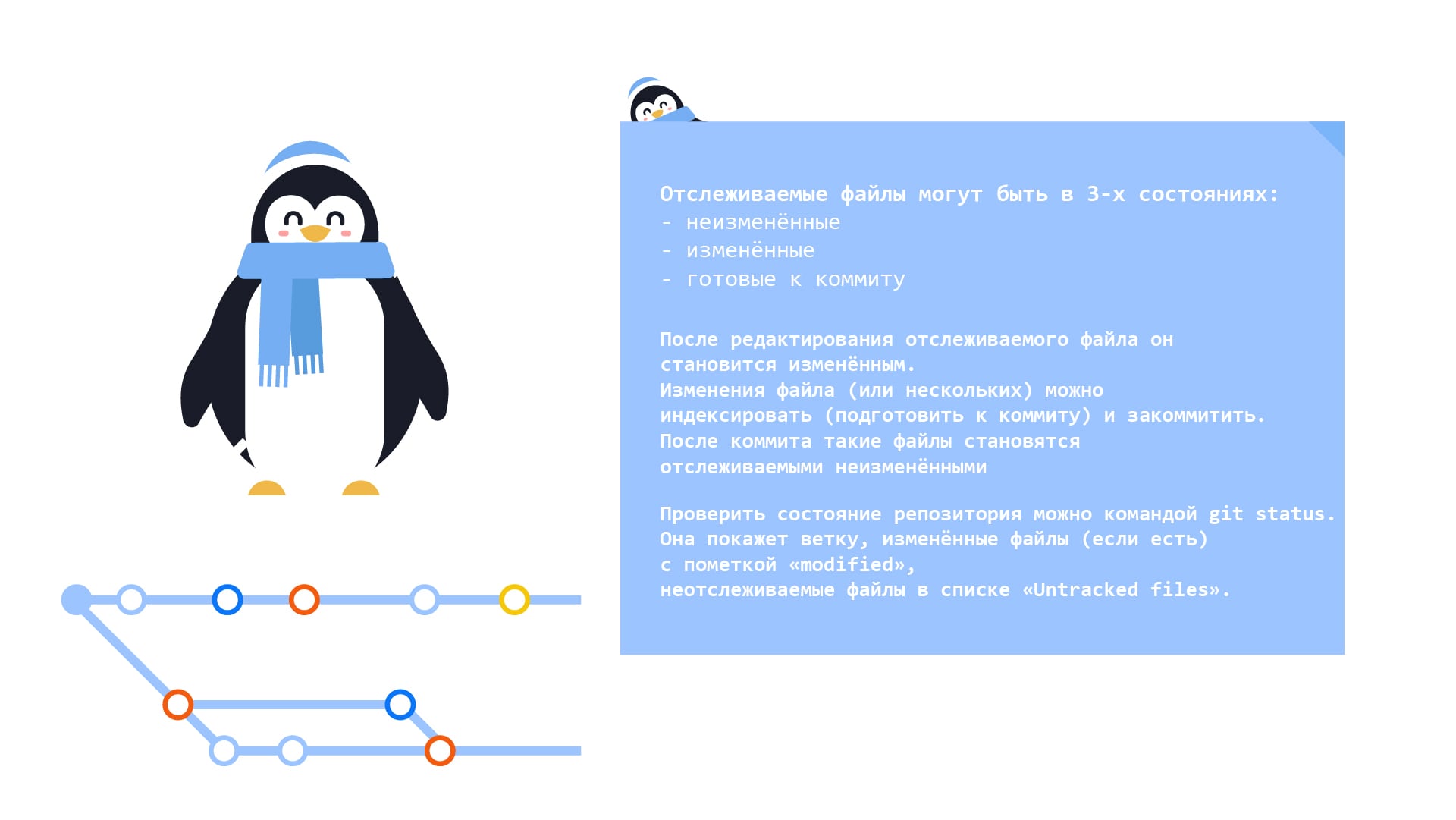
10.Как выполнить фиксацию изменений?

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.

**Коммит**– фиксация изменений, внесенных в индекс. **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?

**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**.



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

.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.**

**Пояснение структуры:**

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  
  
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.



**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