

Урок 4

Git, GitHub

Структура 4-го занятия

1. GIT введение
2. GIT основные команды
3. GIT создание репозитория
4. Работа с GitHub
5. Практические задания

Система контроля версий

Система контроля версий(СКВ, *VCS*, *Version Control Systems*) — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии.

Система контроля версий(СКВ) 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 и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить, так как вся история изменения сохраняется.

Проще говоря, это "машина времени" для программиста.

Очень похожий пример это компьютерные игры. В процессе игры мы можем сохранить перед каким-нибудь сложным уровнем и если вдруг что-то пойдёт не так, всегда можно вернуться к той точке, где мы сохранились.

Примечание: если смотрите презентацию с гугл диска(не скачивая), то возможны небольшие проблемы с вёрсткой некоторых слайдов.



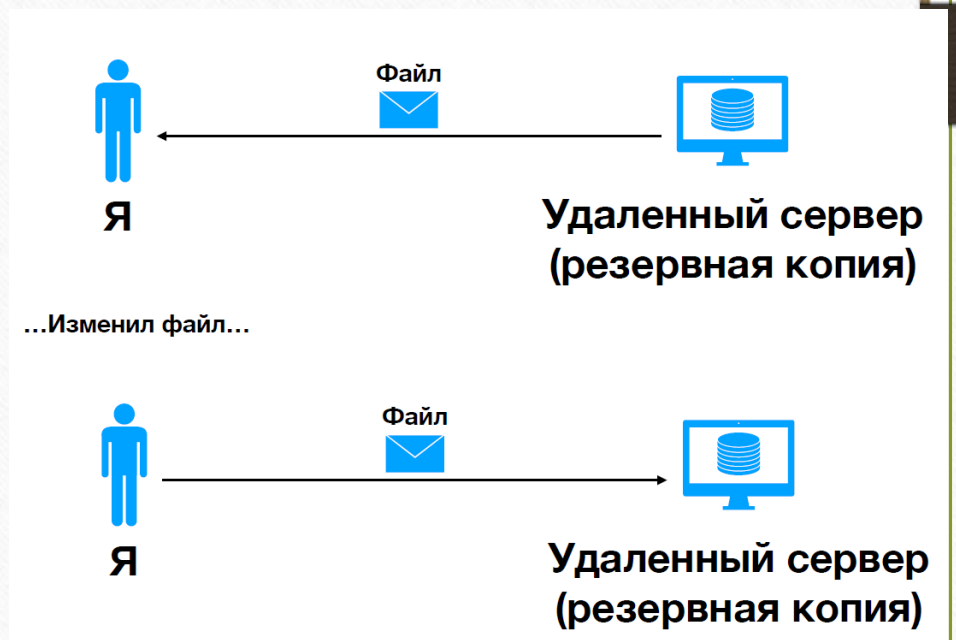
Зачем нужен Git?

Представим вариант разработки по старинке, без Git.

Есть сервер где хранится весь код. Он же будет резервной копией кода, так как если бы код хранился только на нашем компьютере и нигде больше — то это слишком рискованный вариант, ведь если с компьютером что-то произойдет, вся работа будет утеряна.

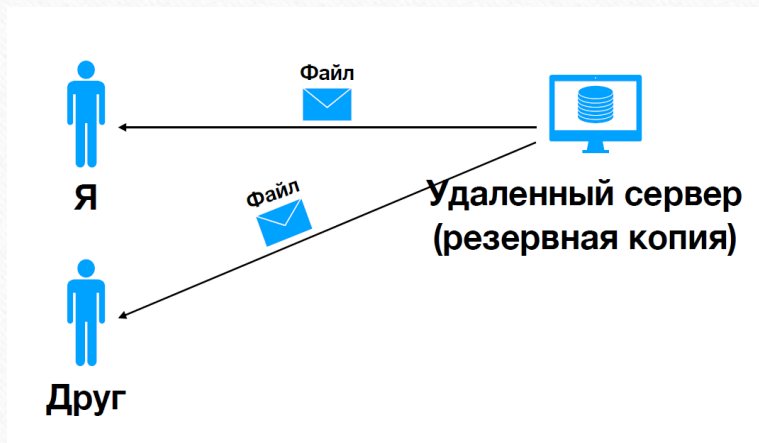
- Поэтому, каждый раз когда мы хотим поработать с кодом, нам нужно его сначала скачать с сервера в виде файла.
- Допустим, мы внесли какие-нибудь изменения в коде загруженного файла.
- Теперь мы должны обратно загрузить файл на сервер.

Это вполне рабочая схема если над проектом работаем только мы, но что, если вместе с нами работают ещё десятки других коллег?

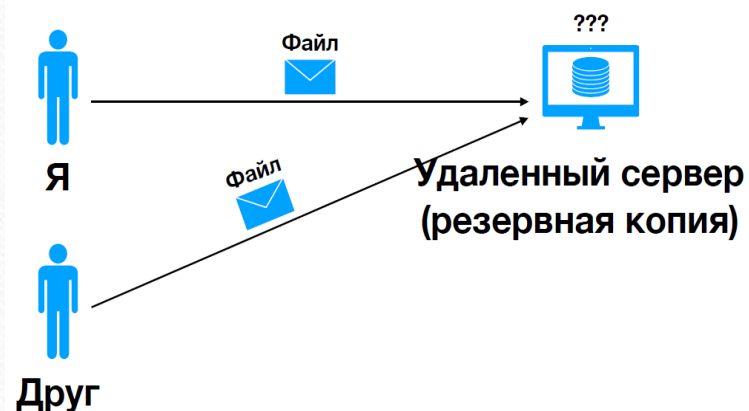


Зачем нужен Git?

Может возникнуть такая ситуация, когда коллега также скачает к себе на компьютер тот же самый файл что и мы.



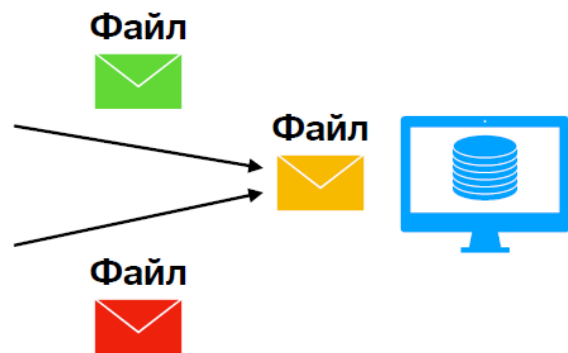
Далее мы оба вносим какие-то изменения в файл и хотим загрузить его обратно на удаленный сервер. В итоге изменения одного из нас (того, кто первый загрузит файл) будут потеряны. Так произойдет, так как последний загруженный файл просто перезапишет файл на сервере.



Зачем нужен Git?

Работать в таких условиях – очень неудобно и неэффективно.

И здесь, на помощь приходит Git, который берет на себя функциональность слияния разных версий файлов.



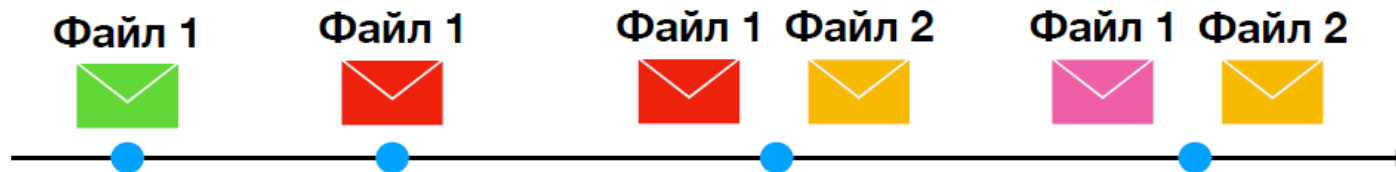
Например, если мы изменили файл одним образом, а наш коллега изменил его другим образом, и мы оба хотим загрузить этот файл на сервер, то Git, постарается слить два файла в один.

- Процесс слияния называется **merging**. Может производиться автоматически или в ручном режиме.
- Git попытается слить на сервер изменения из вашего файла и из файла коллеги в один файл на сервер.
- В итоге на сервере будет файл, в котором есть как наши изменения, так и изменения коллеги.

Но, иногда, в автоматическом режиме, так сделать не получится, так как могут возникнуть другие конфликты(которые разберём в дальнейшем), здесь тогда процесс слияния будет происходить в ручном режиме.(также, далее обсудим).

Зачем нужен Git?

Также, Git нужен потому что он является системой контроля версий. Когда Вы используете Git в проекте, у вас есть полная история изменения всех файлов, и если нужно, Вы можете вернуться к любой версии проекта.



- На рисунке выше изображена шкала времени слева направо
- Вначале был "Файл 1"
- Дальше "Файл 1" изменился(красный конверт)
- Спустя время создали новый "Файл 2"
- В финальной точке "Файл 1" ещё как-то поменялся

И все эти контрольные точки(версии) будут сохранены в Git. Можно будет в любой момент откатиться к любой из этих версий(или просто посмотреть что там было) и это очень удобно.

Это будет полезно, например, если нашли какой-то баг из-за которого система вообще перестала функционировать.

Есть много и других ситуаций, когда такая возможность будет полезна, об этом узнаем чуть позже.

Способы хранения данных

Говоря о способах хранения данных, можно выделить два способа: централизованный и распределенный.

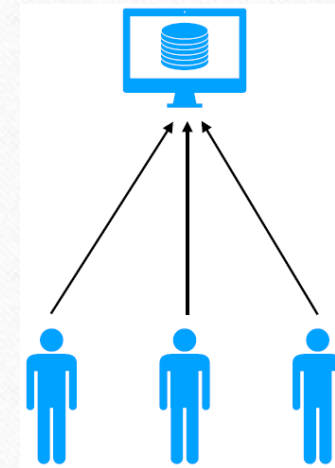
Централизованный - когда данные хранятся только в одном месте на центральном сервере, а все сотрудники сливают свои части кода в этот сервер.(рисунок 1)

- Весь проект находится только на центральном сервере. Такой подход имеет существенный недостаток - выход сервера из строя обернётся потерей всех данных.

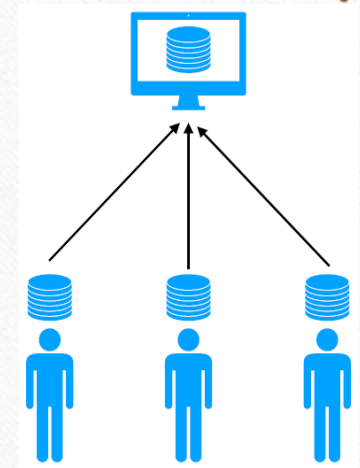
Распределенный - когда есть какая-то версия на центральном сервере, но помимо этого, у каждого разработчика есть полная копия это проекта у себя на компьютере(такая же копия как и на центральном сервере).

- И этот подход лучше, так как разработчики могут вносить изменения в код даже будучи в оффлайне. Можно внести изменения у себя на компьютере, а потом, подключившись к интернету, останется только слить эти изменения в центральный сервер.
- Также, здесь нет зависимости от центрального сервера. Если он сломается или его украдут или сгорит или что-то ещё, то у нас какая-то из версий проекта целиком останется у кого-нибудь из программистов, так как у каждого из них есть копия проекта.

GIT - распределенная система контроля версий.



Централизованный



Распределенный

Термины

- **Git или Гит** – система контроля и управления версиями файлов.
- **CVS (Control Version System), система контроля версий** – программное обеспечение, облегчающее работу с изменяющейся информацией. Позволяет, например, вернуться к определённой версии файла или определить, кем и когда были сделаны изменения в файле.
- **GitHub или Гитхаб** – веб-сервис для размещения репозитория и совместной разработки проектов.
- **Репозиторий Git** – каталог файловой системы, в котором находятся: файлы конфигурации, файлы журналов операций, выполняемых над репозиторием, индекс расположения файлов и хранилище, содержащее сами контролируемые файлы.
- **Локальный репозиторий** – репозиторий, расположенный на локальном компьютере разработчика в каталоге. Именно в нём происходит разработка и фиксация изменений, которые отправляются на удалённый репозиторий.
- **Удалённый репозиторий** – репозиторий, находящийся на удалённом сервере. Это общий репозиторий, в который приходят все изменения и из которого забираются все обновления.
- **Терминал, консоль, оболочка командной строки** – интерфейс командной строки.
- **Коммит (commit)** – фиксация состояния проекта в репозитории Git.
- **Индекс Git** – список файлов, изменения в которых система Git отслеживает. В индексе также предварительно сохраняются все изменённые файлы, которые будут зафиксированы в репозитории при следующем коммите.
- **Ветка Git** – изолированный независимый поток разработки, в котором можно делать коммиты, недоступные для других веток.

Git. Краткие итоги

Разберём ещё несколько возможностей и преимуществ Git и подытожим всё сказанное:

- Git – распределённая система контроля версий, которая в принципе может работать вообще без единого централизованного сервера. Это означает, что у каждого разработчика есть локальная копия общего репозитория (*хранилище для истории разработки проекта*), содержащая полную историю разработки проекта.
- При этом постоянное подключение к сети не требуется, поэтому система работает быстро, что выгодно отличает Git от централизованных CVS типа SVN (Subversion).
- При командной разработке репозитории разработчиков должны синхронизироваться (проект – общий и его состояние должно быть у всех одинаковым).
- Git позволяет при помощи определённых команд синхронизировать репозитории пользователей друг с другом, однако удобнее использовать один общий репозиторий на сервере.
- В этом случае все разработчики будут синхронизировать свои локальные репозитории Git с одним удалённым репозиторием. При этом не обязательно самостоятельно разворачивать и настраивать инфраструктуру для подобных репозиториях, можно воспользоваться бесплатными облачными ресурсами GitHub, GitLab или Bitbucket.

Основные преимущества Git, обеспечивающие ему большую популярность в мире CVS:

- Наличие удобных инструментов для работы с системой. Управлять Git можно с помощью консольных команд, также доступны различные графические утилиты (например, GitKraken, SmartGit, SourceTree) и расширения для популярных IDE (Visual Studio, PhpStorm, IntelliJ IDEA и пр.).
- Локальность операций. Для осуществления практически всех операций Git требуются только локальные файлы и ресурсы – это сильно ускоряет работу по сравнению с централизованными системами, обращающимися к ресурсам на сетевых серверах.
- Высокий уровень надёжности системы. Для всех файлов в репозитории хранятся их контрольные суммы, поэтому повреждения данных быстро обнаруживаются. В случае повреждения локального репозитория данные можно восстановить с централизованного сервера или из репозитория коллеги, работающего над тем же проектом.
- Возможность хранения в репозитории результатов работы по нескольким параллельно открытым задачам без влияния промежуточных результатов друг на друга. При этом окончательные результаты можно быстро скомбинировать в одну итоговую версию приложения.
- Существование и доступность бесплатных ресурсов GitHub и Bitbucket, на которых размещены тысячи репозиториях с программами на различных языках.

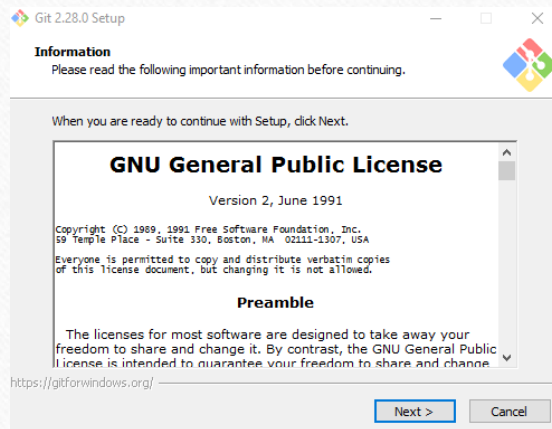
Установка Git на Windows (1/7)

1. Для начала нужно скачать Git. Для этого на [официальной странице](#) (загрузка подходящей версии начнётся автоматически), но если этого не произошло, выберите ручную версию для Windows (64-разрядная или 32-разрядная).

Downloading Git

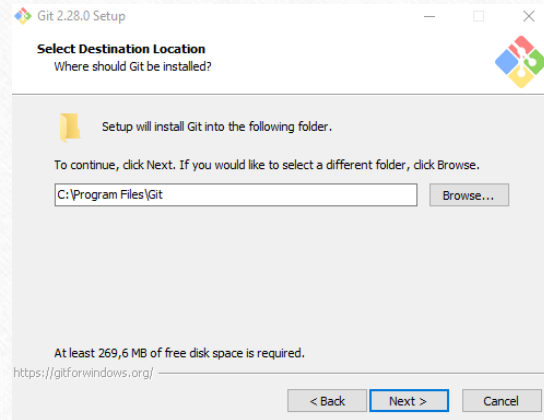


- Запустите файл и в появившемся окне нажмите "Next"

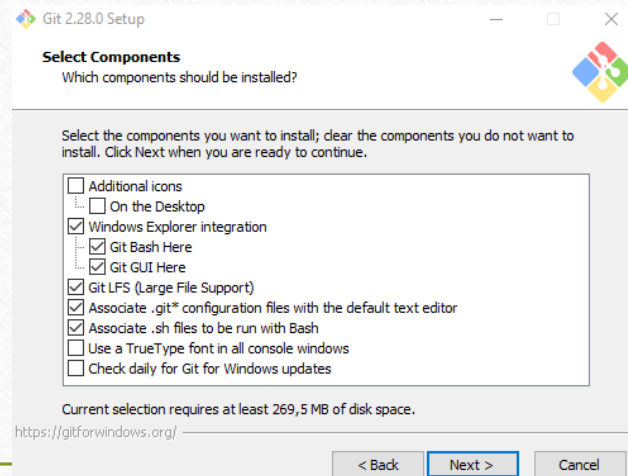


Установка Git на Windows (2/7)

2. Путь установки лучше оставить без изменений, нажмите "Next"

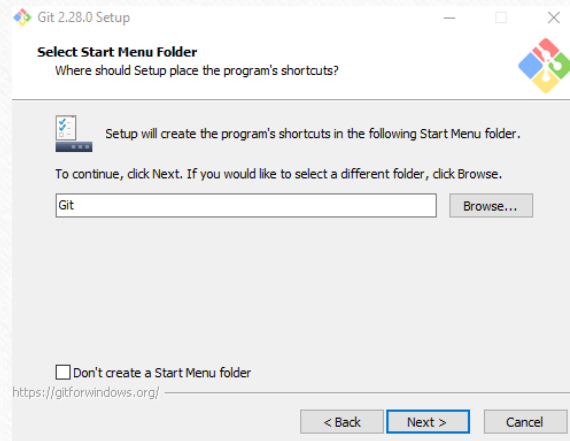


3. Здесь все компоненты оставляем по умолчанию, нажмите "Next"

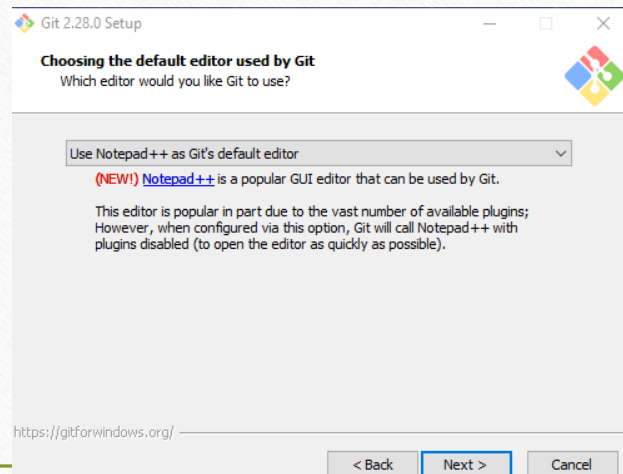


Установка Git на Windows (3/7)

4. Нажмите "Next"

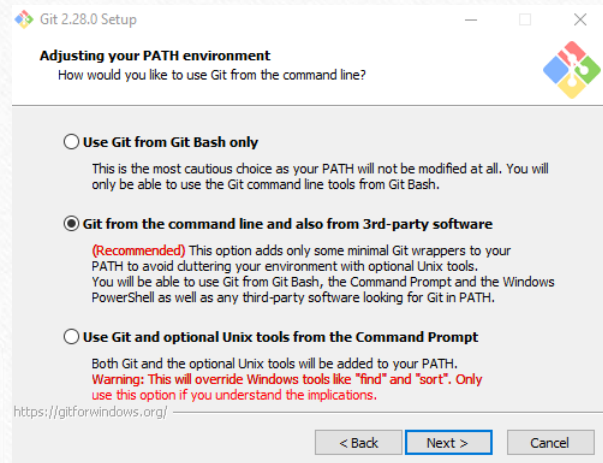


5. В этом окне нужно указать, какой текстовый редактор будет использоваться в Git по умолчанию. Можно выбрать любой удобный для Вас

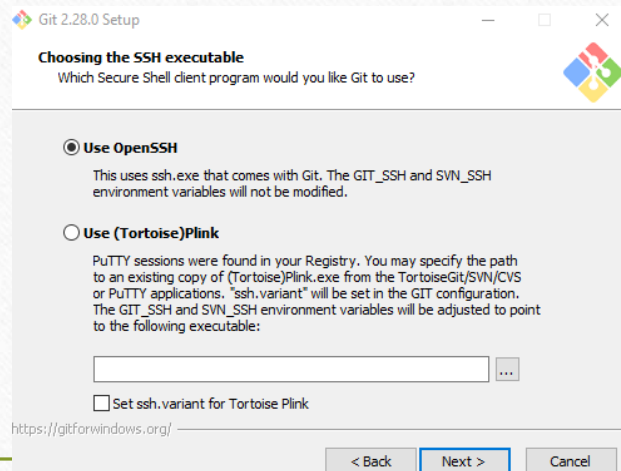


Установка Git на Windows (4/7)

6. Оставим предложенный по умолчанию **второй** вариант, нажмите "Next"

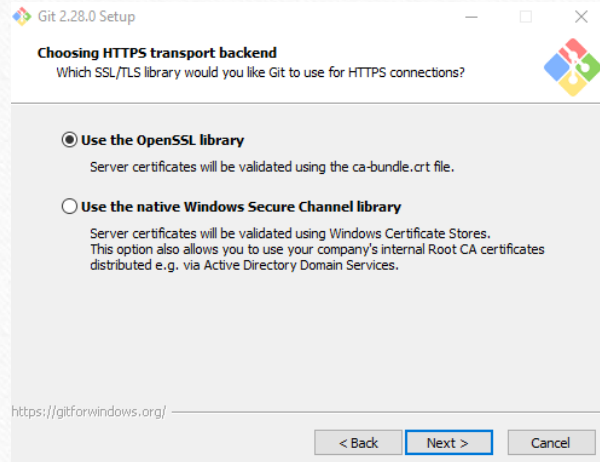


7. Оставим без изменений первый вариант (это выбор утилиты с помощью которой Git будет обращаться к сетевым ресурсам по протоколу SSH.)

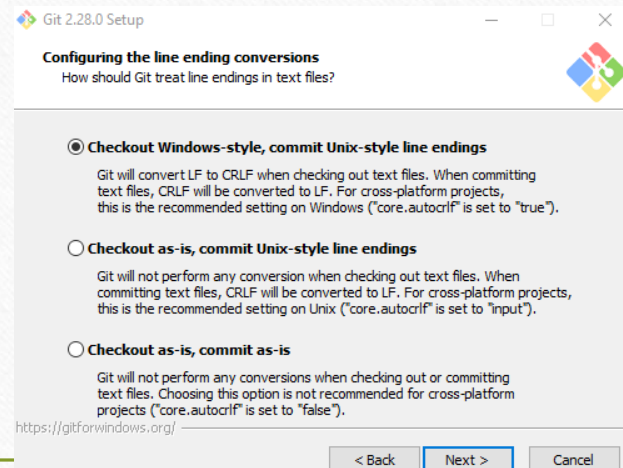


Установка Git на Windows (5/7)

8. Оставим предложенный по умолчанию первый вариант, нажмите "Next"

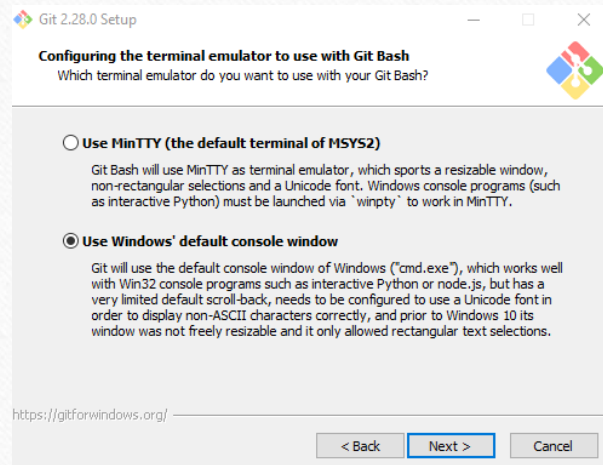


9. Оставим предложенный по умолчанию первый вариант, нажмите "Next"

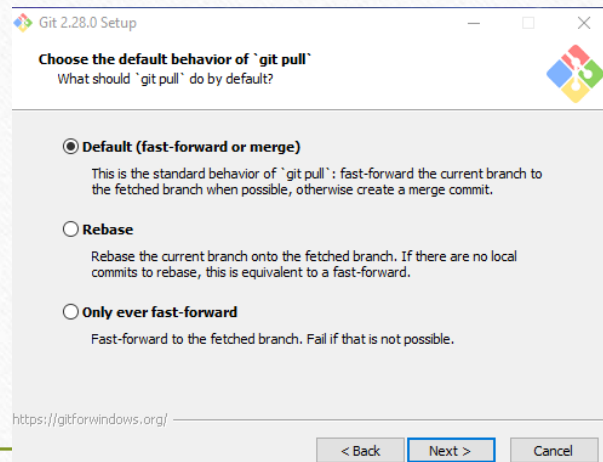


Установка Git на Windows (6/7)

10. Рекомендуем выбрать второй вариант (относится к Git Bash. Это специальная оболочка командной строки для запуска Git)



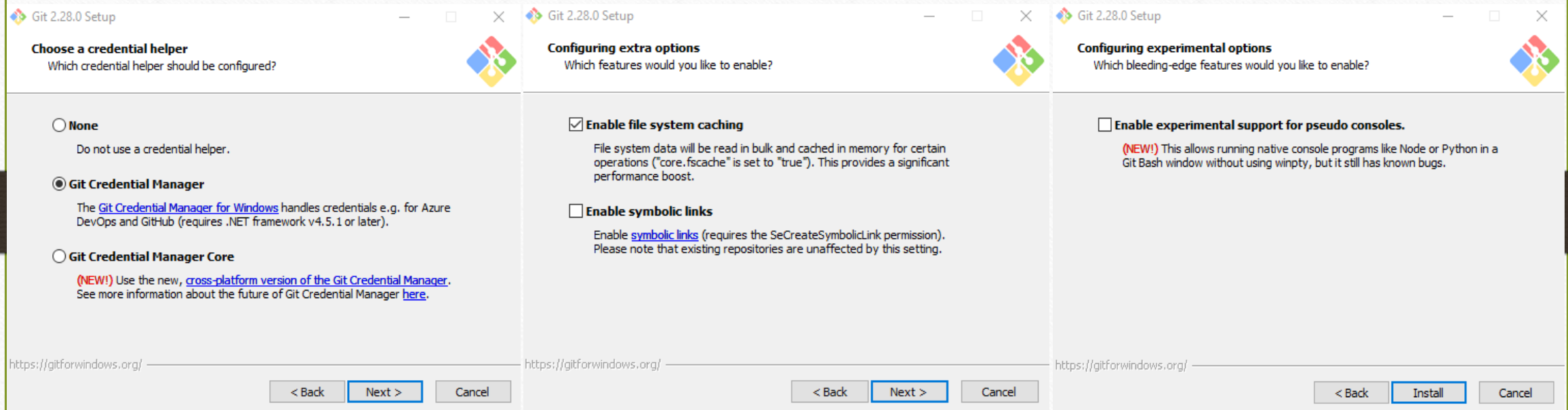
11. Далее можно оставить Default и нажать "Next"



Установка Git на Windows (7/7)

12. В оставшихся 3-х окнах можно оставить всё по умолчанию и нажимать "Next"

13. В последнем окне Нажмите "Install"



- Готово! Git уже на нашем компьютере, теперь можно приступать к работе с ним :)

Установка Git на Mac

1. Скачать Git можно на [официальной странице](#)

Download for macOS

There are several options for installing Git on macOS. Note that any non-source distributions are provided by third parties, and may not be up to date with the latest source release.

Homebrew

Install [homebrew](#) if you don't already have it, then:

```
$ brew install git
```

Xcode

Apple ships a binary package of Git with [Xcode](#).

Binary installer

Tim Harper provides an [installer](#) for Git. The latest version is [2.27.0](#), which was released 13 days ago, on 2020-07-22.

Building from Source

If you prefer to build from source, you can find tarballs [on kernel.org](#). The latest version is [2.28.0](#).

На выбор будет предложено несколько способов установки. Можно установить через homebrew(как было в примере с установкой python на 1-м уроке), через xcode или через установщик(binary installer).

В процессе установки достаточно нажимать "Next", каких-то особых настроек не потребуется.

Ещё можно установить через терминал, введя команду: **git --version** (установка начнётся автоматически)

Использование Git в консольном режиме

Хотя существует несколько удобных графических оболочек для управления Git(например GitKraken, SmartGit, SourceTree), мы будем учиться работать с Git в консольном режиме, вводя команды в терминале. Почему?

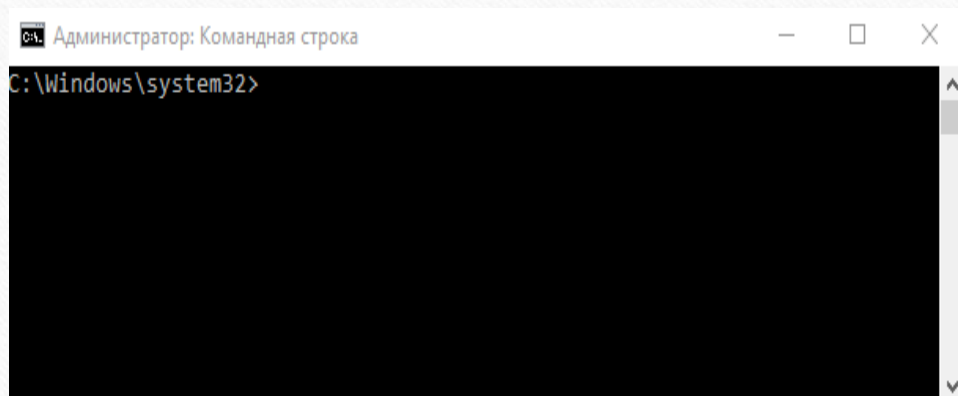
- Во-первых, именно командная строка даёт доступ ко всем возможностям Git.
- Во-вторых, непосредственное выполнение команд в терминале позволяет лучше понять логику и алгоритм работы системы.
- В-третьих, это очень широко распространённая практика.
- Наконец, освоив работу с Git из командной строки, Вы в дальнейшем сможете разобраться с интерфейсом любой графической утилиты для управления Git.

Оболочки командной строки в разных ОС

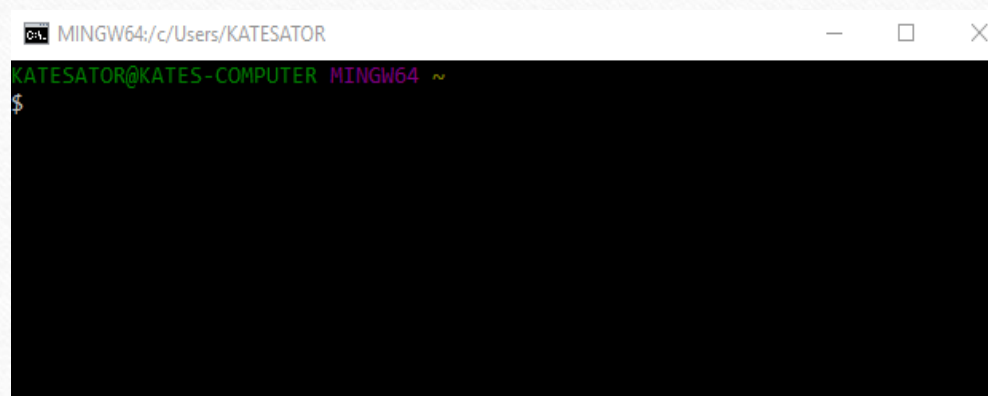
В Unix-подобных операционных системах Mac OS и Linux самыми популярными оболочками командной строки являются `bash` и `zsh`. В Windows используются две оболочки: стандартный командный интерпретатор `cmd.exe` и современная оболочка Windows Powershell.

Принцип работы у всех оболочек одинаков: вводим с клавиатуры команду, нажимаем `Enter`, команда выполняет заданные действия и выводит сообщение о результатах работы на экран.

Внешне, все оболочки также похожи друг на друга. Слева выводится приветствие командной строки (`command prompt`), в котором обычно написан путь к текущему каталогу и, возможно, другие дополнительные параметры (например, имя активного пользователя или компьютера, на котором мы работаем). Затем печатается символ окончания приветствия (обычно это `"$"` или `">"`), после которого размещается мигающий курсор - именно здесь вводятся команды.



Интерпретатор командной строки `cmd.exe` в Windows



Оболочка Git Bash

Консоль для Git в Windows

При установке Git в систему автоматически ставится специальная оболочка командной строки Git Bash, которая поддерживает команды интерпретатора bash и умеет дополнять команды Git. В приветствии командной строки этой оболочки выводятся имя пользователя, имя компьютера (для Git Bash это MINGW64), путь к текущему каталогу и текущая ветка Git (основная ветка проекта обычно называется master).

Ярлык для запуска **Git Bash** находится в группе Git меню "Пуск" (после открытия оболочки текущим будет каталог с профилем пользователя).



Начальная настройка

После установки на компьютере системы Git нужно настроить ее окружение. Делается это с помощью команды **git config**, позволяющей получать и устанавливать переменные конфигурации Git. Эти переменные бывают трех типов и хранятся в разных файлах.

- **Системные переменные**, действующие для всех пользователей системы и всех их репозиториях. Хранятся эти переменные в файле `/etc/gitconfig` (в Windows в файле `C:\Program Files\Git\etc\gitconfig`), работать с ними позволяет ключ **—system** команды **git config**.
- **Глобальные переменные**, влияющие на все репозитории конкретного пользователя. Хранятся эти переменные в файле `~/.gitconfig` (в Windows в файле `C:\Users\<USERNAME>\.gitconfig`), работать с ними позволяет ключ **--global** команды **git config**.
- **Локальные переменные**, хранящиеся в конкретном репозитории в файле `<project>/.git/config` и действующие только на этот репозиторий. Для работы с локальными переменными используется команда **git config** без дополнительных ключей.

Начальная настройка

Установка имени и почтового адреса пользователя:

Первое, что нужно сделать после установки Git – указать свое имя и адрес электронной почты. Эту информацию Git будет включать во все создаваемые вами коммиты и именно по этим реквизитам устанавливается авторство изменений в общем репозитории. Имя и адрес электронной почты удобнее сохранить в глобальные переменные с помощью следующих команд:

1. В меню пуск, найдите и запустите Git Bash
 2. Введите: **git config --global user.name "Ваше Имя Фамилия"** # лучше написать на английском
 3. Введите: **git config --global user.email "Ваша почта"**
- Посмотреть все установленные параметры Git можно с помощью ключа **--list**:
 - **git config --list** # можно проверить, что в списке отобразились данные из шага 2 и 3

Создание репозитория

Давайте создадим свой первый репозиторий. Для этого, также, откроем консоль, создадим новую папку `lesson_4_git`, в которой будем работать в дальнейшем, и перейдем в нее.

Введите в консоли Git Bash:

1. `mkdir lesson_4_git` # `mkdir` - создание новых директорий(создаём папку), через пробел указывается имя папки
2. `cd lesson_4_git` # `cd` - смена текущей директории, через пробел указывается новая директория для перехода
3. `git init` # инициализация системы Git для текущей папки и создание нового пустого репозитория

В результате увидим, что инициализирован пустой репозиторий Git в `/Users/имя пользователя/lesson_4_git/.git/`

В текущей папке будет создан каталог **.git**, в котором будут находиться все необходимые для Git файлы.

Важно: при каждом новом запуске консоли не забывайте сперва переходить в директорию репозитория.

- Для очистки содержимого в консоли можно ввести команду `clear` # горячие клавиши: Windows: **CTRL + I**; Mac: **CMD + k**

Несколько советов:

- Чтобы быстро повторить предыдущую команду, можно нажать на клавиатуре "**↑**"

- Для получения списка всех команд, можно ввести `git --help`

Добавление файлов в индекс

Создадим в папке `lesson_4_git` файл `"index.html"` со следующим содержимым:

```
<h1>Hello Git</h1>
```

Для этого, находясь в директории репозитория, введём в консоли:

1. `touch index.html` # создаём новый файл `index.html`
2. `notepad index.html` # открываем файл в блокноте (для `mac/linux` вместо `notepad` можно использовать `vi` или `start notepad++` или `nano`)
3. В блокноте введём `"<h1>Hello Git</h1>"`. После чего сохраним и закроем блокнот

Посмотрим статус файлов нашего проекта с помощью команды `git status`

4. `git status`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ touch index.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ notepad index.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.html

nothing added to commit but untracked files present (use "git add" to track)
```

```
$ git status
```

На ветке master

Пока нет коммитов

Неотслеживаемые файлы:

(используйте «`git add <файл>...`», чтобы добавить в то, что будет включено в коммит)
index.html

ничего не добавлено в коммит, но есть неотслеживаемые файлы (используйте «`git add`», чтобы отслеживать их)

Мы видим, что у нас пока нет файлов для фиксации состояния проекта (**коммита**), а за изменениями в файле `index.html` Git не следит. Нужно явно указать, за какими файлами Git должен наблюдать — для этого используется команда `git add`

Добавление файлов в индекс

Добавим файл для отслеживания:

git add index.html

Снова проверим статус:

git status

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git add index.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   index.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$
```

Теперь в нашем проекте есть новый файл **index.html**, который попадёт в репозиторий Git при следующем коммите.

Важно: обратите внимание, что нельзя сохранить состояние файла в репозитории сразу после его создания или изменения — файл необходимо предварительно индексировать (записать его в индекс), для этого используется команда **git add**

Добавление файлов в индекс

Создадим ещё один файл "second_file.html", состоящий из одной строки:

```
<h2>Second file text</h2>
```

Проверим статус:

`git status`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ touch second_file.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ notepad second_file.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    second_file.html
```

Сейчас в нашей папке два файла, Git следит за изменениями в **index.html**, а файл **second_file.html** не отслеживается, изменения в нем игнорируются.

Добавление файлов в индекс

Создадим ещё один файл "third_file.html", состоящий из одной строки:

```
<h3>Third file text</h3>
```

Проверим статус:

git status

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ touch third_file.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ notepad third_file.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        second_file.html
        third_file.html
```

Сейчас в нашей папке три файла, Git следит за изменениями в **index.html**, а файлы **second_file.html**, **third_file.html** не отслеживаются, изменения в них игнорируются. Изменим эту ситуацию:

Добавление файлов в индекс

Укажем Git следить за всеми файлами проекта. Для этого можно в команде **git add** перечислить имена всех неотслеживаемых файлов: **git add second_file.html third_file.html**

Второй более удобный вариант — указать в качестве параметра команды **git add** путь к текущей папке (точку): **git add .**

При этом в индекс добавятся все файлы из текущего каталога и всех его подкаталогов:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git add .

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html
        new file:   second_file.html
        new file:   third_file.html
```

Теперь Git следит за изменениями во всех трёх файлах.

Добавление файлов в индекс

Изменим теперь содержимое файла index.html:

```
<h1>Hello new Git!</h1>
```

Проверим статус:

git status

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ notepad index.html

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   index.html
    new file:   second_file.html
    new file:   third_file.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   index.html
```

Как видим, файл index.html представлен два раза: как индексированный файл (в следующий коммит этот файл попадёт как раз в данном состоянии) и как файл, последние изменения в котором не попадут в следующий коммит.

Добавление файлов в индекс

Создадим еще один файл `fourth.html` из одной строки:

```
<h4>Fourth file</h4>
```

Проверим статус:

`git status`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html
        new file:   second_file.html
        new file:   third_file.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        fourth.html
```

Здесь мы видим три типа записей о файлах и изменениях в них:

- Файлы, находящиеся в индексе Git и готовые к включению в коммит (имена файлов выделены зеленым цветом).
- Файлы, за которыми Git следит, но информация об их изменениях не будет добавлена в коммит, так как последние версии этих файлов еще не добавлены в индекс (файлы не проиндексированы).
- Файлы, за изменениями которых Git не следит.

```
$ git status
```

На ветке master

Пока нет коммитов

Изменения, которые будут включены в коммит:

(используйте «`git rm --cached <файл>...`», чтобы убрать из индекса)

новый файл: `index.html`

новый файл: `second_file.html`

новый файл: `third_file.html`

Изменения, которые не в индексе для коммита:

(используйте «`git add <файл>...`», чтобы добавить файл в индекс)

(используйте «`git restore -- <файл>...`», чтобы отменить изменения в рабочем каталоге)

изменено: `index.html`

Неотслеживаемые файлы:

(используйте «`git add <файл>...`», чтобы добавить в то, что будет включено в коммит)

`fourth.html`

Добавление файлов в индекс

Команда **git add** позволяет добавлять в индекс как новые файлы, так и те файлы, за которыми Git следит и которые изменились с момента последнего коммита. Фактически эта команда добавляет актуальную версию файла в снимок состояния (моментальную копию) нашего проекта, который мы хотим запомнить в репозитории Git.

Добавим в индекс Git все новые и модифицированные файлы из текущего каталога и посмотрим на статус:

```
git add .
```

```
git status
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git add .

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   fourth.html
    new file:   index.html
    new file:   second_file.html
    new file:   third_file.html
```

Все четыре файла добавлены в индекс Git – снимок состояния проекта, который можно теперь сохранить в репозитории (выполнить коммит).

Создание коммита

В предыдущих шагах мы подготовили снимок состояния системы, но еще не сохранили его в репозитории. Здесь можно провести аналогию с бумажной фотокарточкой и альбомом для фотографий — у нас на руках уже есть распечатанный снимок, но мы его еще не вставили в альбом. После того, как фотография (снимок состояния проекта) будет помещена в альбом (репозиторий Git), можно будет листать его и возвращаться к нужной нам фотографии (состоянию файлов проекта). Запись содержимого индекса в репозиторий называется фиксацией изменений или коммитом.

Создадим наш первый коммит командой **git commit**

1. Откройте консоль(перейдите в папку проекта, если Вы всё еще не там)
2. Введите **git commit** # **если будет возникать ошибка**, тогда введите **git commit -m 'Первый тестовый коммит'** (3 и 4 шаг пропускайте)

В результате выполнения этой команды откроется текстовый редактор, привязанный к Git (выбирали его на этапе установки) в котором нужно описать содержимое коммита. Описание коммита может занимать несколько строк, символ **#** является признаком комментария (такие комментарии генерируются автоматически и в репозитории Git не сохраняются).

3. Напишите в текстовом редакторе в самой первой строчке описание коммита, например: "Первый тестовый коммит"
 - Коммиты могут быть на русском или английском, в зависимости от принятой практики в компании
 - В коммитах принято писать понятные, осмысленные сообщения, чтобы другие разработчики тоже понимали что происходило в этом коммите
4. Сохраните и закройте текстовый редактор

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git commit
[master (root-commit) 30b5ca4] Первый тестовый коммит
4 files changed, 4 insertions(+)
create mode 100644 fourth.html
create mode 100644 index.html
create mode 100644 second_file.html
create mode 100644 third_file.html
```

В качестве результата команда **git commit** выводит семизначный шестнадцатеричный хэш (идентификатор) выполненного коммита (в нашем случае 30b5ca4), описание коммита и список файлов, вошедших в коммит.

Создание коммита

Проверим статус:

git status

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

\$ git status
На ветке master
нечего коммитить, нет изменений в рабочем каталоге

Сейчас команда **git status** не вывела информации о файлах — это означает, что содержимое рабочего каталога проекта, индекса Git и последнего зафиксированного коммита в репозитории Git соответствуют друг другу. В этом случае говорят, что проект имеет чистый статус.

Посмотреть историю коммитов можно с помощью команды **git log**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f (HEAD -> master)
Author: Be Tester
Date:   Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```

Здесь мы увидим полный хэш (идентификатор) коммита, его описание и информацию об авторе коммита и времени его создания.

Обратите внимание, что семизначный идентификатор коммита, выводимого после выполнения команды **git commit**, совпадает с первыми символами в полном идентификаторе. Git позволяет обращаться к коммитам как по длинному, так и по короткому идентификатору.

Создание коммита

Изменим содержимое файла fourth.html на:

```
<h4>Fourth file new text</h4>
```

Проверим статус:

git status

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   fourth.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Добавим изменения в индекс — раньше мы это делали с помощью команды **git add .**, а теперь воспользуемся ключом **-A** (добавить в индекс все файлы, как отслеживаемые, так и не отслеживаемые):

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git add -A

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   fourth.html
```

Создание коммита

Создадим новый коммит:

`git commit -m 'changed line commit'` # описание указывается в кавычках

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git commit -m 'changed line commit'
[master f917cba] changed line commit
1 file changed, 1 insertion(+), 1 deletion(-)
```

Ключ **-m** в команде **git commit** позволяет задавать описание коммита сразу, без формирования файла с описанием в текстовом редакторе. Такой вариант используется чаще, так как он удобнее и быстрее.

Важно: если при написании коммита случайно допустили ошибку(например, не поставили кавычки в комментарии), консоль может перестать реагировать на ввод с клавиатуры, в таком случае, используйте горячие клавиши **CTRL + C** для выхода из этого состояния.

Команда **git log** теперь покажет нам информацию о двух сделанных нами коммитах:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit f917cba786340d1b66eee68b98f89de1824f4413 (HEAD -> master)
Author: Be Tester
Date:   Wed Aug 5 19:59:20 2020 +0300

    changed line commit

commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f
Author: Be Tester
Date:   Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```


Создание коммита

Git позволяет увидеть, что именно менялось в файлах коммита. Для этого нужно скопировать идентификатор нужного коммита и выполнить команду:

git diff 30b5ca483c25bb9bf2fe9a6779053482dad8115f # у вас будет другой ключ, его можно скопировать из консоли(первый тестовый коммит)

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git diff 30b5ca483c25bb9bf2fe9a6779053482dad8115f
diff --git a/fourth.html b/fourth.html
index 09ba281..6eba8c4 100644
--- a/fourth.html
+++ b/fourth.html
@@ -1,1 @@
-<h4>Fourth file</h4>
\ No newline at end of file
+<h4>Fourth file new text</h4>
\ No newline at end of file
```

Если вызвать команду **git diff** без идентификатора коммита, то она покажет изменения в текущем коммите (аналогично команде **git status**, но в более расширенном варианте).

Для проверки изменим файл **index.html**, добавим в него ещё одно слово "new":

```
<h1>Hello new new Git!</h1>
```

git diff

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git diff
diff --git a/index.html b/index.html
index 29f481e..376b050 100644
--- a/index.html
+++ b/index.html
@@ -1,1 @@
-<h1>Hello new Git!</h1>
\ No newline at end of file
+<h1>Hello new new Git!</h1>
\ No newline at end of file
```

Здесь мы видим не только имя файла, в котором произошли изменения, как это было в команде **git status**, но и сами изменённые строки.

Создание коммита

Закоммитим все наши изменения:

`git add .`

`git commit -m 'третий коммит'`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git add .

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git commit -m 'третий коммит'
[master 0a578a0] третий коммит
1 file changed, 1 insertion(+), 1 deletion(-)
```

В репозитории теперь будут храниться три коммита:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 0a578a0a7f0952c1b6c01ef675c75b9d4ba338ed (HEAD -> master)
Author: Be Tester
Date:   Wed Aug 5 21:26:26 2020 +0300

    третий коммит

commit f917cba786340d1b66eee68b98f89de1824f4413
Author: Be Tester
Date:   Wed Aug 5 19:59:20 2020 +0300

    changed line commit

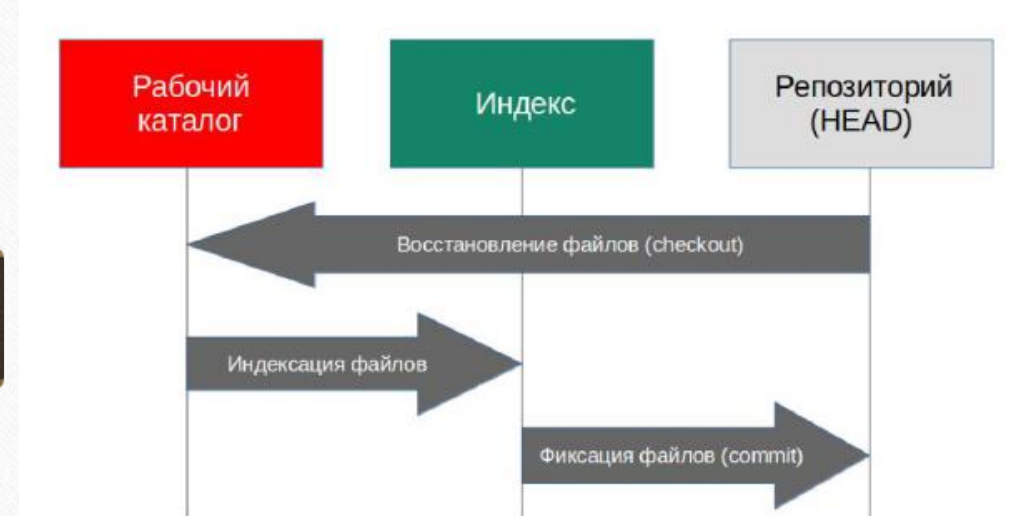
commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f
Author: Be Tester
Date:   Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```


Навигация по коммитам: отмена изменений

Рассмотрим, каким образом во время разработки можно отменять сохранённые в Git изменения файлов. Для понимания этой процедуры разберём, как Git работает с файлами проекта.

Каждый Git-проект состоит из трёх областей: **рабочий каталог**, **индекс** и **репозиторий**.

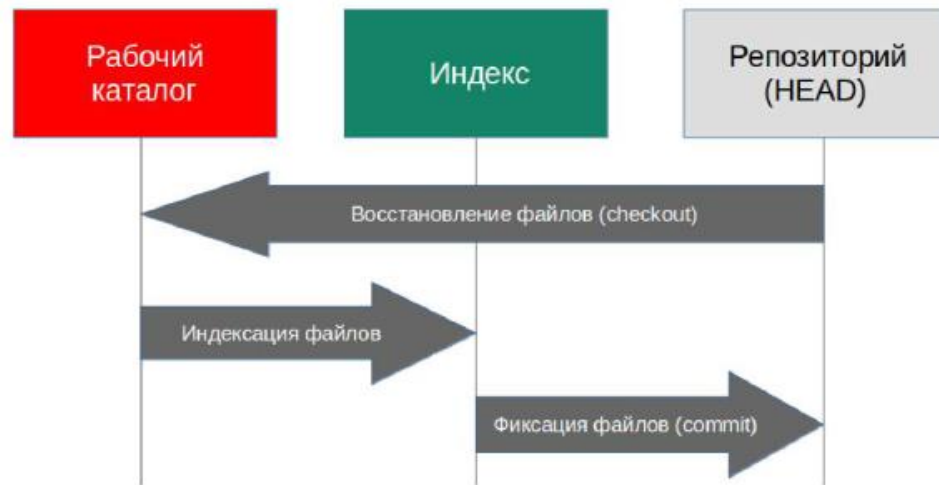


- **Рабочий каталог** - это место, куда выполняется выгрузка одной из версий проекта. Когда мы создаем новые файлы или изменяем уже существующие, то эти файлы сохраняются в рабочем каталоге. Рабочий каталог не имеет отношения к папке `.git`.
- **Область индексирования (индекс или промежуточная область)** находится в папке `.git` и содержит информацию о том, какие именно файлы и в каком состоянии будут зафиксированы при следующем коммите.
- **Репозиторий** - хранится в папке `.git` и содержит все выполненные коммиты, то есть снимки состояния файлов для каждого коммита. При этом на последний коммит ссылается специальный указатель `HEAD`.

Навигация по коммитам: отмена изменений

Обработка файлов в Git происходит следующим образом:

- Мы редактируем файл в рабочем каталоге. У такого файла будет состояние "Изменён" (modified).
- Индексируем изменённый файл, добавляя его в индекс командой **git add**. Статус такого файла – "Индексирован" (staged).
- Сохраняем изменения в репозитории, выполняя коммит с помощью **git commit**. Статус такого файла – "Зафиксирован" (fixed).



С помощью Git можно легко отменить изменения в не зафиксированном файле, вернув его в состояние последнего коммита, или откатить файл в состояние, в котором он был после выполнения определённого коммита.

Давайте проделаем эти процедуры для нашего проекта:

Навигация по коммитам: отмена изменений

Изменим содержимое файла index.html убрав одно слово "new":

```
<h1>Hello new Git!</h1>
```

Эти изменения мы проделали в рабочем каталоге, о чем и говорит команда **git status**:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Добавим изменённые файлы в индекс:

```
git add .
```

```
git status
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html
```

Навигация по коммитам: отмена изменений

Теперь файл `index.html` можно закоммитить:

`git commit -m 'index changed'`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git commit -m 'index changed'
[master 2833b36] index changed
1 file changed, 1 insertion(+), 1 deletion(-)
```

`git log`

Указатель **HEAD** сейчас ссылается на ЭТОТ КОММИТ:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 2833b36111eae1f022c083e861def4cd4e73a3e0 (HEAD -> master)
Author: Be Tester
Date: Thu Aug 6 11:50:00 2020 +0300

    index changed

commit 0a578a0a7f0952c1b6c01ef675c75b9d4ba338ed
Author: Be Tester
Date: Wed Aug 5 21:26:26 2020 +0300

    третий коммит
```

Из текущего коммита, на который ссылается **HEAD**, можно вернуть файл в рабочий каталог с помощью команды `git checkout`. Попробуем сделать это:

Навигация по коммитам: отмена изменений

Проверим сначала содержимое файла index.html:

```
cat index.html      # cat отображает содержимое файла
```

```
<h1>Hello new Git!</h1>
```

Внесем изменения в файл second_file.html

```
<h2>Changed for commit</h2>
```

Выполним коммит с этим файлом:

```
git add .
```

```
git commit -m 'second_file changed'
```

Убедимся, что последний коммит попал в историю и указатель HEAD переместился на него: git log

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git commit -m 'second_file changed'
[master 0d68487] second_file changed
1 file changed, 1 insertion(+), 1 deletion(-)

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 0d684879c1db84deb3d1f0b8316255c223d0bbbb (HEAD -> master)
Author: Be Tester
Date: Thu Aug 6 12:10:05 2020 +0300

    second_file changed

commit 2833b36111eae1f022c083e861def4cd4e73a3e0
Author: Be Tester
Date: Thu Aug 6 11:50:00 2020 +0300

    index changed
```

если список изменений будет большим
и в конце отобразится (END), то для того
чтобы выйти из этого режима нужно нажать на 'Q'

Давайте откатим состояние проекта к третьему коммиту(на следующем слайде сделаем), где в файле index.html был записан заголовок "Hello new new Git!". Делается это с помощью команды **git reset**, которая поддерживает два возможных режима: мягкий (ключ **--soft**) и жесткий (ключ **--hard**).

Мягкий reset, замена и объединение коммитов

Команда **git reset --soft коммит** переместит указатель HEAD на нужный коммит, при этом рабочий каталог и индекс остаются прежними (соответствующими коммиту, с которого мы ушли).

Задание:

1. Создайте в нашем проекте еще один файл "fifth.html", в котором будет записана строка "this is text in fifth file".
2. Добавьте все файлы в индекс
3. Проверьте статус
4. Закоммитьте изменения в проекте с сообщением 'added fifth file'

Посмотрим, как теперь выглядит история коммитов нашего проекта:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 4cbb668b19987d963cf0bbccc637317e9795025b (HEAD -> master)
Author: Be Tester
Date: Thu Aug 6 12:25:56 2020 +0300

    added fifth file

commit 0d684878c1db84deb3d1f0b8316255c223d0bbbbb
Author: Be Tester
Date: Thu Aug 6 12:10:05 2020 +0300

    second_file changed

commit 2833b36111eae1f022c083e861def4cd4e73a3e0
Author: Be Tester
Date: Thu Aug 6 11:50:00 2020 +0300

    index changed

commit 0a578a0a7f0952c1b6c01ef675c75b9d4ba338ed
Author: Be Tester
Date: Wed Aug 5 21:26:26 2020 +0300

    третий коммит
```

Первый тестовый коммит
(END)

если список изменений будет большим и в конце отобразится (END), то для того чтобы выйти из этого режима нужно нажать на 'Q' Если ошиблись с вводом коммита, тогда используйте CTRL + C

Откатимся к третьему коммиту в мягком режиме. Для этого скопируем идентификатор этого коммита (достаточно первых четырех или более символов) и выполним команду:

git reset --soft 0a57 # у вас будет другой ключ для третьего коммита

Мягкий reset, замена и объединение коммитов

Указатель HEAD теперь переместился на третий коммит: `git log`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 0a578a0a7f0952c1b6c01ef675c75b9d4ba338ed (HEAD -> master)
Author: Be Tester
Date:   Wed Aug 5 21:26:26 2020 +0300

    третий коммит

commit f917cba786340d1b66eee68b98f89de1824f4413
Author: Be Tester
Date:   Wed Aug 5 19:59:20 2020 +0300

    changed line commit
```

Посмотрим статус проекта и содержимое рабочего каталога: `git status ; ls -la` # отобразит содержимое каталога

Как видим, в рабочем каталоге и в индексе проекта остались все изменения, которые были сделаны после третьего коммита, в том числе и новый файл `fifth.html`.

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   fifth.html
    modified:   index.html
    modified:   second_file.html
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ ls -la
total 41
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 12:24 ./
drwxr-xr-x 1 KATESATOR 197121  0 авг  5 23:48 ../
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 12:59 .git/
-rw-r--r-- 1 KATESATOR 197121 26 авг  6 12:25 fifth.html
-rw-r--r-- 1 KATESATOR 197121 29 авг  5 19:52 fourth.html
-rw-r--r-- 1 KATESATOR 197121 24 авг  6 11:44 index.html
-rw-r--r-- 1 KATESATOR 197121 27 авг  6 12:09 second_file.html
-rw-r--r-- 1 KATESATOR 197121 24 авг  5 14:01 third_file.html
```

Убедимся, что текущее содержимое файлов `index.html` и `second_file.html` не изменилось:

```
cat index.html          # <h1>Hello new Git!</h1>
```

```
cat second_file.html    # <h2>Changed for commit</h2>
```

Мягкий reset, замена и объединение коммитов

При необходимости можно откатить состояние нужных нам файлов к тому, что было сохранено в третьем коммите. Для этого нужно сначала убрать эти файлы из индекса, а затем восстановить их содержимое в рабочем каталоге из репозитория.

Уберем index.html из индекса:

```
git reset HEAD index.html
```

При этом содержимое файла в рабочем каталоге пока не изменилось:

```
cat index.html
```

```
<h1>Hello new Git!</h1>
```

Как теперь вернуть **index.html** к состоянию, в котором он был зафиксирован при выполнении третьего коммита, к которому мы откатились? С помощью **checkout**

```
git checkout -- index.html # обратите внимание пробел между '--' и 'index.html'
```

```
cat index.html
```

```
<h1>Hello new new Git!</h1>
```

Задача выполнена – мы вернули файл index.html в то состояние, которое было зафиксировано в третьем коммите. Проверим статус проекта:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   fifth.html
    modified:   second_file.html
```

Как видим, файл **index.html** не упоминается, так как его состояние в рабочем каталоге соответствует состоянию в репозитории. Файлы second_file.html и fifth.html при этом соответствуют всем изменениям, которые мы проводили над ними после третьего коммита.

Мягкий reset, замена и объединение коммитов

Добавим все изменения в индекс и создадим новый коммит:

`git add .`

`git commit -m 'rejected commit'`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git commit -m 'rejected commit'
[master f32cd56] rejected commit
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 fifth.html
```

История проекта теперь изменилась: коммиты, сделанные после третьего, пропали, а сразу после третьего коммита находится только что выполненный коммит:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit f32cd5635937b41c4b2aeb61dca1b662cbc6f65b (HEAD -> master)
Author: Be Tester
Date: Thu Aug 6 13:41:06 2020 +0300

    rejected commit

commit 0a578a0a7f0952c1b6c01ef675c75b9d4ba338ed
Author: Be Tester
Date: Wed Aug 5 21:26:26 2020 +0300

    третий коммит

commit f917cba786340d1b66eee68b98f89de1824f4413
Author: Be Tester
Date: Wed Aug 5 19:59:20 2020 +0300

    changed line commit

commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f
Author: Be Tester
Date: Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```

Для отката изменений в проекте рекомендуется использовать мягкий режим(`--soft`) команды **reset**, однако есть и другой, более радикальный, способ – жесткий режим команды **reset**.

Жёсткий reset, отмена коммитов

Команда **git reset --hard** позволяет полностью отменить все изменения, которые были сделаны после определённого коммита.

Посмотрим, какие файлы содержатся сейчас в рабочем каталоге нашего проекта: **ls -la**

Как видим, у нас есть файл **fifth.html**, который мы создали и зафиксировали при выполнении пятого коммита. Этот файл остался после отката к третьему коммиту с помощью мягкого **reset**.

Содержимое файла **second_file.html** также соответствует состоянию, в котором он был при выполнении пятого коммита:

```
cat second_file.html # <h2>Changed for commit</h2>
```

Давайте вернёмся к состоянию проекта после выполнения второго коммита: **git log**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit f32cd5635937b41c4b2aeb61dca1b662cbc6f65b (HEAD -> master)
Author: Be Tester <sigmarus47@mail.ru>
Date: Thu Aug 6 13:41:06 2020 +0300

    rejected commit

commit 0a578a0a7f0952c1b6c01ef675c75b9d4ba338ed
Author: Be Tester <sigmarus47@mail.ru>
Date: Wed Aug 5 21:26:26 2020 +0300

    третий коммит

commit f917cba786340d1b66eee68b98f89de1824f4413
Author: Be Tester <sigmarus47@mail.ru>
Date: Wed Aug 5 19:59:20 2020 +0300

    changed line commit

commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f
Author: Be Tester <sigmarus47@mail.ru>
Date: Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```

Для этого скопируем идентификатор второго коммита (напомним, что допускается копировать только первые четыре или больше символов) и выполним команду:

```
git reset --hard f917c # у вас будет другой ключ для второго коммита(смотрим снизу вверх)
```


Жёсткий reset, отмена коммитов

Посмотрим, какие файлы сейчас есть в проекте: `ls -la`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ ls -la
total 40
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 14:09 ./
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 15:52 ../
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 18:59 .git/
-rw-r--r-- 1 KATESATOR 197121 29 авг  5 19:52 fourth.html
-rw-r--r-- 1 KATESATOR 197121 24 авг  6 14:09 index.html
-rw-r--r-- 1 KATESATOR 197121 25 авг  6 14:09 second_file.html
-rw-r--r-- 1 KATESATOR 197121 24 авг  5 14:01 third_file.html
```

Как видим, файл **fifth.html** отсутствует, он был удален. Файл **second_file.html** вернулся к состоянию второго коммита:

`cat second_file.html` # <h2>Second file text</h2>

Проверим статус файлов: `git status`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Таким образом, файлы проекта вернулись в то состояние, в котором они были после выполнения второго коммита. Все изменения, которые производились над файлами после второго коммита, оказались потеряны, восстановить их нельзя. История фиксации изменений теперь выглядит так, как будто мы выполняли только два коммита.

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit f917cba786340d1b66eee68b98f89de1824f4413 (HEAD -> master)
Author: Be Tester
Date:   Wed Aug 5 19:59:20 2020 +0300

    changed line commit

commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f
Author: Be Tester
Date:   Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```

Итак, результат выполнения команды `git reset --hard` нельзя отменить, поэтому жёсткий режим команды `reset` нужно использовать с крайней осторожностью только в исключительных случаях (особенно если разработка ведется не индивидуально, а в команде).

Когда нужно делать коммиты

Обсудим, как обычно выглядит рабочий цикл при работе с Git.

Коммит – это целостное осмысленное изменение проекта. С одной стороны, коммиты должны быть небольшими, затрагивающими по возможности минимальное число файлов. С другой стороны, нет смысла создавать коммит после редактирования каждой строки – это приведёт к разрастанию истории изменений, в ней будет сложно разобраться и найти нужный коммит.

Нужно стараться выполнять коммиты так, чтобы по их истории было понятно, какие новые возможности добавлялись или какие ошибки исправлялись.

Продолжим работу над нашим проектом. Предположим, что нам нужно добавить функционал в файл **index.html**, например, сделать в этом файле разметку для бокового меню.

Откроем **index.html** в редакторе кода и внесем нужные изменения:

1. Выполним: **notepad index.html** # или можно открыть в другом редакторе
2. Вставьте код в **index.html** (код на следующем слайде) сохраните в редакторе и закройте его

Когда нужно делать коммиты

```
<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8">

    <title>Project</title>

  </head>

  <body><!-- code -->


    <div class="sidebar">

      <div class="sidebar__title">Боковое меню</div>

      <ul class="sidebar__list">

        <li class="sidebar__item"></li>

        <li class="sidebar__item"></li>

        <li class="sidebar__item"></li>

        <li class="sidebar__item"></li>

        <li class="sidebar__item"></li>

      </ul>

    </div>


  </body>

</html>
```

Когда нужно делать КОММИТЫ

Коммит будем делать тогда, когда нужный функционал будет полностью реализован, т.е. весь код для формирования бокового меню в файл внесен.

Проверим статус проекта, чтобы убедиться, что в коммит не попадет ничего лишнего: **git status**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Добавляем файлы в индекс и выполним коммит:

git add .

git commit

В результате откроется текстовый редактор, в котором нужно написать комментарий для коммита.

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git add .

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git commit
hint: Waiting for your editor to close the file...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
1
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 #
5 # On branch master
6 # Changes to be committed:
7 #   modified:   index.html
8 #
9
```


Комментарии к коммитам

Комментарий к коммиту может быть как лаконичным, состоящим из одной строки, так и развернутым. Если в комментарии необходимо написать несколько строк, то принято на первой строке помещать короткий заголовок из одной фразы, а остальной комментарий записывать после одной пустой строки.

Введем в качестве комментария к нашему коммиту следующий текст:

"Добавил боковое меню

Lorem Ipsum – это текст-"рыба", часто используемый в печати и вэб-дизайне.

Lorem Ipsum является стандартной "рыбой" для текстов на латинице с XVI века.

В то время некий безымянный печатник создал большую коллекцию размеров и форм шрифтов, используя Lorem Ipsum для распечатки образцов.

Lorem Ipsum не только успешно пережил без заметных изменений пять веков, но и перешагнул в электронный дизайн."

Сохраним изменения в редакторе и закроем его – выполнение коммита завершено. Посмотрим историю коммитов:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 48717a75a7bd1a0331ccb6c780236c6cad768109 (HEAD -> master)
Author: Be Tester
Date: Thu Aug 6 22:55:03 2020 +0300

    Добавил боковое меню

    Lorem Ipsum - это текст-"рыба", часто используемый в печати и вэб-дизайне.
    Lorem Ipsum является стандартной "рыбой" для текстов на латинице с XVI века.
    В то время некий безымянный печатник создал большую коллекцию размеров и форм шрифтов, используя Lorem Ipsum для распечатки образцов.
    Lorem Ipsum не только успешно пережил без заметных изменений пять веков, но и перешагнул в электронный дизайн.

commit f917cba786340d1b66eee68b98f89de1824f4413
Author: Be Tester
Date: Wed Aug 5 19:59:20 2020 +0300

    changed line commit
```

Для чего нужны ветки в проекте

Рассмотрим, как в проекты добавляется новый функционал и как это связано с ветвлениями в проектах.

Проверим с помощью команды **git branch**, какие ветки уже есть в нашем проекте:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git branch
* master
```

Как видим, у нас есть одна основная ветка с именем **master**. В принципе, ветки при разработке можно называть как угодно, но **master** — это общепринятое имя для основной ветки проекта, в которую добавляется функционал, разработанный в других ветках.

В самом проекте у нас сейчас четыре файла:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ ls -la
total 40
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 14:09 ./
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 15:52 ../
drwxr-xr-x 1 KATESATOR 197121  0 авг  6 23:01 .git/
-rw-r--r-- 1 KATESATOR 197121 29 авг  5 19:52 fourth.html
-rw-r--r-- 1 KATESATOR 197121 492 авг  6 22:31 index.html
-rw-r--r-- 1 KATESATOR 197121 25 авг  6 14:09 second_file.html
-rw-r--r-- 1 KATESATOR 197121 24 авг  5 14:01 third_file.html
```

Представим, что наш проект уже работает в промышленном режиме на удалённом сервере, обеспечивая, например, продажу товаров через Интернет. Теперь возникла задача добавить к сервису новый функционал (например, реализовать корзину для покупаемых товаров). Это нужно делать независимо, не затрагивая основной проект, чтобы пользователи сервиса не видели незаконченных изменений, происходящих с проектом во время разработки.

Поэтому новый функционал разрабатывают в новых ветках (они называются тематическими), изолированно от основной ветки проекта. Когда все нужные изменения сделаны, функционал полностью готов и протестирован, тематическая ветка объединяется (сливается) с основной и новая версия (релиз) проекта выкладывается на удалённый сервер.

Создание и переключение веток

Для создания новой ветки и перехода в нее используется команда **git checkout -b имя_ветки**.

Имя для ветки нужно выбрать осмысленно, чтобы было понятно ее назначение (реализация нового функционала или исправление выявленных ошибок).

Нам нужно сделать корзину для покупок, поэтому новую ветку мы назовём "feature-cart":

git checkout -b feature-cart

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git checkout -b feature-cart
Switched to a new branch 'feature-cart'
```

Переключено на новую ветку 'feature-cart'

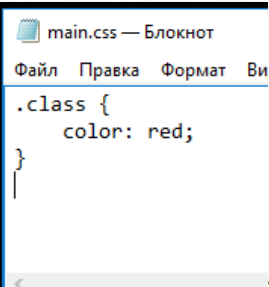
Команда **git branch** теперь говорит, что в проекте есть две ветки и мы находимся в ветке feature-cart:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ git branch
* feature-cart
master
```

Создадим в новой ветке файл **cart.html**, содержащий строку "Markup for cart", и файл **main.css** со стилями:

```
.class {
    color: red;
}
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ touch cart.html
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ notepad cart.html
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ touch main.css
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ notepad main.css
```



main.css — Блокнот

Файл Правка Формат Ви

```
.class {
    color: red;
}
```

Создание и переключение веток

При создании новой ветки в неё копируются все файлы, зафиксированные в последнем коммите. Поэтому теперь в нашем проекте есть все файлы, которые были в ветке master, и два только что созданных файла: **ls -la**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ ls -la
total 42
drwxr-xr-x 1 KATESATOR 197121  0 авг 7 12:16 ./
drwxr-xr-x 1 KATESATOR 197121  0 авг 6 15:52 ../
drwxr-xr-x 1 KATESATOR 197121  0 авг 7 12:04 .git/
-rw-r--r-- 1 KATESATOR 197121 15 авг 7 12:16 cart.html
-rw-r--r-- 1 KATESATOR 197121 29 авг 5 19:52 fourth.html
-rw-r--r-- 1 KATESATOR 197121 492 авг 6 22:31 index.html
-rw-r--r-- 1 KATESATOR 197121 28 авг 7 12:18 main.css
-rw-r--r-- 1 KATESATOR 197121 25 авг 6 14:09 second_file.html
-rw-r--r-- 1 KATESATOR 197121 24 авг 5 14:01 third_file.html
```

Проверим статус файлов: **git status**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ git status
On branch feature-cart
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    cart.html
    main.css

nothing added to commit but untracked files present (use "git add" to track)
```

Добавим новые файлы в индекс и создадим коммит (считаем, что работа над корзиной завершена):

git add .

git commit -m 'cart completed'

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ git add .

KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ git commit -m 'cart completed'
[feature-cart 7e4a79f] cart completed
2 files changed, 4 insertions(+)
create mode 100644 cart.html
create mode 100644 main.css
```


Создание и переключение веток

Посмотрим историю коммитов: `git log`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ git log
commit 7e4a79f1921c80f06f9783f2074b8880623f4ff1 (HEAD -> feature-cart)
Author: Be Tester
Date:   Fri Aug 7 12:23:10 2020 +0300

    cart completed

commit 48717a75a7bd1a0331ccb6c780236c6cad768109 (master)
Author: Be Tester
Date:   Thu Aug 6 22:55:03 2020 +0300

    Добавил боковое меню

    Lorem Ipsum - это текст-"рыба", часто используемый в печати и веб-дизайне.
    Lorem Ipsum является стандартной "рыбой" для текстов на латинице с XVI века.
    В то время некий безымянный печатник создал большую коллекцию размеров и форм шрифтов, используя Lorem Ipsum для распечатки образцов.
    Lorem Ipsum не только успешно пережил без заметных изменений пять веков, но и перешагнул в электронный дизайн.

commit f917cba786340d1b66eee68b98f89de1824f4413
Author: Be Tester
Date:   Wed Aug 5 19:59:20 2020 +0300

    changed line commit

commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f
Author: Be Tester
Date:   Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```

Как видим в истории есть все коммиты, которые были в проекте в ветке master до момента ветвления. Также Git показывает, какой коммит был последним в ветке master.

Вернемся теперь в ветку master. Переключаться между ветками можно с помощью команды `git checkout`:

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ git checkout master
Switched to branch 'master'
```

Проверим что переход произошёл `git branch`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git branch
  feature-cart
* master
```

Создание и переключение веток

Посмотрим историю коммитов и убедимся, что внутри ветки `master` ничего не известно о том коммите, который мы сделали на ветке `feature-cart`: `git log`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 48717a75a7bd1a0331ccb6c780236c6cad768109 (HEAD -> master)
Author: Be Tester
Date: Thu Aug 6 22:55:03 2020 +0300

    Добавил боковое меню

    Lorem Ipsum – это текст-"рыба", часто используемый в печати и веб-дизайне.
    Lorem Ipsum является стандартной "рыбой" для текстов на латинице с XVI века.
    В то время некий безымянный печатник создал большую коллекцию размеров и форм шрифтов, используя Lorem Ipsum для распечатки образцов.
    Lorem Ipsum не только успешно пережил без заметных изменений пять веков, но и перешагнул в электронный дизайн.

commit f917cba786340d1b66eee68b98f89de1824f4413
Author: Be Tester
Date: Wed Aug 5 19:59:20 2020 +0300

    changed line commit

commit 30b5ca483c25bb9bf2fe9a6779053482dad8115f
Author: Be Tester
Date: Wed Aug 5 18:41:55 2020 +0300

    Первый тестовый коммит
```

Ветвлений в проекте может быть сколько угодно, причем "отпочковываться" новая ветка может от любой произвольной ветки, а не только от **master**, как это было в нашем примере.

Все ветки существуют параллельно друг другу, разработка в них ведётся независимо. Например, создадим новый коммит в ветке **master** и убедимся, что он не повлияет на состояние ветки **feature-cart**.

Для этого, находясь сейчас в `master`, создайте новый файл `master.html`, содержащий строку "test file for master commit".

1. `touch master.html`
2. `notepad master.html`

Проверим статус созданного файла, добавим его в индекс и выполним коммит:

3. `git status`
4. `git add .`
5. `git commit -m 'крайний коммит в мастер'`

Создание и переключение веток

Убедимся, что последний коммит попал в историю коммитов на ветке master: **git log**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log
commit 05814aa59e74908371aac0fbfb158a598692e314 (HEAD -> master)
Author: Be Tester
Date:   Fri Aug 7 22:08:09 2020 +0300

    крайний коммит в мастер
```

Переключимся теперь на ветку feature-cart и посмотрим историю коммитов там: **git checkout feature-cart**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git checkout feature-cart
Switched to branch 'feature-cart'
```

Как видим, последнего коммита, выполненного на ветке master, здесь нет. Естественно, нет и файла master.html, созданного в master: **ls -la**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ ls -la
total 42
drwxr-xr-x 1 KATESATOR 197121  0 авг 7 22:09 ./
drwxr-xr-x 1 KATESATOR 197121  0 авг 6 15:52 ../
drwxr-xr-x 1 KATESATOR 197121  0 авг 7 22:09 .git/
-rw-r--r-- 1 KATESATOR 197121 15 авг 7 22:09 cart.html
-rw-r--r-- 1 KATESATOR 197121 29 авг 5 19:52 fourth.html
-rw-r--r-- 1 KATESATOR 197121 492 авг 6 22:31 index.html
-rw-r--r-- 1 KATESATOR 197121 28 авг 7 22:09 main.css
-rw-r--r-- 1 KATESATOR 197121 25 авг 6 14:09 second_file.html
-rw-r--r-- 1 KATESATOR 197121 24 авг 5 14:01 third_file.html
```

Вернёмся на ветку master и убедимся, что файл master.html в проекте есть, а файлов, относящихся к разработке корзины – нет: **git checkout master**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (feature-cart)
$ git checkout master
Switched to branch 'master'
```

Посмотрим ещё раз на историю коммитов, но на этот раз в компактном варианте (для этого служит ключ **--oneline**): **git log --oneline**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log --oneline
05814aa (HEAD -> master) крайний коммит в мастер
48717a7 Добавил боковое меню
f917cba changed line commit
30b5ca4 Первый тестовый коммит
```

- На практике командой **git log --oneline** пользуются как раз в таком кратком виде – так удобнее просматривать историю при большом количестве коммитов.

Слияние веток

Выполним теперь слияние двух наших веток и посмотрим, как после этого будет выглядеть история коммитов.

Предположим, что вся работа в ветке **feature-cart** выполнена, нужный функционал разработан. Теперь нам нужно слить ветку **feature-cart** с основной веткой **master**.

Для этого переключимся в ветку, в которую будем добавлять изменения (то есть в ветку master) и выполним команду слияния:

git checkout master

git merge feature-cart # если откроется текстовый редактор, то просто введите описание этого коммита: "Merge branch 'feature-cart' " сохраните и закройте редактор

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git merge feature-cart
Merge made by the 'recursive' strategy.
 cart.html | 1 +
 main.css  | 3 +++
2 files changed, 4 insertions(+)
create mode 100644 cart.html
create mode 100644 main.css
```

Посмотрим на структуру проекта: ls -la

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ ls -la
total 43
drwxr-xr-x 1 KATESATOR 197121  0 авг 7 22:37 ./
drwxr-xr-x 1 KATESATOR 197121  0 авг 6 15:52 ../
drwxr-xr-x 1 KATESATOR 197121  0 авг 7 22:37 .git/
-rw-r--r-- 1 KATESATOR 197121 15 авг 7 22:37 cart.html
-rw-r--r-- 1 KATESATOR 197121 29 авг 5 19:52 fourth.html
-rw-r--r-- 1 KATESATOR 197121 492 авг 6 22:31 index.html
-rw-r--r-- 1 KATESATOR 197121 28 авг 7 22:37 main.css
-rw-r--r-- 1 KATESATOR 197121 27 авг 7 22:29 master.html
-rw-r--r-- 1 KATESATOR 197121 25 авг 6 14:09 second_file.html
-rw-r--r-- 1 KATESATOR 197121 24 авг 5 14:01 third_file.html
```

Итак, теперь в проекте есть и файлы **cart.html** и **main.css**, относящиеся к ветке **feature-cart**, и файл **master.html** из ветки **master**. Таким образом, функционал корзины добавлен в основную ветку проекта.

Слияние веток

Проверим статус файлов: `git status`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Файлы находятся в согласованном состоянии, то есть при слиянии веток автоматически создастся новый коммит.

Выполним команду, которая показывает историю коммитов в наглядном виде:

`git log --oneline --graph`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log --oneline --graph
*   c638efd (HEAD -> master) Merge branch 'feature-cart' into master
\  /
 * 7e4a79f (feature-cart) cart completed
* | 05814aa крайний коммит в мастер
|/
* 48717a7 Добавил боковое меню
* f917cba changed line commit
* 30b5ca4 Первый тестовый коммит
```

Эта диаграмма показывает, что: # история коммитов на скриншоте идет **снизу вверх**

- сначала разработка шла в ветке master (первые три коммита) # видим 3 * с коммитами
- затем была создана ветка feature-cart, # зелёные/красные палки обозначают движение
- в ветке master был создан коммит с описанием 'крайний коммит в мастер' # ещё одна * слева
- в ветке feature-cart был создан коммит с описанием 'cart completed' # об этом говорит предпоследняя *
- ветка feature-cart была слита с веткой master. # пересечение двух палок в самомверху

Данная команда позволяет наглядно отобразить ход проекта: мы видим, какие возможности были добавлены или какие ошибки исправлены.

Удаление веток

У нас произошло слияние двух веток и ветка feature-cart больше не нужна:

git branch

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git branch
  feature-cart
* master
```

Удалить эту ветку можно с помощью ключа -d (delete) команды **git branch**:

git branch -d feature-cart

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git branch -d feature-cart
Deleted branch feature-cart (was 7e4a79f).
```

Убедимся, что в проекте осталась только одна ветка:

git branch

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git branch
* master
```


Конфликты при слиянии веток

В предыдущем примере слияние веток у нас прошло безо всяких проблем, потому что мы работали в разных ветках и изменяли разные файлы.

Но что будет, если мы в разных ветках изменим один и тот же файл? Как при этом пройдёт слияние веток? Если изменения в ветках затрагивали разные строки этого файла, то скорее всего Git сможет слить эти файлы вместе без ошибок. Но часто возникают ситуации, когда изменения в разных ветках затрагивают одни и те же строки в файле — в этом случае возникнет конфликт слияния.

Рассмотрим пример подобного конфликта в нашем проекте. Для этого создадим в проекте новую ветку `bug-cart` для исправления ошибки, связанной с работой корзины, и перейдём в нее: **`git checkout -b bug-cart`**

Изменим файл `index.html` — поменяем заголовок боковой панели: `notepad index.html`

1. В строке `<div class="sidebar__title">Боковое меню</div>` # заменим заголовок "Боковое меню" на "Изменения в новой ветке"

Проверим статус файлов, добавим все изменённые файлы в индекс и выполним коммит:

2. `git status`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git status
On branch bug-cart
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

3. `git add .`

4. `git commit -m 'изменения в index'`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git commit -m 'изменения в index'
[bug-cart 4231b22] изменения в index
1 file changed, 1 insertion(+), 1 deletion(-)
```

Конфликты при слиянии веток

Вернёмся теперь в ветку master:

git checkout master

И изменим тот же самый файл index.html: notepad index.html

В строке `<div class="sidebar__title">Боковое меню</div>` # заменим заголовок "Боковое меню" на "Текст который был добавлен в мастере"

Зафиксируем это изменение:

git add .

git commit -m 'изменения в файле index'

Последние выполненные коммиты для разных веток можно посмотреть с помощью команды: git branch -v

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git branch -v
  bug-cart 4231b22 изменения в index
* master   b4acab4 изменения в файле index
```

Попробуем слить две наши ветки: git merge bug-cart

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git merge bug-cart
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Автослияние index.html

КОНФЛИКТ (содержимое): Конфликт слияния в index.html

Не удалось провести автоматическое слияние; исправьте конфликты и сделайте коммит результата.

Слияние веток не прошло из-за конфликта в файле index.html. Если конфликт произошёл в нескольких файлах, то увидеть их можно с помощью `git status`

Конфликты при слиянии веток

Команда **git diff** позволяет более подробно увидеть, какая именно часть файла вызывает конфликт

Открыв файл index.html в текстовом редакторе, мы увидим, что Git внес в него информацию о конфликте (HEAD ссылается на последний коммит в текущей ветке):

```
index.html — Блокнот
Файл  Правка  Формат  Вид  Справка
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Project</title>
  </head>
  <body><!-- code -->

    <div class="sidebar">
<<<<<< HEAD
  <div class="sidebar__title">Текст который был добавлен в мастере</div>
=====
  <div class="sidebar__title">Изменения в новой ветке</div>
>>>>>> bug-cart
  <ul class="sidebar__list">
    <li class="sidebar__item"> </li>
    <li class="sidebar__item"></li>
    <li class="sidebar__item"></li>
    <li class="sidebar__item"></li>
    <li class="sidebar__item"></li>
  </ul>
</div>
```

Мы должны самостоятельно разрешить конфликт слияния, то есть определить, какие изменения должны остаться в итоговой версии файла index.html.

Оставим в этом файле код из новой ветки (bug-cart) и сохраним файл. Для этого, удалим изменения (включая пустые строки после удаления):

```
<<<<<< HEAD # до знаков равно Git отобразил нам как эта строчка выглядит в master
```

```
  <div class="sidebar__title">Текст который был добавлен в мастере</div>
```

```
=====
```

```
>>>>>> bug-cart
```

Конфликты при слиянии веток

Добавим измененные файлы в индекс и выполним коммит:

```
git add .
```

```
git commit -m 'устранили конфликт и поправили баги'
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master|MERGING)
$ git commit -m 'устранили конфликт и поправили баги'
[master 6e9caf7] устранили конфликт и поправили баги
```

Если бы в коммитах из разных веток кроме файлов, вызывающих конфликт, были и другие файлы, то они сразу бы слились с веткой master. Проблемные же файлы с конфликтами пришлось бы изменять вручную, а потом отдельно добавлять их в индекс и коммитить.

Убедимся, что слияние веток попало в историю коммитов: `git log --oneline`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git log --oneline
6e9caf7 (HEAD -> master) устранили конфликт и поправили баги
b4acab4 изменения в файле index
4231b22 (bug-cart) изменения в index
c638efd Merge branch 'feature-cart' into master
05814aa крайний коммит в мастер
7e4a79f cart completed
48717a7 Добавил боковое меню
f917cba changed line commit
30b5ca4 Первый тестовый коммит
```

Файл index.html находится в том же состоянии, в котором мы сохранили его при разрешении конфликта слияния.

```
index.html — Блокнот
Файл Правка Формат Вид Справка
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Project</title>
  </head>
  <body><!-- code -->

  <div class="sidebar">
    <div class="sidebar_title">Изменения в новой ветке</div>
    <ul class="sidebar_list">
      <li class="sidebar_item"> </li>
      <li class="sidebar_item"></li>
```


Временное (без коммита) сохранение данных

Иногда возникает необходимость оперативно переключиться в другую ветку и что-то там сделать (например, исправить выявленную критическую ошибку), но в текущей рабочей ветке файлы еще не готовы для коммита. В этом случае Git позволяет сохранить сделанные изменения, не создавая при этом коммит. Смоделируем такую ситуацию в нашем проекте.

Сейчас у нас есть две ветки и мы находимся в ветке master: **git branch**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (master)
$ git branch
  bug-cart
* master
```

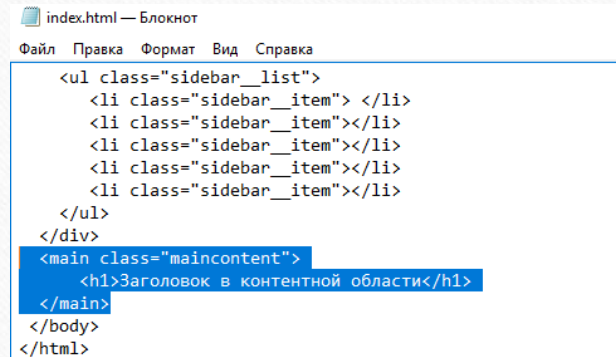
Переключимся на ветку bug-cart: **git checkout bug-cart**

Добавим что-нибудь в файл index.html, например новый заголовок перед `</body>`: **notepad index.html**

`<main class="maincontent">`

`<h1>Заголовок в контентной области</h1>`

`</main>`



```
index.html — Блокнот
Файл  Правка  Формат  Вид  Справка

<ul class="sidebar_list">
  <li class="sidebar_item"> </li>
  <li class="sidebar_item"></li>
  <li class="sidebar_item"></li>
  <li class="sidebar_item"></li>
  <li class="sidebar_item"></li>
</ul>
</div>
<main class="maincontent">
  <h1>Заголовок в контентной области</h1>
</main>
</body>
</html>
```

Проверим, что файл index.html находится в изменённом состоянии, в индекс мы его не добавляли: **git status**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git status
On branch bug-cart
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html
```

Временное (без коммита) сохранение данных

Теперь поступает задание - прервать работу в ветке `bug-cart` и вернуться на ветку `master`. Но сейчас мы ещё не готовы выполнить коммит, так как работа над задачей в `bug-cart` ещё не завершена. Конечно, технически коммит создать можно, но логически, он будет неполноценным, лишним в истории проекта (ведь коммит - это целостное осмысленное изменение проекта, он должен соответствовать какой-то полностью законченной задаче). Более того, если текущая ветка и ветка, в которую нам нужно перейти, не являются прямыми потомками друг друга, то при наличии незафиксированных изменений Git не позволит переключиться между ними, так как в этом случае незакоммиченные изменения в ветке пропадут.

Если сделанные в текущей ветке изменения не нужны, то можно заставить Git переключиться на другую ветку с помощью ключа `--force` (или кратко `-f`) команды `git checkout`. Но чаще бывает нужно сохранить уже сделанные изменения, не выполняя коммита, и потом вернуться к ним. В этом нам поможет команда `git stash`, которая собирает незакоммиченные изменения, удаляет их из файлов и в специальном виде архивирует их в Git (помещает в стек незавершенных изменений).

Выполним эту команду в нашем проекте (на ветке `bug-cart`): `git stash`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git stash
Saved working directory and index state WIP on bug-cart: 4231b22 изменения в index
```

Рабочий каталог и состояние индекса сохранены WIP on bug-cart: 0163eaa
изменения в index

Теперь мы можем спокойно переключиться на ветку `master`: `git checkout master`

Предположим, мы внесли какие-нибудь изменения в `master`

После выполнения работы в `master` мы снова переключаемся на `bug-cart`: `git checkout bug-cart`

Посмотрим, что у нас сохранено в стеке незавершенных изменений (в стеке): `git stash list` # если вдруг потребуется очистить list, используйте команду `stash list clear`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git stash list
stash@{0}: WIP on bug-cart: 4231b22 изменения в index
```

Здесь мы видим, из какой ветки и из какого коммита мы сохранили изменения, а также порядковый номер, под которым эти изменения записаны в стек. При необходимости мы можем добавлять в стек незаконченные изменения из разных веток, они будут сохраняться под разными номерами.

Временное (без коммита) сохранение данных

Посмотрим, какое содержимое сейчас имеет файл `index.html`: `cat index.html`

```
<ul class="sidebar__list">
  <li class="sidebar__item"> </li>
  <li class="sidebar__item"></li>
  <li class="sidebar__item"></li>
  <li class="sidebar__item"></li>
  <li class="sidebar__item"></li>
</ul>
</div>
</body>
</html>
```

Как видим, в файле нет разметки для блока `main`, которую мы делали ранее. Вернуть изменения из степа позволяет команда **`git stash apply`**. Если вызвать эту команду без дополнительных параметров, то она скопирует из степа файлы, находящиеся в последнем сохранённом элементе.

Также можно указать конкретный элемент степа, откуда нужно извлечь файлы в текущую ветку:

```
git stash apply stash@{0}
```

Теперь в `index.html` есть разметка для блока `main`

Давайте снова добавим изменённый файл `index.html` в степ:

```
git stash
```

```
git stash list
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git stash list
stash@{0}: WIP on bug-cart: 4231b22 изменения в index
stash@{1}: WIP on bug-cart: 4231b22 изменения в index
```

Теперь в стеке у нас две записи (более поздняя запись имеет больший индекс).

Временное (без коммита) сохранение данных

Можно просто удалить элемент из степа (без извлечения сохраненных в нем файлов): **git stash pop**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git stash pop
On branch bug-cart
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (648e05f065aabea41dc425bad04544c02cb61bc9)
```

Теперь в стеке остался только один элемент: **git stash list**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git stash list
stash@{0}: WIP on bug-cart: 4231b22 изменения в index
```

Можно просто удалить элемент из степа (без извлечения сохраненных в нем файлов): **git stash drop stash@{0}**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/lesson_4_git (bug-cart)
$ git stash drop stash@{0}
Dropped stash@{0} (a5b7c96a754f2a551afa8450552c1d4ceb876369)
```

Таким образом, команда **git stash** позволяет сохранить изменённые файлы, после чего можно переключаться на другие ветки, работать там, и вернуть файлы из степа после возвращения на нужную ветку. После этого можно завершить задачу, над которой шла работа, и создать коммит.

Заметим, что измененные файлы из степа можно извлечь на любой ветке, а не только на той, откуда они были сохранены, поэтому нужно быть внимательным, чтобы не перепутать файлы!

Удалённые репозитории

Регистрация, настройка, работа

Работа с удалёнными репозиториями

До настоящего момента мы работали с Git только на локальной машине, без удалённого репозитория.

Рассмотрим, на каких интернет-сервисах можно размещать Git-проекты для удобной работы над проектами.

Самый популярный сервис подобного рода - **GitHub** (работу с этим сервисом рассмотрим далее).

Альтернативный сервис - **Bitbucket**. До недавнего времени особенностью Bitbucket была бесплатная поддержка приватных репозиториях. На Github все бесплатные репозитории были публичными, то есть доступными на чтение всем, а за подключение приватного репозитория нужно было платить. Однако начиная с января 2019 года бесплатный приватный репозиторий можно подключить и на GitHub.

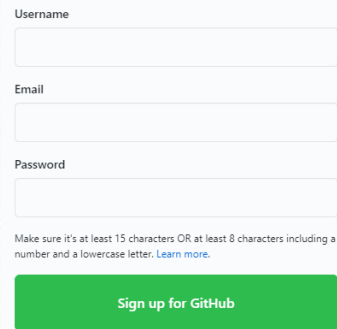
Ещё один интересный сервис - **GitLab**. Это решение можно развернуть на собственном сервере.

Регистрация и работа с GitHub

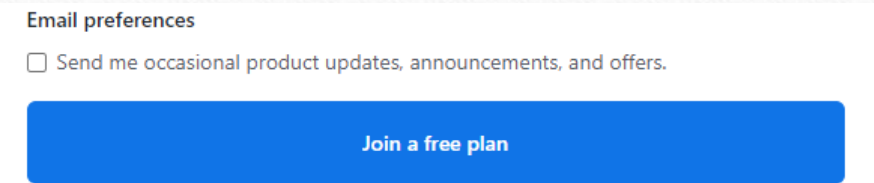
GitHub - крупнейший в мире веб-сервис для хостинга ИТ-проектов и их совместной разработки.

Создание аккаунта:

1. Откройте страницу <https://github.com/>
2. Введём данные для регистрации

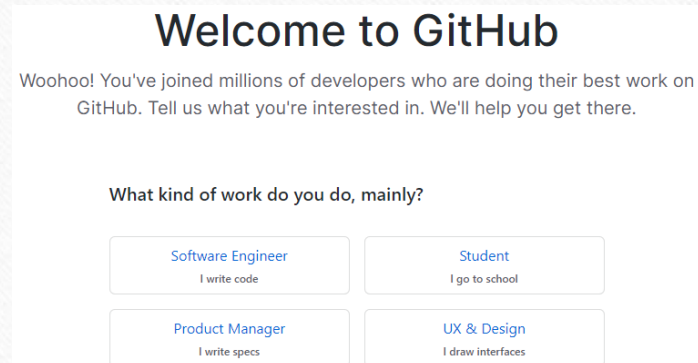
A screenshot of the GitHub registration form. It features three input fields: 'Username', 'Email', and 'Password'. Below the password field is a small text note: 'Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)'. At the bottom of the form is a green button labeled 'Sign up for GitHub'.

3. Далее может быть предложено пройти капчу на проверку что Вы не робот
4. Нажмём на "Join a free plan"

A screenshot of the 'Email preferences' section of the GitHub registration process. It includes a checkbox labeled 'Send me occasional product updates, announcements, and offers.' which is currently unchecked. Below this is a large blue button with the text 'Join a free plan'.

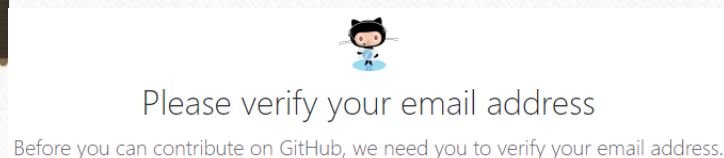
Регистрация и работа с GitHub

5. Возникнет окно, где GitHub собирает статистику о пользователях, здесь можно ничего не отвечать. Далее в самом низу нужно нажать на кнопку **"Complete Setup"**

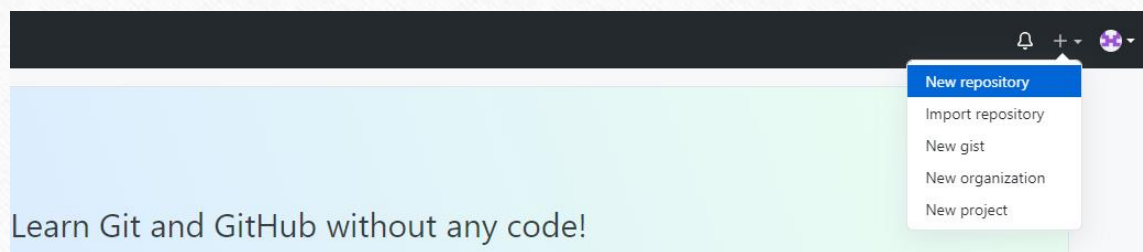


The screenshot shows the 'Welcome to GitHub' page. At the top, it says 'Woohoo! You've joined millions of developers who are doing their best work on GitHub. Tell us what you're interested in. We'll help you get there.' Below this is a question: 'What kind of work do you do, mainly?'. There are four buttons with rounded corners and light blue borders: 'Software Engineer' (with subtext 'I write code'), 'Student' (with subtext 'I go to school'), 'Product Manager' (with subtext 'I write specs'), and 'UX & Design' (with subtext 'I draw interfaces').

6. Теперь зайдите на Вашу почту, указанную при регистрации, откройте письмо от GitHub и нажмите **"Verify email address"**



7. Произойдёт переход в аккаунт GitHub # возможно будет страница с выбором с чего начать, здесь можно тогда внизу нажать на "skip this for now"
8. Нажмите на "+" -> "New repository"



Регистрация и работа с GitHub

9. В **Repository name** введите имя проекта, например "my-shop"

10. Остальное можно оставить по умолчанию. Нажмите на кнопку "Create repository"

Сразу после создания репозитория GitHub предлагает инструкции по привязке репозитория на сервере к локальному репозиторию на вашем компьютере. Мы будем использовать вариант с командой **git push**.

Запустим терминал, создадим каталог my-project и перейдём в него:

```
mkdir my-project
```

```
cd my-project
```

Создадим файл index.html со следующим содержимым: <h1>Hello World</h1>

```
touch index.html
```

```
notepad index.html
```

Развернем в нашем каталоге локальный репозиторий:

```
git init
```

Проверим статус файлов:


```
git status
```

Добавим файлы в индекс Git и создадим первый коммит:

```
git add .
```

```
git commit -m 'our first commit'
```

Quick setup — if you've done this kind of thing before

 Set up in Desktop or

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# my-shop" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/be-tester1/my-shop.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/be-tester1/my-shop.git
git push -u origin master
```

Привязка локального репозитория к GitHub

Чтобы загрузить теперь наш локальный репозиторий на GitHub, мы должны сообщить Git'у адрес нужного нам удалённого репозитория.

Выполним команду, которая добавит в систему наш удалённый репозиторий, обращаться к которому можно будет по ссылке (псевдониму) origin:

```
git remote add origin https://github.com/ваш username/ваше название проекта.git # например: https://github.com/be-tester1/my-shop.git
```

Отметим, что удалённый репозиторий не обязательно называть origin, имя этой ссылки может быть любым.

Теперь мы должны с помощью команды **git push удалённый репозиторий ветка** отправить локальный репозиторий на удалённый сервер ("запустить репозиторий").

```
git push -u origin master # появится окно, где нужно ввести email и токен от GitHub (будет написано поле password, но нужен будет именно токен)
```

для получения токена нужно: в аккаунте github нажать на иконку пользователя вверху справа->Settings->Developer settings>Personal access tokens->Generate new token

->задать любое название в note->отметить все чекбоксы ниже->Generate token->скопировать токен(который на зеленом фоне). Значение токена – будет паролем

Может быть такое, что с первого раза не пройдет авторизация и появится новое окно SSH, в нем нужно точно также повторить ввод логина и токена

```
KATESATOR@KATES-COMPUTER MINGW64 ~/my-project (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 232 bytes | 232.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/be-tester1/my-shop.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

если все пройдет успешно, то возникнет подобное сообщение как на скриншоте

Если показывает ошибку 403/ you don't have permission to access: Панель управления -> Учётные записи пользователей -> Администрирование учетных записей -> Учетные данные Windows -> удалите строку с git, перезапустите консоль, снова зайдите в нужную директорию(через **cd**) и повторите заново **git push -u origin master**

Привязка локального репозитория к GitHub

Обновите страницу GitHub. Теперь мы можем увидеть свой репозиторий на сервере GitHub. На вкладке "<> Code" (или на главной странице в списке слева) выводится информация о текущей ветке репозитория и о том, какие коммиты и когда выполнялись в проекте. С помощью раскрывающегося списка "Branch"(сейчас выбран master) можно переключиться на другую ветку проекта.

The screenshot shows the GitHub interface for a repository named 'my-shop' by user 'be-tester1'. The top navigation bar includes the GitHub logo, a search bar, and links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. The main content area features a large light blue box with the text 'Learn Git and GitHub without any code!' and a green button labeled 'Read the guide'. Below this, the repository name 'be-tester1 / my-shop' is displayed with icons for 'Unwatch', 'Star', and 'Fork'. A secondary navigation bar contains links for 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. The 'Code' tab is active, showing a dropdown for the 'master' branch, a list of files including 'index.html', and a commit by 'KATESATOR'. On the right, the 'About' section is empty, and the 'Releases' section shows 'No releases published'.

Search or jump to... Pull requests Issues Marketplace Explore

Learn Git and GitHub without any code!

Using the Hello World guide, you'll start a branch, write comments, and open a pull request.

Read the guide

be-tester1 / my-shop Unwatch 1 Star 0 Fork 0

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master 1 branch 0 tags Go to file Add file Code

KATESATOR our first commit 530e516 25 minutes ago 1 commits

index.html our first commit 25 minutes ago

Help people interested in this repository understand your project by adding a README. Add a README

About No description, website, or topics provided.

Releases No releases published Create a new release

Привязка локального репозитория к GitHub

Кроме непосредственной поддержки Git-репозитория с историей коммитов GitHub предлагает дополнительные инструменты и возможности для совместной работы над проектом:

- **Система учёта ошибок (bugtracker) Issues**, позволяющая создавать хорошо документированные, интерактивные обсуждения багов и функций любого проекта. Любой человек может открыть issue (вопрос по проблеме) в чем-то проекте, и он останется открытым и доступным всем для просмотра, пока его не закроет инициатор (например, если ему удалось разобраться с проблемой) или владелец репозитория.
- **Форки (forks) и пул-реквесты (Pull Requests)**. Любой пользователь может сделать форк (создать копию) вашего репозитория, внести какие-то изменения, а затем создать пул-реквест, чтобы попросить вас слить предложенные им изменения с вашим проектом. В больших проектах типа React или Vue.js могут быть открыты сотни пул-реквестов.
- Подписка на определённого разработчика или репозиторий, позволяющая видеть связанные с ними активности на своей панели инструментов.

Файлы .gitignore, readme.md

В нашем репозитории не хватает двух файлов, которые должны быть в любом проекте, выставляемом на GitHub:

- Файл **.gitignore**, располагающийся в корне проекта и содержащий список файлов и каталогов, за которыми Git не должен следить и которые не будут сохраняться в репозитории.
- Файл **readme.md** в формате Markdown с описанием проекта.

Создадим эти файлы:

```
touch .gitignore
```

```
touch readme.md
```


Привязка локального репозитория к GitHub

Создадим каталог, который мы добавим в .gitignore. Пусть это будет каталог .idea, в котором хранятся параметры среды разработки при работе с IDE фирмы JetBrains (WebStorm, PhpStorm, PyCharm, IntelliJ IDEA):

```
mkdir .idea
```

```
ls -la
```

Добавим в каталог .idea файл (так как пустые папки без файлов Git игнорирует): `touch .idea/test.xml`

Теперь папка .idea будет показана в git status как неотслеживаемая: `git status`

Папка .idea не имеет отношения к самому проекту, сохранять в репозитории ее не нужно. В принципе можно каждый раз исключать ее из команды git add, чтобы она не попадала в индекс, но это очень неудобно.

Поэтому добавим имя каталога .idea в файл .gitignore, тогда Git просто забудет о существовании этой папки:

```
notepad .gitignore
```

Введем: `.idea/` # слеш в конце указывает, что .idea является папкой — в этом случае Git будет игнорировать все ее содержимое)

Убедимся, что папка .idea пропала из списка интересующих Git файлов: `git status`

Закоммитим наши изменения и пошлём их на удалённый сервер:

```
git add .
```

```
git commit -m 'added gitignore and readme'
```

```
git push
```

По умолчанию Git "пушит" файлы в ту ветку, в которой мы работали локально (в нашем случае это ветка master).

Клонирование репозитория с ГН на компьютер

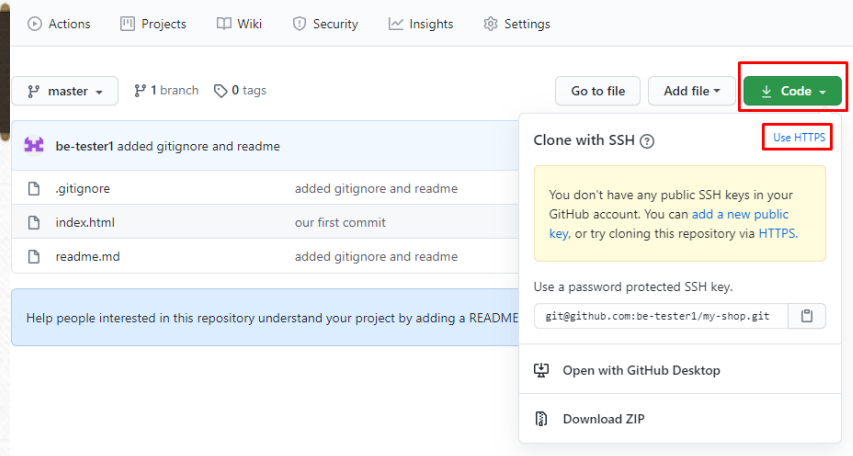
Часто возникает необходимость работать с одним и тем же репозиторием на GitHub с разных машин (например, с домашнего и рабочего компьютеров). Рассмотрим, как можно клонировать удалённый GitHub-репозиторий на компьютер, где этого проекта ещё не было.

Выйдем из каталога my-project на один уровень вверх и удалим весь этот каталог:

```
cd ../
```

rm -rf my-project/ # проверить что сработало, можно выполнив команду **cd my-project**; должно появиться сообщение: no such file or directory

Теперь на нашем компьютере проекта нет. Склонируем его из репозитория GitHub. Для этого зайдём в свой аккаунт на GitHub, откроем страницу нашего репозитория и нажмём кнопку "Code".



Клонировать репозиторий можно по протоколу HTTPS или по протоколу SSH. Воспользуемся сначала протоколом HTTPS.

Клонирование репозитория с ГН на компьютер

Скопируем ссылку на репозиторий и выполним в терминале команду: # обратите внимание на пробел перед my-shop-project

git clone **ваша ссылка** **my-shop-project** # например: `git clone https://github.com/be-tester1/my-shop.git my-shop-project`

Последний параметр выполненной команды — имя каталога, в который будет клонирован удалённый репозиторий (my-shop-project). Если его не указать, то клонирование производится в папку, совпадающую с названием репозитория (в нашем случае my-shop).

```
KATESATOR@KATES-COMPUTER MINGW64 ~  
$ git clone https://github.com/be-tester1/my-shop.git my-shop-project  
Cloning into 'my-shop-project'...  
remote: Enumerating objects: 7, done.  
remote: Counting objects: 100% (7/7), done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 7 (delta 0), reused 7 (delta 0), pack-reused 0  
Unpacking objects: 100% (7/7), 518 bytes | 39.00 KiB/s, done.
```

Зайдём в папку my-shop-project и убедимся, что все файлы проекта восстановлены:

cd my-shop-project

ls -la

```
$ ls -la  
total 42  
drwxr-xr-x 1 KATESATOR 197121 0 авг 10 12:02 ./  
drwxr-xr-x 1 KATESATOR 197121 0 авг 10 12:02 ../  
drwxr-xr-x 1 KATESATOR 197121 0 авг 10 12:02 .git/  
-rw-r--r-- 1 KATESATOR 197121 6 авг 10 12:02 .gitignore  
-rw-r--r-- 1 KATESATOR 197121 20 авг 10 12:02 index.html  
-rw-r--r-- 1 KATESATOR 197121 0 авг 10 12:02 readme.md
```

Доступ к GitHub с помощью SSH-ключа

Пока мы работали с удалённым репозиторием на GitHub по протоколу HTTPS. В этом случае приходится вводить имя пользователя и пароль при каждом новом сеансе, что не очень удобно.

При постоянной работе с удалённым репозиторием можно настроить подключение к нему по протоколу HTTPS, после чего не нужно будет больше вводить учётные данные для подключения к репозиторию GitHub.

Для этого нужно сгенерировать SSH-ключ. Делается это с помощью команды: `ssh-keygen`

Будет предложено ввести название файла для сохранения и создать кодовую фразу, всё это можно пропустить, трижды нажав `enter`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/my-shop-project (master)
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/KATESATOR/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/KATESATOR/.ssh/id_rsa
Your public key has been saved in /c/Users/KATESATOR/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:b7Cp7DM0ESkf8ON09wA1i7Njep1pXfnC+vDOW6u3HDM KATESATOR@KATES-COMPUTER
The key's randomart image is:
+---[RSA 3072]-----+
|  ..  ...o  |
|  ..+  o o  |
|  o+o+ +    |
|  ooo + o .  |
|  ..S  +    |
|  oo @ o .  |
|  ...0 = o E  |
|  .O+ . = +.= |
|  .+o  .oB++  |
+-----[SHA256]-----+
```

Для дополнительной безопасности команда запрашивает кодовую фразу, которую можно оставить пустой. В результате сгенерируется пара ключей: приватный (секретный) и публичный (открытый), которые по умолчанию сохраняются в виде файлов `id_rsa` и `id_rsa.pub` в домашнем каталоге в папке `.ssh`. Публичный ключ размещается на сетевых ресурсах, к которым обращаются по SSH, и реализованные в этом протоколе алгоритмы шифрования позволяют предоставить доступ только тому пользователю, на компьютере которого имеется соответствующий приватный ключ.

Доступ к GitHub с помощью SSH-ключа

Перейдем в папку с ключами и посмотрим ее содержимое:

```
cd ~/.ssh
```

```
ls -la
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/my-shop-project (master)
$ cd ~/.ssh

KATESATOR@KATES-COMPUTER MINGW64 ~/.ssh
$ ls -la
total 41
drwxr-xr-x 1 KATESATOR 197121  0 авг 10 12:38 ./
drwxr-xr-x 1 KATESATOR 197121  0 авг 10 12:02 ../
-rw-r--r-- 1 KATESATOR 197121 2610 авг 10 12:38 id_rsa
-rw-r--r-- 1 KATESATOR 197121  578 авг 10 12:38 id_rsa.pub
-rw-r--r-- 1 KATESATOR 197121  799 авг  9 22:31 known_hosts
```

Выведем содержимое публичного ключа:

```
cat id_rsa.pub
```

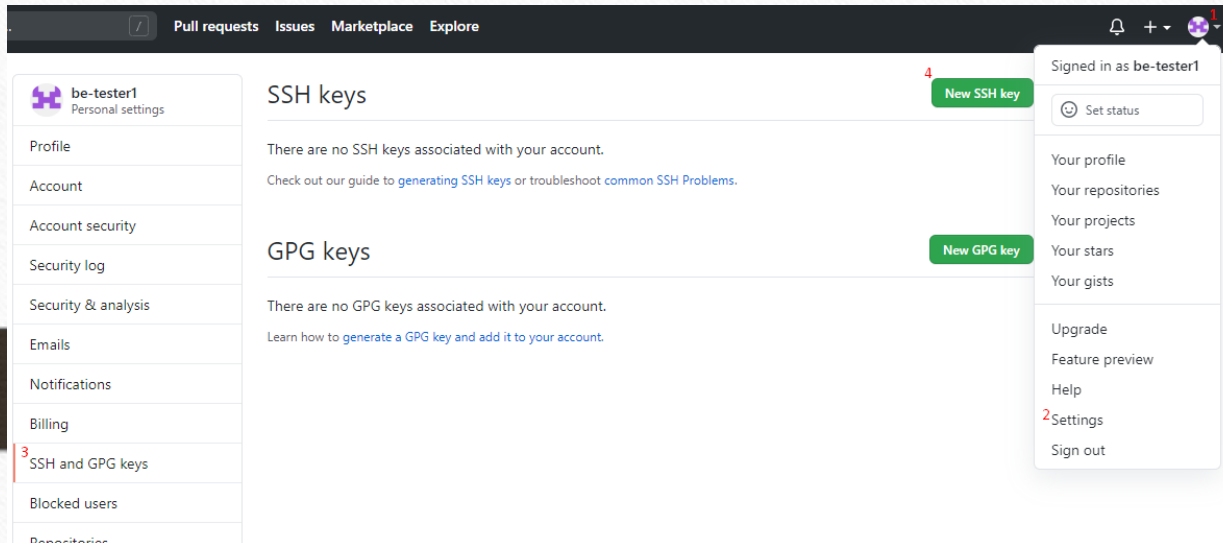
```
KATESATOR@KATES-COMPUTER MINGW64 ~/.ssh
$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQGCqUyW9szNGUZYodhiDDMGBFWKgtRkSY/yaAILFFVmexmep6uB3nQYYBhY7QiScQJFWdNQLuZ01rFLdRx02
FP58PHLhNX7vUH98G785XvZuCxNW06MPQywqlXbo1JgjMY/dZSdxBA3TgDges2A1kCJGW9gLS5oPPEfPVbChrgI7rUVy6l1SVxFMB85Q8fw3yAycDK4JF3iy
oEpo+Ex/bwnZ62GIcqq0/C21UXj9WHTHm6fXEpeOETul1XCvubsvw4YQaX4BSfiS+oJzdHr14YqS1KWeqAqIJ9i0aqJPu2Is049QV6r+2ydzeNxnE2q9Ew8P
l4MntOmE35u9oHX5F5EufeZawGFnL23g3mCOLpHB/IUe2nWBSYDVkcDdA0/QjsG3jMX3v1LX9IUFNypuXDHoFz9q+czLFKeIJWL9/KnevX3HvsPo69BxTH6z
```

Выгрузим содержимое ключа в отдельный файл на рабочем столе:

```
cat id_rsa.pub > ~/Desktop/public-key.txt
```

Доступ к GitHub с помощью SSH-ключа

Теперь нужно зарегистрировать полученный публичный ключ в аккаунте GitHub. Для этого скопируем в буфер содержимое файла `id_rsa.pub` (содержимое выгрузили на рабочий стол в файл `public-key.txt`), откроем в браузере страницу с настройками нашего аккаунта на GitHub и выберем пункт **"SSH and GPG keys"** -> **"New SSH key"**



Введём название `local-machine` для ключа в поле **"Title"** и скопируем из буфера сам ключ в поле **"Key"**:

[SSH keys / Add new](#)

Title

local-machine

Key

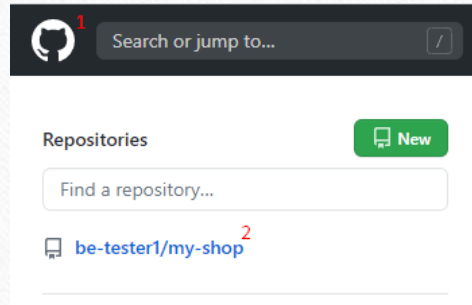
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQgQCqUyW9szNGUZyodhiDDMGBFWKgtRkSY/jaAILFFVmxemep6uB3nQYYB
hY7QJScQFWdNQuZO1rLdRx02FP58PHLhNX7vUH98G785XvZuCXNW06MPQywwqIXbo1JgJMY/dZSdxBA3TgDges2
A1kCJGW9gLS5oPPEfPvBChrgl7rUVy6l1SVxvFM885Q8fW3yAycDK4JF3iyoEpo+Ex/bwnZ62Glcqq0/C21UXj9WHTHm
6fXEpeOETul1XCvubsvw4YQaX4B5fS+oJzdHrI4YqS1KWeqAqIJ9i0aqIPu2Is049QV6r+2ydzNxmE2q9Ew8PI4MntOm
E35u9oHXSF5EufeZawGFnl23g3mCOLpHB/IUe2nWBSYDVkdDdA0/QjsG3jMX3v1LX9IUFNypuXDHoFz9q+czLFKeIJ
WL9/KnevX3HvsPo69BxTH6zf9vzhKYFC9AWOxVY9vwqXG7Ej8YnTPa5HqRUKtMvnh0/ukyjjRflht1ncB

Add SSH key

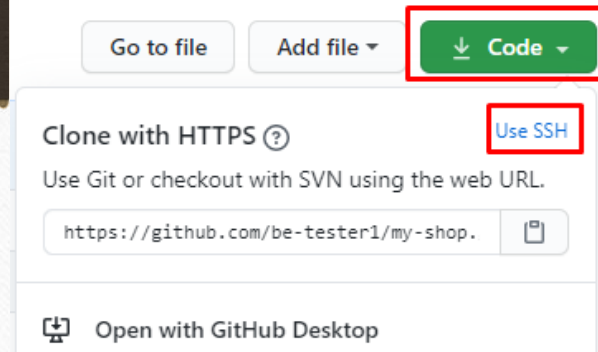
После нажатия на кнопку **"Add SSH key"** ключ будет добавлен (при этом GitHub может запросить ввести пароль для вашего аккаунта).

Доступ к GitHub с помощью SSH-ключа

Вернёмся теперь на страницу репозитория my-shop:



Нажмем кнопку "Code" и выберем ссылку "Use SSH" – мы увидим адрес, по которому можно обращаться к репозиторию по протоколу SSH



Теперь перезапустим терминал и настроим работу с удалённым репозиторием по протоколу SSH.

cd my-shop-project

Посмотрим сначала, какие удалённые репозитории зарегистрированы в нашей системе:

```
git remote show      # origin
```

Доступ к GitHub с помощью SSH-ключа

Переключим репозиторий, доступный по ссылке origin, с протокола HTTPS на протокол SSH.

git remote set-url origin *ваша ssh-ссылка* # например: **git remote set-url origin git@github.com:be-tester123/book_store_testing_1.git**

После этого мы сможем "пушить" коммиты по протоколу SSH, без ввода имени и пароля. Для демонстрации этого изменим файл `readme.md` и зафиксируем изменения в удалённом репозитории.

Как уже отмечалось ранее, файл `readme.md` должен содержать описание проекта, а текст в нём должен быть записан в формате Markdown. Markdown – это очень простой язык разметки, позволяющий выделять заголовки различных уровней, списки, ссылки, картинки. <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>

Запишем в `readme.md` заголовок первого уровня, список из двух элементов и команду на языке `bash`, используя cheatsheet по ссылке выше:

• **notepad readme.md**

Скопируем содержимое ниже в файл, сохраните и закройте файл:

• "# Это текстовый проект для школы тестировщиков be-tester

+ Показать как работает гит

+ Познакомиться с основными командами

```bash

\$ git clone git@github..."



# Доступ к GitHub с помощью SSH-ключа

Запишем изменённые файлы в индекс и выполним новый коммит:

```
git add .
```

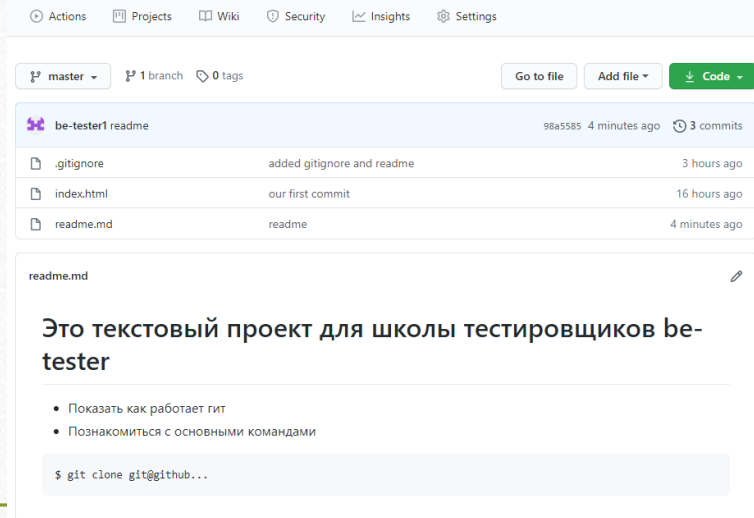
```
git commit -m 'readme'
```

"Запустим" наш коммит в удалённый репозиторий (при первом обращении к GitHub по протоколу SSH система спросит, доверяем ли мы серверу, — напомним "yes"):

```
git push
```

```
KATESATOR@KATES-COMPUTER MINGW64 ~/my-shop-project (master)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 386 bytes | 386.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:be-tester1/my-shop.git
269d093..98a5585 master -> master
```

Теперь форматированное содержимое файла `readme.md` будет отображаться на веб-странице нашего проекта на GitHub.



The screenshot shows the GitHub interface for a repository named 'be-tester1'. At the top, there are navigation links for Actions, Projects, Wiki, Security, Insights, and Settings. Below these, there's a header with 'master' branch selected, '1 branch', and '0 tags'. A table lists the files in the repository: '.gitignore', 'index.html', and 'readme.md'. The 'readme.md' file is selected, and its content is displayed below. The content includes a title 'Это текстовый проект для школы тестировщиков be-tester', a bulleted list of topics, and a code block for cloning the repository.

| File       | Description                | Time          |
|------------|----------------------------|---------------|
| .gitignore | added gitignore and readme | 3 hours ago   |
| index.html | our first commit           | 16 hours ago  |
| readme.md  | readme                     | 4 minutes ago |

**readme.md**

## Это текстовый проект для школы тестировщиков be-tester

- Показать как работает гит
- Познакомиться с основными командами

```
$ git clone git@github...
```

Файл `readme.md` обязательно должен быть в проекте, размещённом на GitHub. В нем можно писать документацию к проекту, приводить примеры кода, вставлять ссылки.

# Использование GitHub Pages для хостинга проектов

GitHub можно использовать для бесплатного статического хостинга своих проектов, не требующих серверной части. Другими словами, если ваш проект содержит только фронтенд (работает в браузере), то вы можете опубликовать его на GitHub и он будет доступен всему миру как веб-страница.

Для публикации нашего проекта создадим новый публичный репозиторий со специальным именем формата **имя\_пользователя.github.io**, то есть начинаться оно должно с имени аккаунта, а заканчиваться на **"github.io"** (**username.github.io**).


### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?  
[Import a repository.](#)

---

Owner \*

Repository name \*

 be-tester1 ▾


 / 

be-tester1.github.io


Great repository names are short and memorable. Need inspiration? How about [super-carnival](#)?

Description (optional)

---

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.


---

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**  
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾

Add a license: **None** ▾

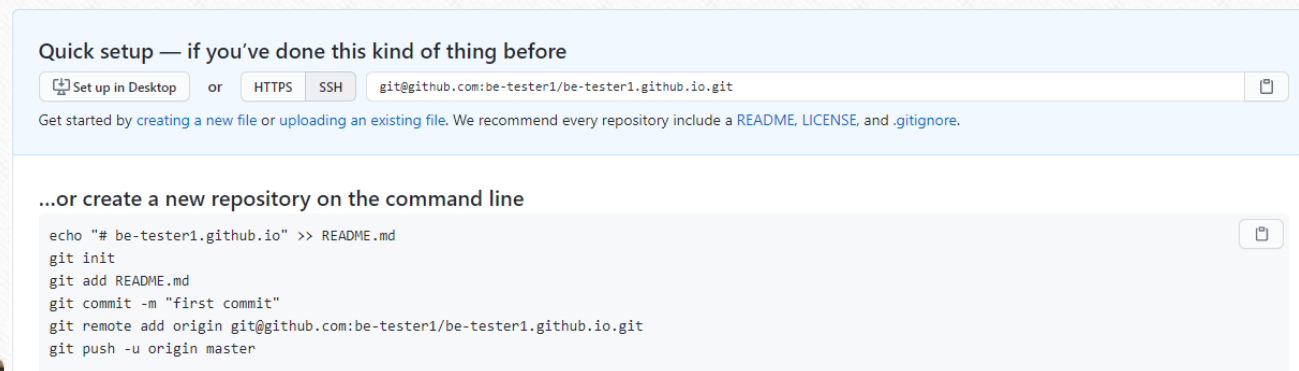


Репозитории с подобными именами считываются сервисом GitHub Pages, позволяющим посмотреть их содержимое по прямой ссылке непосредственно в браузере (не нужно предварительно клонировать репозиторий и собирать проект).



# Использование GitHub Pages для хостинга проектов

После создания репозитория скопируем в буфер его **SSH** адрес:



Запустим терминал, перейдём в папку **my-shop-project** с нашим проектом и проверим её содержимое: **ls -la**

```
KATESATOR@KATES-COMPUTER MINGW64 ~/my-shop-project (master)
$ ls -la
total 43
drwxr-xr-x 1 KATESATOR 197121 0 авг 10 12:02 ./
drwxr-xr-x 1 KATESATOR 197121 0 авг 10 12:02 ../
drwxr-xr-x 1 KATESATOR 197121 0 авг 10 14:06 .git/
-rw-r--r-- 1 KATESATOR 197121 6 авг 10 12:02 .gitignore
-rw-r--r-- 1 KATESATOR 197121 20 авг 10 12:02 index.html
-rw-r--r-- 1 KATESATOR 197121 164 авг 10 14:06 readme.md
```

Откроем в текстовом редакторе файл **index.html** и заменим его содержимое на строку "**<h1>Hello from GitHubPages</h1>**":

Добавим еще один удалённый репозиторий с псевдонимом **pages**, который будет ссылаться на созданный нами репозиторий с именем **ваш username.github.io** (адрес этого репозитория мы скопировали в буфер с сервера GitHub):

**git remote add pages git@github.com:username/username.github.io.git**    # можно скопировать ssh адрес с страницы GitHub (1-й шаг этого слайда)

# Использование GitHub Pages для хостинга проектов

Проверим статус файлов в проекте: `git status`

```
KATESATOR@KATES-COMPUTER MINGW64 ~/my-shop-project (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
 modified: index.html
```

Добавим изменённые файлы в индекс и выполним коммит:

`git add .`

`git commit -m 'commit for pages'`

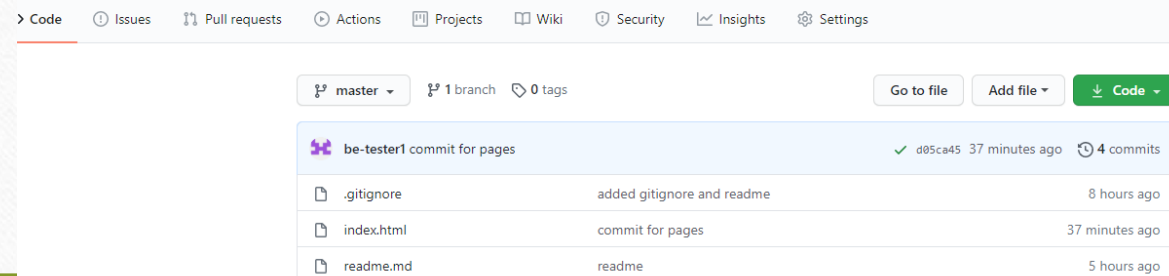
Отправим коммит в удалённый репозиторий pages (нужно указать флаг `-u` (upstream), так как в удалённом репозитории ещё нет ветки master):

`git push pages master -u`

Теперь репозиторий **ваш `username.github.io`** содержит файлы нашего проекта.

# можно найти его в списке репозитиев

[be-tester1 / be-tester1.github.io](#)



> Code   Issues   Pull requests   Actions   Projects   Wiki   Security   Insights   Settings

master 1 branch 0 tags   Go to file   Add file   Code

|                                                                |                            |                |
|----------------------------------------------------------------|----------------------------|----------------|
| be-tester1 commit for pages ✓ d05ca45 37 minutes ago 4 commits |                            |                |
| .gitignore                                                     | added gitignore and readme | 8 hours ago    |
| index.html                                                     | commit for pages           | 37 minutes ago |
| readme.md                                                      | readme                     | 5 hours ago    |



# Использование GitHub Pages для хостинга проектов

Теперь откроем в браузере адрес [https://ваш\\_username.github.io/](https://ваш_username.github.io/) и увидим контент из `index.html`.



Репозитории на GitHub Pages публикуются не моментально, появление страницы может занять несколько минут.

Давайте проверим, будут ли изменения в проекте влиять на опубликованную страницу. Изменим содержимое файла `index.html` на:

```
<h1>Hello from GitHub Pages again</h1>
```

Создадим новый коммит и отправим на GitHub:

```
git add .
```

```
git commit -m 'another commit'
```

```
git push pages master
```

Обновив в браузере страницу [https://ваш\\_username.github.io/](https://ваш_username.github.io/), мы увидим новое содержимое: # обновление может занять несколько минут



# Автоматизация

---

Интернет-магазина по тест-кейсам

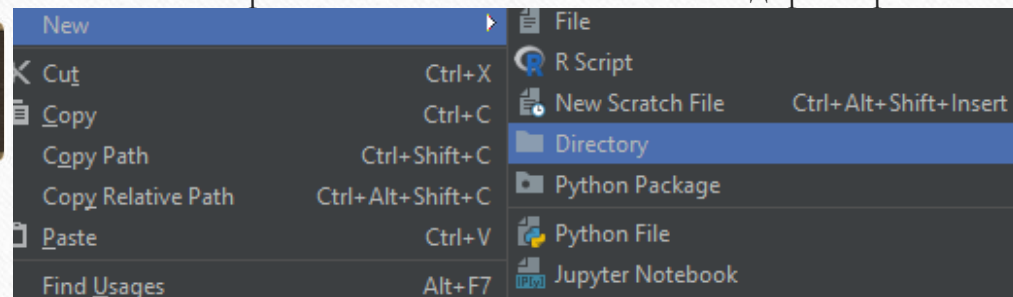


# Практика по тест-кейсам

Для применения полученных знаний по Git, переключимся на автоматизацию по тест-кейсам и в конце создадим новый репозиторий с авто-тестами.

Каждая категория тестов будет разделена по страницам: home, registration\_login, shop

1. В PyCharm, в списке всех файлов слева, нажмите правой кнопкой мыши на пустом пространстве, выберите "New" -> "Directory"
2. Введите название "book\_store\_testing"
3. Нажмите правой кнопки мыши на названии директории "book\_store\_testing", выберите "New" -> "Python File"



4. Создайте внутри директории "book\_store\_testing" три python файла, с названиями: home, login\_registration, shop
  5. В дальнейших заданиях, добавляйте тесты в соответствующий файл (можно ориентироваться на заголовок слайда)
- В некоторых сценариях иногда возможно случайное нажатие на другой элемент, в этом случае, просто подкорректируйте тайминги

# Home: добавление комментария

1. Откройте <https://practice.automationtesting.in/>
2. Прокрольте страницу вниз на 600 пикселей
3. Нажмите на название книги "Selenium Ruby" или на кнопку "READ MORE"
4. Нажмите на вкладку "REVIEWS"
5. Поставьте 5 звёзд
6. Заполните поле "Review" сообщением: "Nice book!"
7. Заполните поле "Name"
8. Заполните "Email"
9. Нажмите на кнопку "SUBMIT"



# Registration\_login: регистрация аккаунта

1. Откройте <https://practice.automationtesting.in/>
2. Нажмите на вкладку "My Account"
3. В разделе "Register", введите email для регистрации
4. В разделе "Register", введите пароль для регистрации
  - составьте такой пароль, чтобы отобразилось "Medium" или "Strong", иначе регистрация не выполнится
  - почту и пароль сохраните, потребуются в дальнейшем
5. Нажмите на кнопку "Register"

# Registration\_login: логин в систему

1. Откройте <https://practice.automationtesting.in/>
2. Нажмите на вкладку "My Account"
3. В разделе "Login", введите email для логина # данные можно взять из предыдущего теста
4. В разделе "Login", введите пароль для логина # данные можно взять из предыдущего теста
5. Нажмите на кнопку "Login"
6. Добавьте проверку, что на странице есть элемент "Logout"



# Shop: отображение страницы товара

1. Откройте <https://practice.automationtesting.in/>
2. Залогиньтесь
3. Нажмите на вкладку "Shop"
4. Откройте книгу "HTML 5 Forms"
5. Добавьте тест, что заголовок книги называется: "HTML5 Forms"

# Shop: количество товаров в категории

1. Откройте <https://practice.automationtesting.in/>
2. Залогиньтесь
3. Нажмите на вкладку "Shop"
4. Откройте категорию "HTML"
5. Добавьте тест, что отображается три товара



# Shop: сортировка товаров

1. Откройте <https://practice.automationtesting.in/>
2. Залогиньтесь
3. Нажмите на вкладку "Shop"
4. Добавьте тест, что в селекторе выбран вариант сортировки по умолчанию
  - Используйте проверку по value
5. Отсортируйте товары по цене от большей к меньшей
  - в селекторах используйте класс Select
6. Снова объявите переменную с локатором основного селектора сортировки # т.к после сортировки страница обновится
7. Добавьте тест, что в селекторе выбран вариант сортировки по цене от большей к меньшей
  - Используйте проверку по value

# Shop: отображение, скидка товара

1. Откройте <https://practice.automationtesting.in/>
2. Залогиньтесь
3. Нажмите на вкладку "Shop"
4. Откройте книгу "Android Quick Start Guide"
5. Добавьте тест, что содержимое старой цены = "₹600.00" # используйте assert
6. Добавьте тест, что содержимое новой цены = "₹450.00" # используйте assert
7. Добавьте явное ожидание и нажмите на обложку книги
  - Подберите такой селектор и тайминги, чтобы открылось окно предпросмотра картинки (не вся картинка на всю страницу)
8. Добавьте явное ожидание и закройте предпросмотр нажав на крестик (кнопка вверху справа)



# Shop: проверка цены в корзине

1. Откройте <https://practice.automationtesting.in/> # в этом тесте логиниться не нужно
2. Нажмите на вкладку "Shop"
3. Добавьте в корзину книгу "HTML5 WebApp Development" # см. комментарии в самом низу
4. Добавьте тест, что возле корзины(вверху справа) количество товаров = "1 Item", а стоимость = "₹180.00"
  - Используйте для проверки assert
5. Перейдите в корзину
6. Используя явное ожидание, проверьте что в Subtotal отобразилась стоимость
7. Используя явное ожидание, проверьте что в Total отобразилась стоимость

# если эта книга будет out of stock - тогда вместо неё добавьте книгу HTML5 Forms и выполните тесты по аналогии

# если все книги будут out of stock - тогда пропустите это и следующие два задания

# Shop: работа в корзине

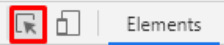
Иногда, даже явные ожидания не помогают избежать ошибки при нахождении элемента, этот сценарий один из таких, используйте `time.sleep()`

1. Откройте <https://practice.automationtesting.in/> # в этом тесте логиниться не нужно
2. Нажмите на вкладку "Shop"
3. Добавьте в корзину книги "HTML5 WebApp Development" и "JS Data Structures and Algorithm"
  - Перед добавлением первой книги, проскрольте вниз на 300 пикселей
  - После добавления 1-й книги добавьте `sleep`
4. Перейдите в корзину
5. Удалите первую книгу
  - Перед удалением добавьте `sleep`
6. Нажмите на Undo (отмена удаления)
7. В Quantity увеличьте количество товара до 3 шт для "JS Data Structures and Algorithm"
  - Предварительно очистите поле с помощью `локатор_поля.clear()`
8. Нажмите на кнопку "UPDATE BASKET"
9. Добавьте тест, что value элемента quantity для "JS Data Structures and Algorithm" равно 3 # используйте `assert`
10. Нажмите на кнопку "APPLY COUPON"
  - Перед нажатием добавьте `sleep`
11. Добавьте тест, что возникло сообщение: "Please enter a coupon code."

# если эти книги будут out of stock - тогда вместо них добавьте книгу HTML5 Forms и любую доступную книгу по JS и выполните тесты по аналогии

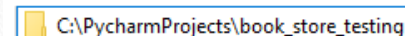


# Shop: покупка товара

1. Откройте <https://practice.automationtesting.in/> # в этом тесте логиниться не нужно
2. Нажмите на вкладку "Shop" и проскрольте на 300 пикселей вниз
3. Добавьте в корзину книгу "HTML5 WebApp Development"
4. Перейдите в корзину
5. Нажмите "PROCEED TO CHECKOUT"
  - Перед нажатием, добавьте явное ожидание
6. Заполните все обязательные поля
  - Перед заполнением first name, добавьте явное ожидание
  - Для заполнения country нужно: нажать на селектор - > ввести название в поле ввода - > нажать на вариант который отобразится ниже ввода
  - Чтобы выбрать селектор нижний вариант после ввода, используйте кнопку  нажмите на неё, затем на вариант в списке ниже
7. Выберите способ оплаты "Check Payments"
  - Перед выбором, проскрольте на 600 пикселей вниз и добавьте sleep
8. Нажмите PLACE ORDER
9. Используя явное ожидание, проверьте что отображается надпись "Thank you. Your order has been received."
10. Используя явное ожидание, проверьте что в Payment Method отображается текст "Check Payments"

# Создание своего репозитория

1. Создайте репозиторий на GitHub
2. Откройте консоль GitBash
3. Перейдите в директорию папки book\_store\_testing
  - Можно нажать ПКМ на папке в PyCharm - > Show in Explorer -> перейти в папку -> скопировать её адрес
  - Нужно заменить слеш в обратную сторону, например: C:/ваш адрес/book\_store\_testing
4. Проверьте в консоли что вы точно в той папке, узнав её содержимое
5. Создайте в этой директории локальный репозиторий
6. Добавьте в индекс все файлы этой директории
7. Создайте коммит с комментарием
8. Добавьте в систему удалённый репозиторий через git remote
9. Отправьте локальный репозиторий на удаленный сервер
10. Проверьте что файлы успешно отправились на Ваш GitHub репозиторий



C:\PycharmProjects\book\_store\_testing

- При дальнейшем изменении этих файлов в PyCharm – их название будет отображаться синим цветом(так как Git их теперь отслеживает)



# Домашнее задание

---

1. Попробуйте решить как можно больше задач из всего урока. Код нужно отправить по ссылке: <https://forms.gle/2CwcJGEiJLxfRJRp7>
2. Заполните анкету обратной связи по ссылке: <https://forms.gle/qnb6sKioC9BfGtp16>
3. Прочитайте файл с инструкцией в папке с этим уроком

