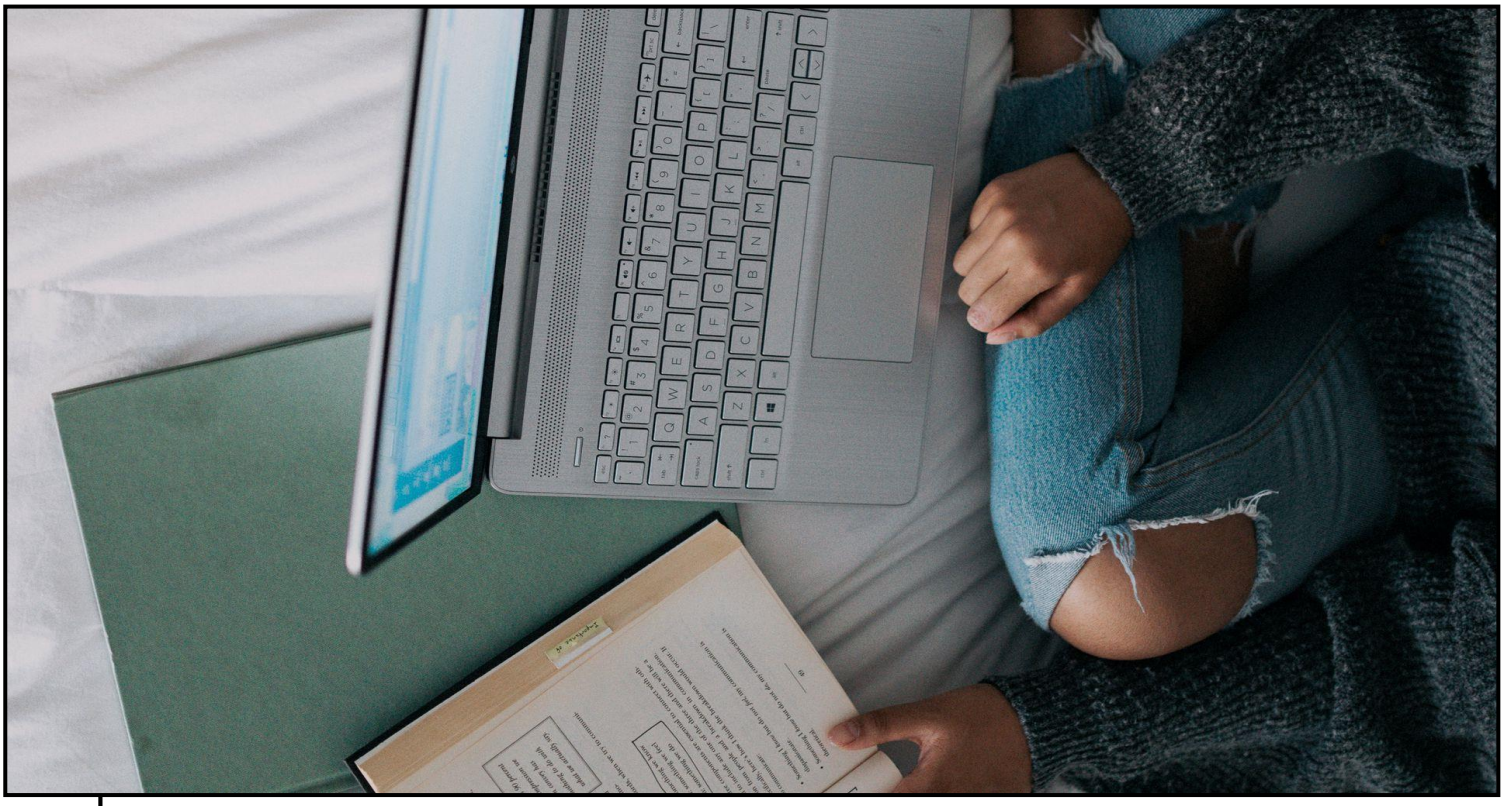

СИСТЕМА
КОНТРОЛЯ
ВЕРСИЙ
GIT



Содержание:

1. Системы контроля версий и их применение
2. Классификация СКВ
3. Установка Git на ПК
4. GitHub
5. Работа с репозиториями
6. Основные команды

Системы контроля версий

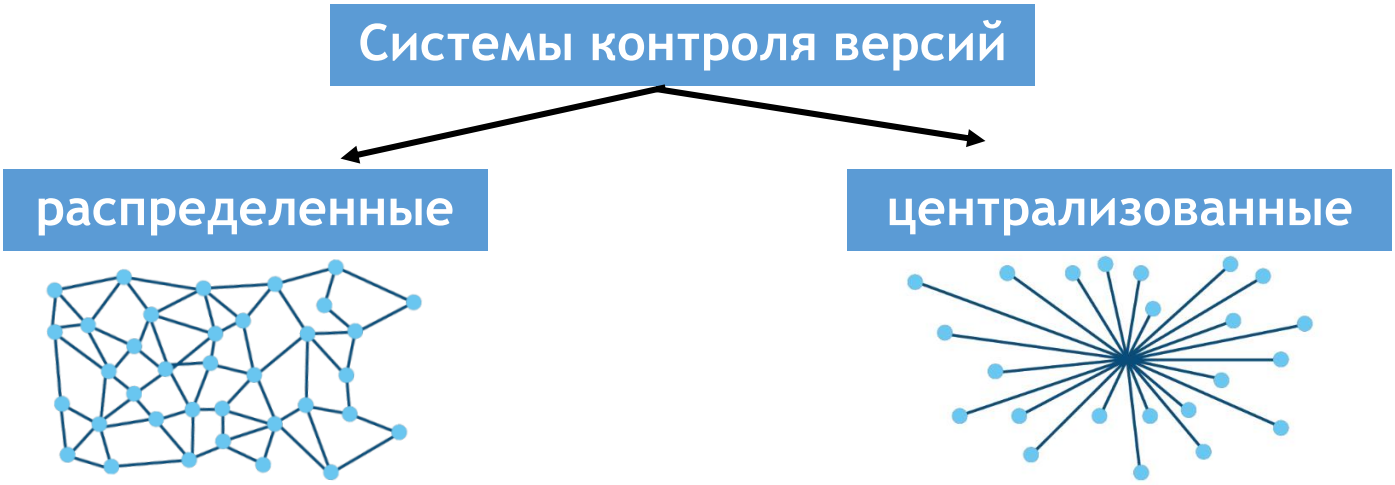
Система контроля версий (Version Control System)—это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Позволяет упорядоченно хранить несколько версий одного и того же файла, объединять изменения в файле, сделанные несколькими разработчиками, просматривать историю изменений, возвращаться обратно к предыдущим версиям и др.

Для чего она нужна? Во-первых, чтобы отследить изменения, произошедшие с проектом, со временем. Проще говоря, мы можем посмотреть, как менялись файлы программы, на всех этапах разработки и при необходимости вернуться назад и что-то отредактировать. Часто бывают ситуации, когда, во вполне себе работающий код, вам нужно внести определенные правки или улучшить какой-то функционал. Однако после внедрения нововведений, вы с ужасом понимаете, что все сломалось. У вас начинается судорожно дергаться глаз, а в воздухе повисает немой вопрос: “Что делать?” Без системы контроля версий, вам надо было бы долго напряженно просматривать код, чтобы понять, как было до того, как все перестало работать. С системой контроля версий же, все что нужно сделать - это откатиться на commit назад.

Во-вторых, она чрезвычайно полезна при одновременной работе нескольких специалистов, над одним проектом. Без системы контроля версий случится коллапс, когда разработчики, скопировав весь код из главной папки и сделав с ним задуманное, попытаются одновременно вернуть весь код обратно.

Типы систем контроля версий

Одними из первых были *локальные системы*, когда все изменения хранились на одном компьютере. По мере развития систем в них стали появляться *централизованные хранилища*, к которым участники обращались по сети. Потеря такого хранилища могла заблокировать работу команды или привести к исчезновению накопленной истории. Поэтому современные хранилища стараются *децентрализовать*: копия всех изменений находится у всех участников проекта. Централизованное хранилище может использоваться, но его потеря не приводит к потере истории. Любая копия может использоваться для восстановления центрального репозитория.



Системы контроля версий бывают локальными, централизованными или распределёнными.

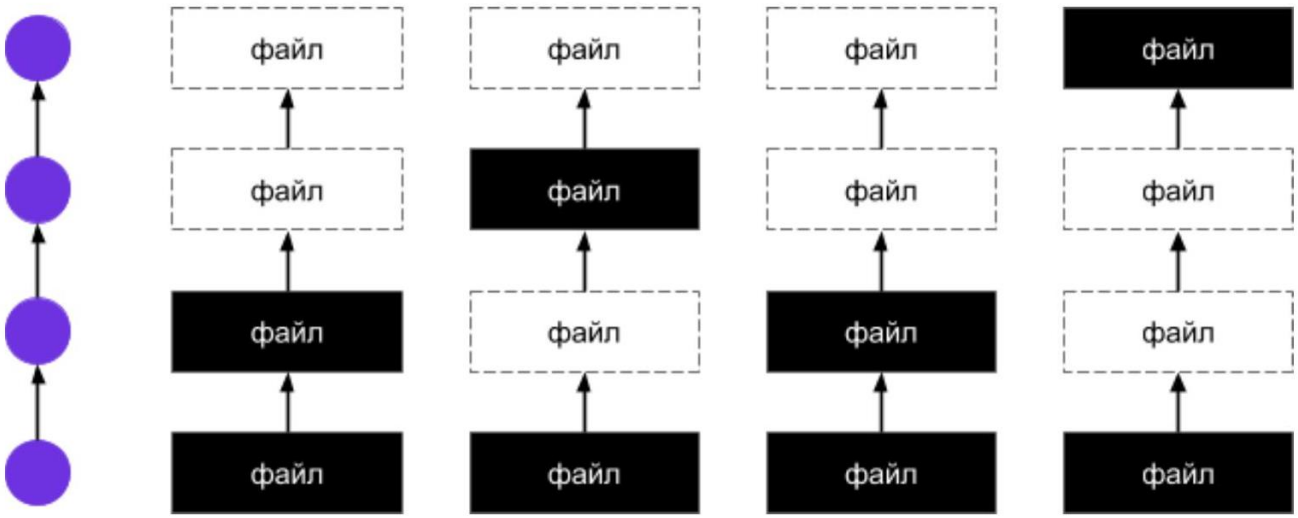
- **Локальная система** хранит файлы на одном устройстве, централизованная использует общий сервер, а распределённая — общее облачное хранилище и локальные

устройства участников команды. В локальной системе удобно работать с большими проектами, но сложно взаимодействовать с удалённой командой.
- В **централизованной системе** налажена удалённая работа, но всё привязано к одному серверу. Любой сбой или взлом может повредить файлы проекта.
- В **распределённой системе** налажена удалённая работа. Если с файлами основного репозитория что-то случится — проект легко восстановить из копии любого участника команды.

Существует множество систем контроля версий: CVS, Subversion, SVN. Однако сейчас самая популярная — открытая и свободная система Git. Она создавалась для обеспечения командной работы над ядром операционной системы Linux.

Особенности системы контроля версий Git

Git — это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ, но можно использовать его для любых файлов). Изначально Git был создан Линусом Торвальдсом при разработке ядра Linux.



Однако инструмент так понравился разработчикам, что в последствии, он получил широкое распространение и его стали использовать в других проектах. С его помощью можно сравнивать, анализировать, редактировать изменения и возвращаться назад к последнему сохранению. Этот процесс называется контролем версий.

Git является распределенным, то есть не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в директориях на жестком диске, которые называются **репозиторием**. Копию репозитория можно хранить онлайн - это сильно облегчает работу над одним проектом для нескольких людей. Для этого используются сайты вроде github и bitbucket.

Система контроля версий Git имеет ряд особенностей, которые ее отличают от других систем.

Хранение файлов

Git не хранит полные копии состояний, вместо этого реализуется своеобразная файловая система:

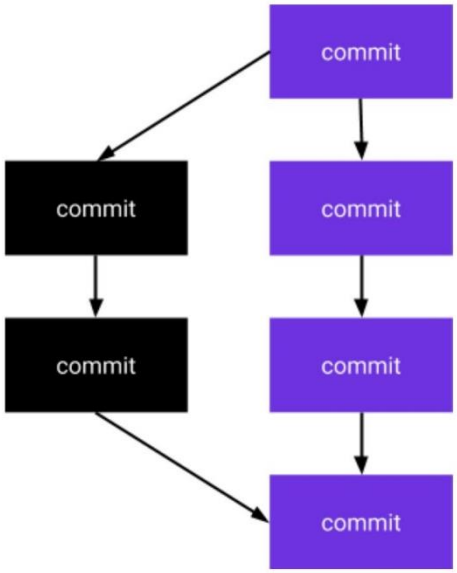
- загружаются только изменившиеся файлы. Вместо файлов, не претерпевших изменения,
- используются ссылки на более ранние версии.

Файлы, для которых хранится физическая копия, представлены в виде черных прямоугольников.

Белые прямоугольники — ссылка на предыдущий файл. Таким образом, несмотря на то, что хранится четыре снимка, в каждом из которых по четыре файла, вместо 16 файлов реально существует лишь 8. Снимок системы называют *коммитом*.(*commit*).

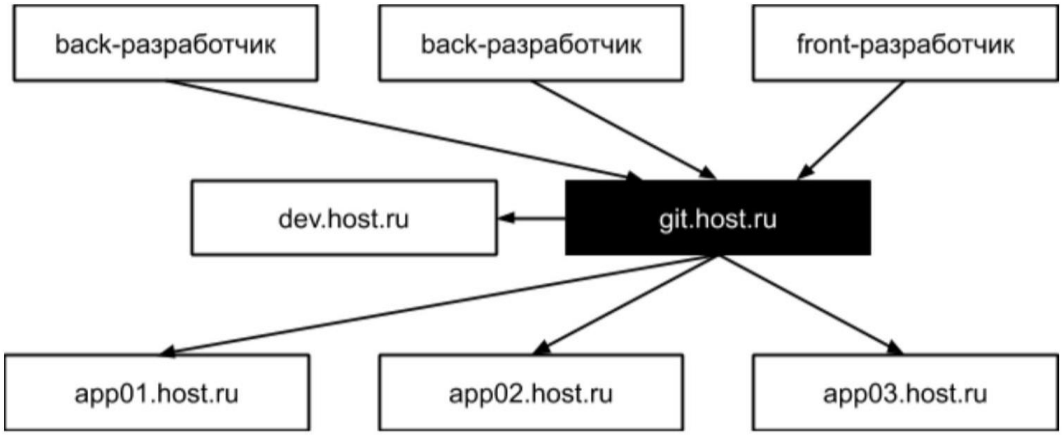
Ветки

Хронологическая последовательность коммитов называется веткой (*branch*). Ветки могут разделяться, идти параллельно и сливаться. Если необходимо реализовать какой-то функционал, можно разделить ветку от основного проекта и вести разработку в ней. При этом изменения не будут отражаться на коде до тех пор, пока ветка с разрабатываемым функционалом не будет слита в основную ветку проекта.



Git — распределенная система. Допускается наличие центрального репозитория, выделенного сервера, через который участники команды могут обмениваться изменениями. Однако каждый участник имеет локальную копию всех файлов, веток и истории правок.

На рисунке черным прямоугольником обозначен центральный репозиторий, в который участники могут отправлять свои изменения и скачивать обновления от других участников команды.



Git-репозиторий может выступать источником кода при развертывании приложения. Если вдруг центральный репозиторий выходит из строя, заменить его может любой хост, на котором использовалась система контроля версий. Таким образом, выход из строя центрального репозитория не парализует работу отдельных участников и не приводит к безвозвратной потере данных — репозиторий может быть восстановлен из любой локальной копии.

Большинство операций выполняется локально

Наличие центрального репозитория не является обязательным условием: Git можно использовать на локальной машине, в одиночной разработке. Более того, для большинства операций достаточно локальных файлов, Git не будет тормозить из-за сетевых задержек: все операции, которые можно провести без сети, будут проведены без сетевых обращений.

Git следит за изменениями

Когда вы производите какие-то действия с файлами, Git практически всегда добавляет новые данные в свою базу. Удалить что-то почти невозможно: как в Wiki-системе, что-то, добавленное один раз, навсегда остается в истории системы. Вы в любой момент можете откатиться в любую точку и получить копию даже удаленных файлов.

Документация

В ходе обучения не стесняйтесь прибегать к документации на официальном сайте **https://git-scm.com/book/en/v2**, которая оформлена в виде книги ProGit. Книга переведена на русский язык **https://git-scm.com/book/ru/v2** и доступна для загрузки в форматах PDF и mobi. В отличие от документации других компьютерных систем, книга очень хорошо написана: живым, понятным языком.

Установка

Установить git на свою машину очень просто:

- Linux — нужно просто открыть терминал и установить приложение при помощи пакетного менеджера вашего дистрибутива. Для Ubuntu команда будет выглядеть следующим образом:

  ```
  sudo apt-get install git
  ```

- Windows — скачать git for windows с официального сайта: https://git-scm.com/download/win git for windows, так как он содержит и клиент с графическим интерфейсом, и эмулятор bash.
- OS X — проще всего воспользоваться homebrew. После его установки запустите в терминале:

  ```
  brew install gi
  ```

Настройка

Итак, мы установили git, теперь нужно добавить немного настроек. Есть довольно много опций, с которыми можно играть, но мы настроим самые важные: наше имя пользователя и адрес электронной почты. Откройте терминал и запустите команды:

```
git config --global user.name "My Name"
```

```
git config --global user.email myEmail@example.com
```

Теперь каждое наше действие будет отмечено именем и почтой. Таким образом, пользователи всегда будут в курсе, кто отвечает за какие изменения — это вносит порядок.

Git хранит весь пакет конфигураций в файле .gitconfig, находящемся в вашем локальном каталоге. Чтобы сделать эти настройки глобальными, то есть применимыми ко всем проектам, необходимо добавить флаг –global. Если вы этого не сделаете, они будут распространяться только на текущий репозиторий. Для того, чтобы посмотреть все настройки системы, используйте команду:

```
git config --list
```

Для удобства и легкости зрительного восприятия, некоторые группы команд в git можно выделить цветом, для этого нужно прописать в консоли:

```
git config --global color.ui true
git config --global color.status auto
git config --global color.branch auto
```

Если вы не до конца настроили систему для работы, в начале своего пути - не беда. Git всегда подскажет разработчику, если тот запутался, например:

1. Команда `git --help` - выводит общую документацию по git
2. Если введем `git log --help` - он предоставит нам документацию по какой-то определенной команде (в данном случае это - log)
3. Если вы вдруг сделали опечатку - система подскажет вам нужную команду
4. После выполнения любой команды - отчитается о том, что вы натворили
5. Также Гит прогнозирует дальнейшие варианты развития событий и всегда направит разработчика, не знающего, куда двигаться дальше

Тут стоит отметить, что подсказывать система будет на английском, но не волнуйтесь, со временем вы изучите несложный алгоритм ее работы и будете разговаривать с ней на одном языке.

Создание нового репозитория

Как мы отметили ранее, git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нам нужно открыть терминал, зайти в папку нашего проекта и выполнить команду init. Это включит приложение в этой конкретной папке и создаст скрытую директорию .git, где будет храниться история репозитория и настройки. Создайте на рабочем столе папку под названием git_exercise. Для этого в окне терминала введите:

```
$ mkdir Desktop/git_exercise/
$ cd Desktop/git_exercise/
$ git init
```

Командная строка должна вернуть что-то вроде:

```
Initialized empty Git repository in /home/user/Desktop/git_exercise/.git/
```

Это значит, что наш репозиторий был успешно создан, но пока что пуст. Теперь создайте текстовый файл под названием hello.txt и сохраните его в директории git_exercise.

Определение состояния

status — это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли информация на нём, нет ли чего-то нового, что поменялось, и так далее. Запуск git status на нашем свежесозданном репозитории должен выдать:

```
$ git status
On branch master
Initial commit
Untracked files:
(use "git add ..." to include in what will be committed)
hello.txt
```

Сообщение говорит о том, что файл hello.txt неотслеживаемый. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специальным образом объявить.

Подготовка файлов

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на него файлы (или части файлов, или даже одиночные строчки) командой add и, наконец, коммитим все нужное в репозиторий (создаем слепок нужного нам состояния) командой commit.
В нашем случае у нас только один файл, так что добавим его:

```
$ git add hello.txt
```

Если нам нужно добавить все, что находится в директории, мы можем использовать

```
$ git add -A
```

Проверим статус снова, на этот раз мы должны получить другой ответ:

```
$ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached ..." to unstage)
new file: hello.txt
```

Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки — в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

Фиксация изменений

Как сделать коммит

Представим, что нам нужно добавить пару новых блоков в html-разметку (index.html) и стилизовать их в файле style.css. Для сохранения изменений, их необходимо закоммитить. Но сначала, мы должны обозначить эти файлы для Гита, при помощи команды git add, добавляющей (или подготавливающей) их к коммиту. Добавлять их можно по отдельности:

```
git add index.html
git add css/style.css
```

или вместе - всё сразу:

```
git add .
```

Конечно добавлять всё сразу удобнее, чем прописывать каждую позицию отдельно. Однако, тут надо быть внимательным, чтобы не добавить по ошибке ненужные элементы. Если же такое произошло изъять оттуда ошибочный файл можно при помощи команды

```
git reset:
git reset css/style.css
```

Теперь создадим непосредственно сам коммит

```
git commit -m 'Add some code'
```

Флажок -m задаст commit message - комментарий разработчика. Он необходим для описания закоммиченных изменений. И здесь работает золотое правило всех комментариев в коде: «Максимально ясно, просто и содержательно обозначь написанное!»

Как посмотреть коммиты

Для просмотра все выполненных фиксаций можно воспользоваться историей коммитов. Она содержит сведения о каждом проведенном коммите проекта. Запросить ее можно при помощи команды:

```
git log
```

В ней содержиться вся информация о каждом отдельном коммите, с указанием его хэша, автора, списка изменений и даты, когда они были сделаны. Отследить интересующие вас операции в списке изменений, можно по хэшу коммита, при помощи команды git show :

```
git show hash_commit
```

Ну а если вдруг нам нужно переделать commit message и внести туда новый комментарий, можно написать следующую конструкцию:

```
git commit --amend -m 'Новый комментарий'
```

В данном случае сообщение последнего коммита перезапишется. Но злоупотреблять этим не стоит, поскольку эта операция опасная и лучше ее делать до отправки коммита на сервер.

Удаленные репозитории

Сейчас наш коммит является локальным — существует только в директории .git на нашей файловой системе. Несмотря на то, что сам по себе локальный репозиторий полезен, в большинстве случаев мы хотим поделиться нашей работой или доставить код на сервер, где он будет выполняться.

Что такое удаленный репозиторий

Репозиторий, хранящийся в облаке, на стороннем сервисе, специально созданном для работы с git имеет ряд преимуществ. Во-первых - это своего рода резервная копия вашего проекта, предоставляющая возможность безболезненной работы в команде. А еще в таком репозитории можно пользоваться дополнительными возможностями хостинга. К примеру - визуализацией истории или возможностью разрабатывать вашу программу непосредственно в веб-интерфейсе.

Клонирование
Клонирование - это когда вы копируете удаленный репозиторий к себе на локальный ПК. Это то, с чего обычно начинается любой проект. При этом вы переносите себе все файлы и папки проекта, а также всю его историю с момента его создания. Чтобы склонировать проект, сперва, необходимо узнать где он расположен и скопировать ссылку на него. На наших занятиях по разработке баз данных мы будем использовать адрес https://github.com/SergeyGris/DBdev.git, но советую, попробовать создать свой репозиторий в GitHub, BitBucket или любом другом сервисе:

```
git clone https://github.com/SergeyGris/DBdev
```

При клонировании в текущий каталог, там будет создана папка, в которую поместятся все проектные файлы и скрытая директория .git, с самим репозиторием, или с необходимой информацией о нем. В такой ситуации, для клонируемого репозитория, по умолчанию, будет создана папка с одноименным названием, но его можно залить и в другую директорию, например:

```
git clone https://github.com/SergeyGris/DBdev/new-folder
```

Подключение к удаленному репозиторию

Чтобы загрузить что-нибудь в удаленный репозиторий, сначала нужно к нему подключиться. Регистрация и установка может занять время, но все подобные сервисы предоставляют хорошую документацию.
Чтобы связать наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URI репозитория на свой.

```
# This is only an example. Replace the URI with your own repository address.
$ git remote add origin https://github.com/SergeyGris/DBdev.git
```

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется origin.

3. Отправка изменений на сервер

Сейчас самое время переслать наш локальный коммит на сервер. Этот процесс происходит каждый раз, когда мы хотим обновить данные в удаленном репозитории. Команда, предназначенная для этого - push. Она принимает два параметра: имя удаленного репозитория (мы назвали наш origin) и ветку, в которую необходимо внести изменения (master — это ветка по умолчанию для всех репозиториев).

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/SergeyGris/DBdev.git
* [new branch] master -> master
```

Эта команда немного похожа на git fetch, с той лишь разницей, что при помощи fetch мы импортируем коммиты в локальную ветку, а применив push, мы экспортируем их из локальной в удаленную. Если вам необходимо настроить удаленную ветку используйте git remote. Однако пушить надо осторожно, ведь рассматриваемая команда перезаписывает безвозвратно все изменения. В большинстве случаев, ее используют, чтобы опубликовать выгружаемые локальные изменения в центральный репозиторий. А еще ее применяют для того, чтобы поделиться, внесенными в локальный репозиторий, нововведениями, с коллегами или другими удаленными участниками разработки проекта. Подытожив сказанное, можно назвать git push - командой выгрузки, а git pull и git fetch - командами загрузки или скачивания. После того как вы успешно запушили измененные данные, их необходимо внедрить или интегрировать, при помощи команды слияния git merge.
В зависимости от сервиса, который вы используете, вам может потребоваться аутентифицироваться, чтобы изменения отправились. Если все сделано правильно, то когда вы посмотрите в удаленный репозиторий при помощи браузера, вы увидите файл hello.txt

Запрос изменений с сервера

Если вы сделали изменения в вашем удаленном репозитории, другие пользователи могут скачать изменения при помощи команды pull.

```
$ git pull origin master
From https://github.com/SergeyGris/DBdev
* branch master -> FETCH_HEAD
Already up-to-date.
```

Так как новых коммитов с тех пор, как мы склонировали себе проект, не было, никаких изменений доступных для скачивания нет.

Как удалить локальный репозиторий

Вам не понравился один из ваших локальных Git-репозиториев и вы хотите стереть его со своей машины. Для этого вам всего лишь надо удалить скрытую папку «.git» в корневом каталоге репозитория. Сделать это можно 3 способами:

1. Проще всего вручную удалить эту папку «.git» в корневом каталоге «Git Local Warehouse».
2. Также удалить, не устраивающий вас, репозиторий можно на github. Открываете нужный вам объект и переходите в пункт меню Настройки. Там, прокрутив ползунок вниз, вы

попадете в зону опасности, где один из пунктов будет называться «удаление этого хранилища».

3. Последний метод удаления локального хранилища через командную строку, для этого в терминале необходимо ввести следующую команду:

```
cd repository-path/
rm -r .git
```

Ветвление

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

- Уже рабочая, стабильная версия кода сохраняется.
- Различные новые функции могут разрабатываться параллельно разными программистами.
- Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
- В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

Создание новой ветки

Основная ветка в каждом репозитории называется master. Чтобы создать еще одну ветку, используем команду branch <name>

```
$ git branch amazing_new_feature
```

Это создаст новую ветку, пока что точную копию ветки master.

Переключение между ветками

Сейчас, если мы запустим branch, мы увидим две доступные опции:

```
$ git branch
amazing_new_feature
* master
```

master — это активная ветка, она помечена звездочкой. Но мы хотим работать с нашей “новой потрясающей фичей”, так что нам понадобится переключиться на другую ветку. Для этого воспользуемся командой checkout, она принимает один параметр — имя ветки, на которую необходимо переключиться.

```
$ git checkout amazing_new_feature
```

В Git ветка — это отдельная линия разработки. Git checkout позволяет нам переключаться как между удаленными, так и меду локальными ветками. Это один из способов получить доступ к работе коллеги или соавтора, обеспечивающий более высокую продуктивность совместной работы. Однако тут надо помнить, что пока вы не закомитили изменения, вы не сможете

переключиться на другую ветку. В такой ситуации нужно либо сделать коммит, либо отложить его, при помощи команды git stash, добавляющей текущие незакоммиченные изменения в стек изменений и сбрасывающей рабочую копию до HEAD'a репозитория.

Слияние веток

Наша “потрясающая новая фича” будет еще одним текстовым файлом под названием feature.txt. Мы создадим его, добавим и закоммитим:

```
$ git add feature.txt
$ git commit -m "New feature complete.”
```

Изменения завершены, теперь мы можем переключиться обратно на ветку master.

```
$ git checkout master
```

Теперь, если мы откроем наш проект в файловом менеджере, мы не увидим файла feature.txt, потому что мы переключились обратно на ветку master, в которой такого файла не существует. Чтобы он появился, нужно воспользоваться merge для объединения веток (применения изменений из ветки amazing_new_feature к основной версии проекта).

```
$ git merge amazing_new_feature
```

Теперь ветка master актуальна. Ветка amazing_new_feature больше не нужна, и ее можно удалить.

```
$ git branch -d awesome_new_feature
```

Если хотите создать копию удаленного репозитория - используйте git clone. Однако если вам нужна только определенная его ветка, а не все хранилище - после git clone выполните следующую команду в соответствующем репозитории:

```
git checkout -b <имя ветки> origin/<имя ветки>
```

После этого, новая ветка создается на машине автоматически.

Как удалять ветки в Git?

Бывают ситуации, когда после слива каких-то изменений из рабочей ветки в исходную версию проекта, ее, по правилам хорошего тона, необходимо удалить, чтобы она более не мешалась в вашем коде. Но как это сделать?
Для локально расположенных веток существует команда:

```
git branch -d local_branch_name
```

где флажок -d являющийся опцией команды git branch - это сокращенная версия ключевого слова --delete, предназначенного для удаления ветки, а local_branch_name – название ненужной нам ветки.
Однако тут есть нюанс: удалить текущую ветку, в которую вы, в данный момент просматриваете - нельзя. Если же вы все-таки попытаетесь это сделать, система отругает вас и выдаст ошибку с таким содержанием:

```
Error: Cannot delete branch local_branch_name checked out at название_директории
```

Так что при удалении ветвей, обязательно переключитесь на другой branch.

Дополнительно

Настройка .gitignore

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в git add -A при помощи файла .gitignore

1. Создайте вручную файл под названием .gitignore и сохраните его в директорию проекта.
2. Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки.
3. Файл .gitignore должен быть добавлен, закоммичен и отправлен на сервер, как любой другой файл в проекте.

Вот хорошие примеры файлов, которые нужно игнорировать:

- Логи
- Артефакты систем сборки
- Папки node_modules в проектах node.js
- Папки, созданные IDE, например, Netbeans или IntelliJ
- Разнообразные заметки разработчика.

Файл .gitignore, исключающий все перечисленное выше, будет выглядеть так:

```
*.log
build/
node_modules/
.idea/
my_notes.txt
```

Символ слэша в конце некоторых линий означает директорию (и тот факт, что мы рекурсивно игнорируем все ее содержимое). Звездочка, как обычно, означает шаблон.