[Модуль 9. Основные команды GIT](https://lms.skillfactory.ru/courses/course-v1:SkillFactory+INTQAP+2022/course/" \l "block-v1:SkillFactory+INTQAP+2022+type@sequential+block@7ddfbf628c6248eeb4ed77c3182c81e5)

**9.1. Системы контроля версий**

**Архитектура тестов и версионность**

Представьте свою работу на проекте в роли тестировщика-автоматизатора. Проект имеет определённый набор функциональностей, которые реализует команда разработки. Вы как тестировщик-автоматизатор создаёте автотесты, верифицирующие работу функционала и разработанной системы в целом. Всё это происходит итеративно.

С каждой новой итерацией проекта вместе с запланированным к выпуску функционалом появляются новые тесты, добавляясь к уже разработанным. Тесты, разработанные ранее, могут обновляться.

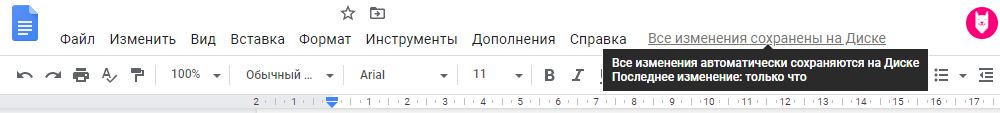
Вы уже убедились, прорабатывая задания к приложениям «Дом питомца» и *PetFriends*, что тестов может быть очень много даже для таких простых приложений. Для более серьёзных приложений счёт пойдёт на сотни и даже на тысячи, особенно там, где требуется автоматизация. Тесты не существуют сами по себе, они записываются в файлы.

Есть множество других файлов, которые делают систему тестов стройной, управляемой и масштабируемой. Например, файлы тестовых данных, которых также может быть огромное множество.

Есть такое понятие, как **архитектура автотестов** — это совокупность решений об организации системы автотестов, компонентах, библиотеках, интеграциях, используемых для создания автотестов.

Вся эта система постоянно меняется. Как правило, над ней работает не один тестировщик, а команда. Поэтому очень важно отслеживать изменения в файлах этой системы, чтобы знать, кто какие изменения вносил, и распределять разработку автотестов между членами команды. Помочь в этой непростой ситуации призвана система контроля версий.

Объяснить понятие версионности поможет аналогия с сервисами *Google*. Многие из вас работают с документами, таблицами и другими файлами облачного сервиса *Google*. С помощью истории изменений можно увидеть изменения, сделанные в файле в течении каждой сессии работы с ним. Переход в просмотр истории изменений позволяет увидеть изменения, сделанные каждым участником в рамках одной сессии. Можно выбрать версию файла на любой момент времени и восстановить её.

Рис. 9.1.1. История изменений в Google Документе

**Виды систем контроля версий**

Системы контроля версий обладают многими другими полезными свойствами, помимо версионности, необходимыми именно для разработки программного кода (неважно, это код веб-приложения или автотестов).

В этом модуле мы рассмотрим важность системы контроля версий на примере одной из них — *Git*.

**Система контроля версий** (далее СКВ) — это программное обеспечение, записывающее изменения в файлах, охваченных системой, в специальный файл или набор специальных файлов, и позволяющее вернуться позже к определённой версии отслеживаемого системой файла.

В целях демонстрации и отработки практических навыков в этом модуле мы будем использовать обычные текстовые файлы и файлы, которые вы создавали в модулях этого курса. Конечно, вы можете использовать контроль версий для любых типов файлов. Прежде чем мы перейдем к *Git*, рассмотрим преимущества и возможные недостатки некоторых **видов систем контроля версий**:

* + локальной,
  + централизованной,
  + децентрализованной.

**Локальные системы контроля версий**

Локальные системы контроля версий хранят информацию о всех изменениях, внесённых в файлы в базе данных, которая находится там же локально на компьютере. Основываясь на этих данных, система контроля версий воссоздаёт нужную версию файла (актуальную на определённый момент времени). В локальных СКВ обычно хранят список изменений, внесённых в файлы.

Среди **достоинств** таких систем нужно отметить:

* + Возможность восстановления данных из определённой версии (точно определяется по времени записи).
  + Высокая скорость выполнения восстановления (база данных чётко структурирована, поэтому сложностей при поиске не возникает, сетевая задержка отсутствует, поскольку данные хранятся непосредственно на рабочем компьютере).

**Недостатки** таких систем очевидны:

* + Возможность потери данных вследствие возникновения физических поломок оборудования.
  + Сложность совместной разработки. Одномоментно изменения в определённый файл могут быть внесены только одним пользователем.

Одна из первых известных систем контроля версий — это [SCCS (source code control system)](https://habr.com/ru/post/478752/). Опишем принципы её функционирования:

* + При добавлении файла для отслеживания в *SCCS* создаётся файл специального типа, который называется **s-файл** или **файл истории**. Он именуется как исходный файл, только с префиксом s., и хранится в подкаталоге *SCCS*. В момент создания файл истории содержит начальное содержимое исходного файла, а также некоторые метаданные, помогающие отслеживать версии. Здесь хранятся контрольные суммы для гарантии, что содержимое не было изменено. Содержимое файла истории не сжимается и не кодируется, как в системах контроля версий следующих поколений.
  + Содержимое исходного файла, хранящееся в файле истории, можно извлечь в рабочий каталог для просмотра, компиляции или редактирования.
  + В файл истории можно внести такие изменения, как добавление строк, изменение и удаление, что увеличивает его номер версии.
  + Последующие добавления файла хранят только дельты или изменения, а не всё его содержимое. Это уменьшает размер файла истории. Каждая дельта сохраняется внутри файла истории в структуре под названием дельта-таблица.
  + Фактическое содержимое файла более или менее копируется дословно, со специальными управляющими последовательностями для маркировки начала и конца разделов добавленного и удалённого содержимого. Поскольку файлы истории *SCCS* не используют сжатие, они обычно имеют больший размер, чем фактический файл, в котором отслеживаются изменения.

*SCCS* использует метод под названием **чередующиеся дельты** (*interleaved deltas*), который гарантирует постоянное время извлечения независимо от давности извлечённой версии, то есть более старые версии извлекаются с той же скоростью, что и новые.

Важно отметить, что все файлы отслеживаются и регистрируются отдельно. У каждого отслеживаемого файла свой файл истории, в котором хранится его история изменений. В общем случае это означает, что номера версий различных файлов в проекте обычно не совпадают друг с другом. Однако эти версии можно согласовать путём одновременного редактирования всех файлов в проекте (даже не внося в них реальные изменения) и одновременного добавления всех файлов.

Когда файл извлекается для редактирования в *SCCS*, на него ставится блокировка, так что его никто больше не может редактировать. Это предотвращает перезапись изменений другими пользователями, но также ограничивает разработку, потому что в каждый момент времени только один пользователь может работать с данным файлом.

Другой популярной локальной системой контроля версий была **RCS** (*revision control system*), которая и сегодня поставляется со многими компьютерами.

У *RCS* много общего с *SCCS*, в том числе:

* + Ведение версий отдельно для каждого файла.
  + Нельзя сгруппировать изменения в нескольких файлах.
  + Отслеживаемые файлы не могут одновременно изменяться несколькими пользователями.
  + Нет поддержки сети.
  + Версии каждого отслеживаемого файла хранятся в соответствующем файле истории.

Технология работы с файлами в системе *RCS* похожа на систему работы с файлами в системе *SCCS*. Но, в отличие от *SCCS*, для хранения изменений *RCS* использует схему **обратных дельт** (*reverse-delta*). При добавлении файла полный снимок его содержимого сохраняется в файле истории. Когда файл изменяется и возвращается снова, вычисляется дельта на основе существующего содержимого файла истории. Старый снимок отбрасывается, а новый сохраняется вместе с дельтой, чтобы вернуться в старое состояние. Это называется обратной дельтой, так как для извлечения более старой версии *RCS* берёт последнюю версию и последовательно применяет дельты до тех пор, пока не достигнет нужной версии.

Этот метод позволяет очень быстро извлекать текущие версии, так как всегда доступен полный снимок текущей ревизии. Однако чем старше версия, тем больше времени занимает проверка, потому что нужно проверить всё больше дельт.

## ****Централизованные системы контроля версий****

В централизованных системах появляется удалённый репозиторий, который находится на сервере в Сети. Проекты, над которыми работают разработчики, импортируются в локальные репозитории. Таким образом появляется возможность работы над одними и теми же файлами одновременно.

Архитектурно централизованные системы имеют существенные различия. Например, система **CVS** (Concurrent version system) для каждого файла хранит свой файл истории, подобно RCS, хотя использует при этом сжатие.

В системе **SVN** (Subversion) используется база данных, хранящая информацию об изменениях файлов и папок. Также в этой системе можно отслеживать наборы связанных изменений в разных файлах вместе, как в сгруппированном блоке, а не отдельно для каждого файла, как в локальных системах и в CVS.

Основное **преимущество централизованных систем** — возможность распределённой разработки и полный контроль администраторов над тем, кто и что может делать. Администрировать централизованную СКВ гораздо проще.

**Недостаток централизованных систем** такой же, как и у локальных — единая точка отказа, которой является центральный сервер. Если этот сервер выйдет из строя, то, пока он недоступен, никто не сможет ни взаимодействовать друг с другом, ни сохранить изменения в файле, с которым он работает. Если жёсткий диск, где хранится центральная база данных, окажется повреждён, а резервных копий не будет, то вся история проекта (за исключением единичных снимков репозитория, которые сохранились на локальных машинах разработчиков) будет потеряна.

## ****Децентрализованные системы контроля версий****

С представителем децентрализованной СКВ — Git — мы будем работать в этом модуле. Если ввести запрос: «Что такое Git» в поисковике, большинство источников ответят так: «Git (от англ. — Global Information Tracker) — это распределённая система управления версиями».

Мы дополним ответ — GIT является распределённой, децентрализованной системой контроля версий. Центрального репозитория не существует: все копии создаются равными, что резко отличается от централизованных СКВ, где работа основана на добавлении и извлечении файлов из центрального репозитория. Это означает, что разработчики могут обмениваться изменениями друг с другом непосредственно перед объединением своих изменений в официальную ветвь.

Разработчики могут вносить свои изменения в локальную копию репозитория без ведома других репозиториев. Это допускает фиксацию изменений без подключения к сети или интернету. Разработчики могут работать локально в автономном режиме, пока не будут готовы поделиться своей работой с другими. В этот момент изменения отправляются в другие репозитории для проверки, тестирования и развёртывания.

Когда файл добавляется для отслеживания в Git, он сжимается с помощью алгоритма сжатия zlib. Результат шифруется с помощью хэш-функции SHA-1. Это даёт уникальный хэш, который соответствует конкретно содержимому в этом файле. Git хранит файл в базе объектов. Имя файла — это сгенерированный хэш, а файл содержит сжатый контент. Данные файлы называются **блобами** и создаются каждый раз при добавлении в репозиторий нового файла или изменённой версии существующего файла.

**Отличие Git от других СКВ**

Главное отличие *Git* от централизованных и локальных СКВ: система делает снимок того, как выглядит каждый файл в данный момент, и сохраняет ссылку на этот снимок каждый раз, когда пользователь делает фиксацию изменений в своём проекте. Если файл не менялся, то *Git* не запоминает этот файл вновь, а просто сохраняет ссылку на предыдущую версию этого файла, который был уже сохранён.

Другое очень важное отличие *Git* от других типов СКВ — это автономность. Большинство операций выполняется локально, и подключение к удалённому центральному серверу необязательно. Копия репозитория хранится у вас локально, можно очень быстро получить любую версию любого файла. Если изменения зафиксированы и синхронизированы с репозиторием на сервере, то потерять данные достаточно сложно, в особенности если над проектом работает целая команда.

**Основные состояния Git**

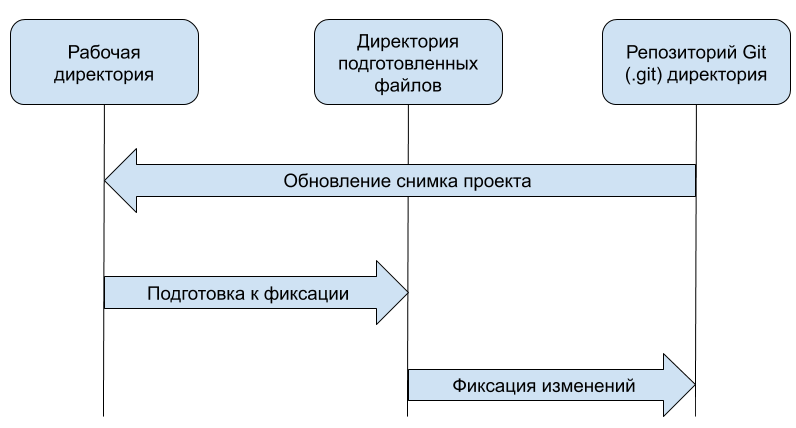
*Git* имеет три основных состояния, в которых могут находиться файлы репозитория:

* 1. Зафиксированное.
  2. Изменённое.
  3. Подготовленное.

**Зафиксированное состояние** означает, что файл уже сохранён в вашей локальной базе. Как только какие-то изменения вносятся в файл, то файл переходит в изменённое состояние.

Для того чтобы зафиксировать файл, необходимо перевести его в подготовленное состояние. Это происходит при выполнении определённой операции, о которой мы расскажем в уроке 9.4. После того как файлы подготовлены, можно провести фиксацию изменений, что также выполняется специальной командой.

Диаграмма перехода от состояния к состоянию показана на рисунке 9.1.2.

Рис. 9.1.2. Диаграмма перехода от состояния к состоянию в *Git*

Технически, когда вы создаёте *Git* репозиторий в файловой системе своего компьютера, то помещаете под версионный контроль файлы определённого вами каталога. Внутри этого каталога находится *Git* директория: место, где *Git* хранит метаданные и базу объектов вашего проекта. Это самая важная часть *Git*, и это та часть, которая копируется при клонировании репозитория с другого компьютера. Также есть рабочая директория *Git* и область подготовленных файлов.

**Рабочая директория** — это снимок версии проекта. *Git* распаковывает файлы в рабочую директорию из сжатой базы данных в *Git* директории.

**Область подготовленных файлов** — это файл, располагающийся в *Git* директории, в котором содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют «индекс».

Таким образом, процесс работы с файлами в *Git* выглядит следующим образом:

* 1. Изменение файлов в рабочей директории.
  2. Добавление файлов в индекс. Это значит, что тем самым добавляются их снимки в область подготовленных файлов.
  3. Фиксация изменений. Это значит, что снимки файлов из индекса сохраняются в *Git* директории.

## 9.2. Сервис GitHub

**Облачные решения для Git репозиториев**

В предыдущем уроке мы познакомились с видами систем контроля версий и подробно рассмотрели распределённую СКВ *Git*. Как вы уже поняли, при использовании *Git* для командной работы вам потребуется развернуть серверную часть распределённого репозитория и установить приложение на свой компьютер для управления локальной версией репозитория.

Есть множество облачных решений для развёртывания *Git* репозиториев, таких как *GitLab*, *GitHub*, *BitBucket*, *Microsoft Azur DevOps Repos*, *Microsoft VSTS*, *AWS CodeCommit* и другие.

Ряд этих сервисов предлагает решения для развёртывания *Git* сервисов и в вашей сети. Это значит, что можно установить, например, *GitHub* или *GitLab* на своём сервере и использовать его как центральное хранилище. На протяжении всего курса мы будем пользоваться [GitHub](https://github.com/) сервисом.

В этом уроке мы рассмотрим этот сервис и его свойства в деталях. В следующем уроке будет рассказано о том, как установить систему управления репозиториями на своём компьютере. Так мы построим *Git* инфраструктуру для работы над своими будущими проектами.

**Социальная сеть для разработчиков**

Почему *GitHub* называют «социальной сетью для разработчиков»? Пользователи *GitHub*, кроме непосредственного хранения кода своих проектов, могут организовывать командную разработку на своих проектах, комментировать изменения друг друга, отслеживать новости других пользователей. У программистов есть возможность объединять репозитории и выводить вклад каждого участника в определённый репозиторий в виде дерева.

**Ключевые особенности *GitHub*, помимо функций *Git* репозитория:**

* + Настройка вариантов совместной работы команды с файлами репозитория.
  + Публичные и личные репозитории для бесплатного аккаунта. Публичные репозитории доступны как для чтения, так и для записи другим авторизованным участникам данного сервиса, а также для чтения любому, имеющему ссылку. Приватные репозитории доступны только их владельцу.
  + Добавление в репозиторий файлов пользователями, имеющими доступ «по записи». Для этого используется интерфейс *Git*, установленный локально на компьютерах, или веб-интерфейс *GitHub*.
  + Клонирование репозитория целиком с помощью интерфейсов локально установленного *Git* и сервиса *GitHub*.
  + Создание организации в рамках своего аккаунта, добавление репозиториев и приглашение пользователей для участия. Гибкое назначение им прав на работу с файлами репозитория.
  + Функционирование в качестве полноценной системы управления проектами. Это значит, что пользователи могут создавать задачи, связывать задачи с конкретным репозиторием, определять процесс работы над задачами, привлекать других пользователей к проекту.
  + Наличие *API*, благодаря которому можно интегрировать *GitHub* в другие приложения, например, в системы управления проектами типа *JIRA*. Это позволяет построить интегрированную систему, в которой задачи в *JIRA* интегрируются с репозиториями в *GitHub*. Благодаря чему пользователи видят, какие изменения в коде связаны с той или иной задачей.
  + Подсветка синтаксиса для большинства языков.
  + Наличие справочной системы, *wiki*, для любого репозитория.
  + Платные аккаунты для работы с *GitHub* добавляют массу других весьма полезных функций.

В этом модуле мы затронем работу с небольшой частью функционала *GitHub*. Более глубоко узнать *GitHub* вы сможете самостоятельно, в том числе читая документацию после авторизации в этом сервисе.

## 9.3. Основы GIT

После установки *git-scm* в каталоге установки появится папка с именем *Git* (если это имя было выбрано при установке). В этой папке будут ссылки для запуска 3-х приложений:

* + *Git GUI* — это приложение с графическим интерфейсом для работы с *Git*.
  + *Git CMD* — командная строка *Git*, использующая интерпретатор командной строки *Windows*.
  + *Git Bash* — командная строка *Git*, использующая эмулятор интерпретатора командной строки *Bash* для *Linux*.
  + Запустите *Git CMD*.

**Внимание!** Важно убедиться, доступно ли приложение *Git cmd* на *Mac*. И его точно нет на *Linux* (там *Bash* интерпретатор). Все примеры этого модуля рассматриваются для ОС *Windows*. Для тех, кому удобнее работать с *Bash*, запустите *Git Bash*.

* + С помощью команды операционной системы *cd* перейдите к каталогу на своём компьютере, в который вы хотите клонировать репозиторий с *GitHub*. Например, *cd c:\projects\Skillfactory* (формат команды для *Windows*).
  + Наберите команду git clone, после которой укажите *URL* к вашему репозиторию на *GitHub*. Вы можете скопировать этот *URL* на *GitHub* и вставить его в командную строку. Например, командная строка может выглядеть так: *git clone https://github.com/fkosta/TestRepoSF00*. Нажатием на клавишу *Enter* завершите ввод. Репозиторий с *GitHub* будет клонирован на компьютер.

Рассмотрим содержимое папки *Skillfactory* (желательно создать папку с этим именем) на компьютере, в которую был клонирован репозиторий с *GitHub*. В этой папке находится ещё одна папка с именем клонированного репозитория. Внутри её содержимое представлено скрытой папкой .git и файлом *Readme*, клонированным из удалённого репозитория. Присутствие в папке директории .git говорит о том, что содержимое данной папки находится под версионным контролем.

**Содержимое директории .git**

Рассмотрим, что содержится в папках и файлах:

* + В **файле config** находятся настройки данного репозитория. Его содержимое представлено в текстовом формате.
  + **Файл HEAD** указывает на текущую ветку.
  + В **файле index** хранится содержимое индекса.
  + В **директории objects** находится, собственно, база данных объектов *Git*. Если открыть каталог *objects*, то в нём будут находиться каталоги, имена которых представлены двумя шестнадцатеричными числами, внутри которых будут файлы, имена которых представлены 38 шестнадцатеричными числами. Вместе имя каталога и файл образуют 40-разрядный хэш, взятый от имени файла и его содержимого.
  + В **директории refs** находятся ссылки на объекты коммитов в этой базе (ветки).
  + **Директория logs** хранит логи коммитов.
  + В **директории info** расположен файл с глобальными настройкам игнорирования файлов. Он позволяет исключить из работы файлы, которые вы не хотите помещать в .gitignore. Позднее мы поговорим о назначении файла .gitignore.
  + В **директории hooks** располагаются клиентские и серверные триггеры. Желающие прочитать про них могут обратиться к руководству по *Git.*

## ****Подведём итоги****

В этом уроке мы клонировали репозиторий GitHub на компьютер, подготовив тем самым основу для совместной работы над вашим проектом.

И,  да... мы не забыли, но действительно чуть позднее дали комментарий по команде  .gitignore.

## 9.4. Первые шаги работы с Git

## ****Команды для работы с репозиторием****

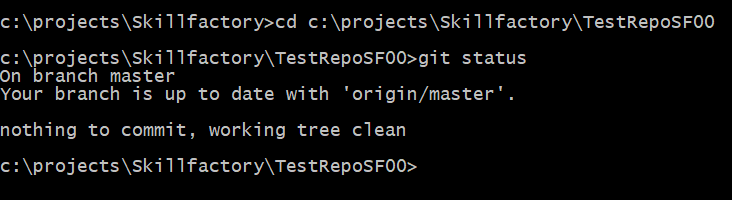
В предыдущем уроке мы рассмотрели первый и важный шаг в начале работы с репозиторием, который находится на GitHub. Вы клонировали репозиторий, созданный на GitHub, на свои компьютеры. Таким образом, вы получили два синхронизированных между собой репозитория.

Рассмотрим, какие команды для работы с репозиторием нам доступны локально.

**git status** — первая команда, с которой мы хотим вас познакомить — команда получения статуса файлов в репозитории.

Для выполнения команды сделайте следующее:

* 1. Запустите Git cmd.
  2. Перейдите в папку с репозиторием, который вы клонировали в предыдущем уроке.
  3. Используйте команду операционной системы cd для перехода в эту папку ([подробнее о командах ОС](http://cmd4win.ru/administrirovanie-computera/upravlenie-failami/29-cd)).
  4. Выполните команду git status.
  5. Прочитайте сообщения, которые вы увидите на экране (как на рисунке 9.4.1.)

Рис. 9.4.1. Выполнение команды git status

## ****Полученные сообщения****

* + **On branch master**. Сообщение означает, что в настоящий момент активна ветка master. Как правило, это основная ветка в репозитории.
  + **Your branch is up to date with origin/master**. Сообщение означает, что файлы в ветке master синхронизированы с веткой master на сервере. Origin — это и есть удалённый сервер.
  + **Working tree clean** дословно означает, что рабочий каталог не содержит изменённых файлов. Система контроля версий Git использует так называемую архитектуру трёх деревьев. Working tree связана с рабочей директорией Git, в которой содержатся рабочие версии файлов.
  + **Nothing to commit** значит, что все изменения были зафиксированы и нет новых изменений в файлах для фиксации изменений.

Команда **git status** — это основная команда для отслеживания изменений файлов, находящихся в репозитории. Команда выводит информацию о тех файлах репозитория, которые пока ещё не находятся под версионным контролем или не отслеживаются, и о тех файлах, которые были изменены, и изменения которых ещё не были зафиксированы.

## ****Алгоритм работы с файлами****

Рассмотрим алгоритм работы с файлами, помещёнными в репозиторий Git, с помощью команд.

Что вы можете делать с файлами:

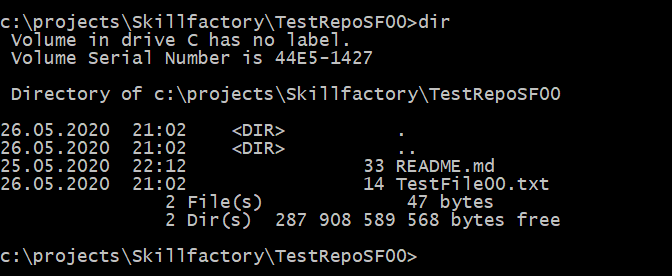
* + Добавлять, изменять и удалять файлы, находящиеся в вашем локальном репозитории.
  + Выполнять синхронизацию с репозиторием на удалённом сервере (в нашем случае с репозиторием на GitHub, с которого вы клонировали свой локальный репозиторий).

Посмотрим, как этот процесс происходит локально с контролем статуса файлов репозитория.

Добавим текстовый файл в каталог вашего репозитория, чтобы его можно было потом легко отредактировать. Можете просто создать текстовый файл в редакторе Wordpad с одной строкой.

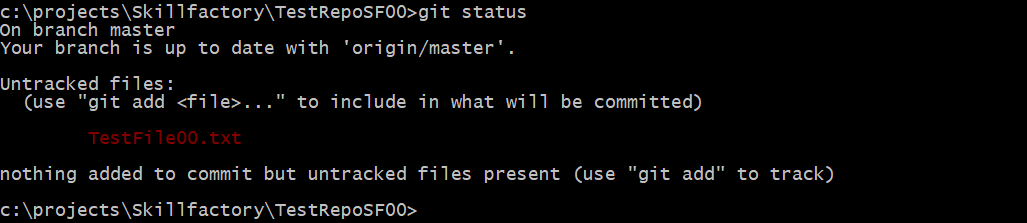
Если вы закрыли окно Git CMD, то откройте его снова и перейдите в каталог вашего репозитория.

На рисунке 9.4.2. вы видите содержимое каталога, выведенное командой dir. В каталоге два файла Readme и TextFile00.txt. Последний — это добавленный файл.

Рис. 9.4.2. Содержимое каталога, выведенное командой dir

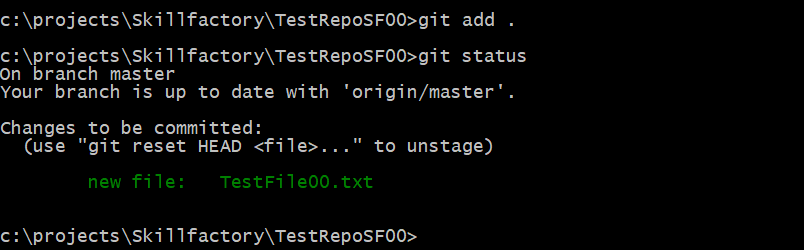
Выполните команду git status. Убедитесь, что появился информационный блок, где была выведена информация о неотслеживаемых файлах. Все новые файлы, которые добавляются в репозиторий, находятся в статусе неотслеживаемых или не находящихся под версионным контролем.

В сообщении есть подсказка, какую команду нужно выполнить, чтобы файл попал под версионный контроль и впоследствии в нём можно было зафиксировать изменения. Эта команда git add.

Рис. 9.4.3. Подсказка о выполнении команды git add

Используем команду git add. Она переносит файл в директорию подготовленных файлов. Эта команда принимает как параметр имя файлов, которые нужно перенести в подготовленные для фиксации. Также можно использовать метасимволы, например ., что означает «добавить все изменённые и неотслеживаемые файлы в подготовленные».

Выполним 2 команды git add . и git status. Последняя покажет, как изменилось состояние репозитория. Добавленный файл стал отслеживаемым, и он проиндексирован.

Рис. 9.4.4. Добавленный файл проиндексирован

**Команда git commit**

*Git* хранит данные в виде набора легковесных «снимков», известных как **коммиты**. Они хранят состояние файловой системы в определённый момент времени, а также указатель на предыдущий(-ие) коммит(-ы).

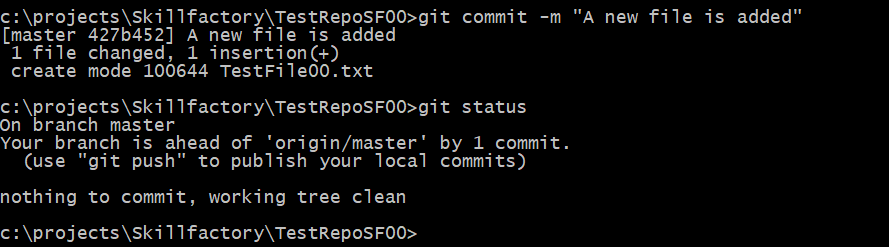
Для фиксации изменений необходимо выполнить команду git commit. Если эту команду подать без параметров, то откроется редактор, в котором вам нужно будет добавить сообщение к этому коммиту.

Можно добавить сообщение после команды git commit с помощью параметра -m, после которого включается сообщение в кавычках. На скриншоте 9.4.5. показано выполнение команды git commit и следом за ней команды git status. Под параметром -m команды на фиксацию изменений передано сообщение о коммите.

Команда выполнения коммита git commit вывела следующую информацию:

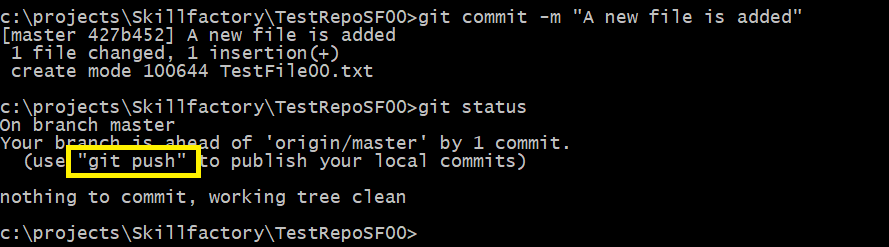
* + На какую ветку был выполнен коммит.
  + Какая контрольная сумма *SHA*-1 у этого коммита.
  + Сколько файлов было изменено.
  + Статистику по добавленным/удалённым строкам в этом коммите.

Команда git status выводит сообщение о том, что ветка *master* локального репозитория опережает ветку *master* на сервере на 1 коммит. И это не удивительно, потому как в удалённом репозитории на *GitHub* пока нет того файла, который вы добавили в репозиторий локально.

Рис. 9.4.5. Выполнение команд *git commit* и *git status*

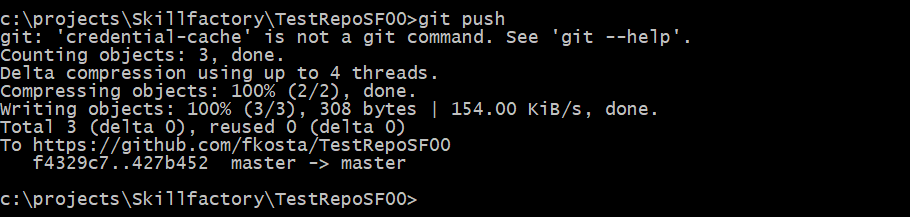
**Команда git push**

Вернёмся к рисунку 9.4.5.  Под сообщением о том, что ветка *master* локального репозитория опережает ветку *master*есть подсказка — команда, которую нужно использовать для публикации изменений на сервере (обратите внимание, что подсказки пишутся в скобках). Это команда git push.

Рис. 9.4.5. Подсказка о команде *git push*

Применим команду  git push.

После её ввода появится запрос на ввод имени пользователя и пароля на доступ к удалённому репозиторию. Введите имя пользователя и пароль, и содержимое локального репозитория синхронизируется с *GitHub*.

Рис. 9.4.6. Выполнение команды *git push*

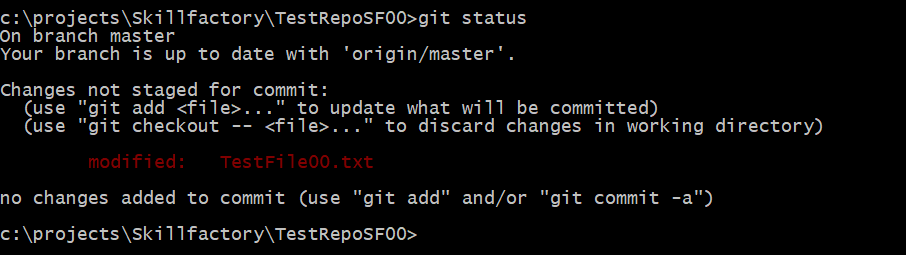
Ниже будет выведена статистика по переданным файлам. Команда git status, выполненная после успешного выполнения команды git push, покажет уже знакомые нам сообщения о том, что локальная ветка *master* синхронизирована с веткой *master* на удалённом сервере, все изменения были зафиксированы.

Откройте свой репозиторий на *GitHub*, убедитесь, что в нём появился новый файл с сообщением, которое вы передали при коммите.

Посмотрим, как будут меняться состояния изменяемых файлов. Откройте один из добавленных в репозиторий файлов и измените его. Например, добавьте в него ещё одну строку. Выполните команду git status.

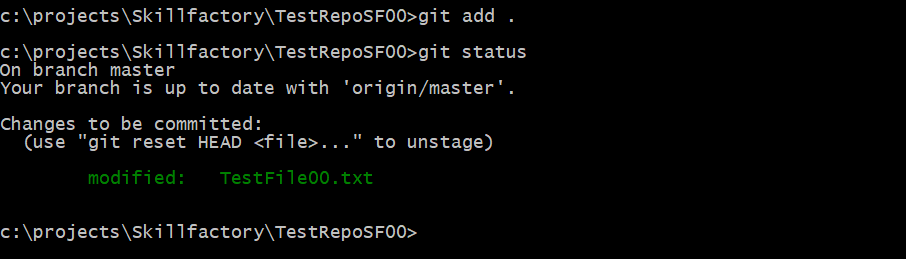
На экране появится информация о том, что:

* + Файл модифицирован.
  + Изменения не подготовлены для коммита.

Рис. 9.4.7. Сообщение о модификации файла

Выполните команду git add. Изменённый файл будет добавлен в индекс.

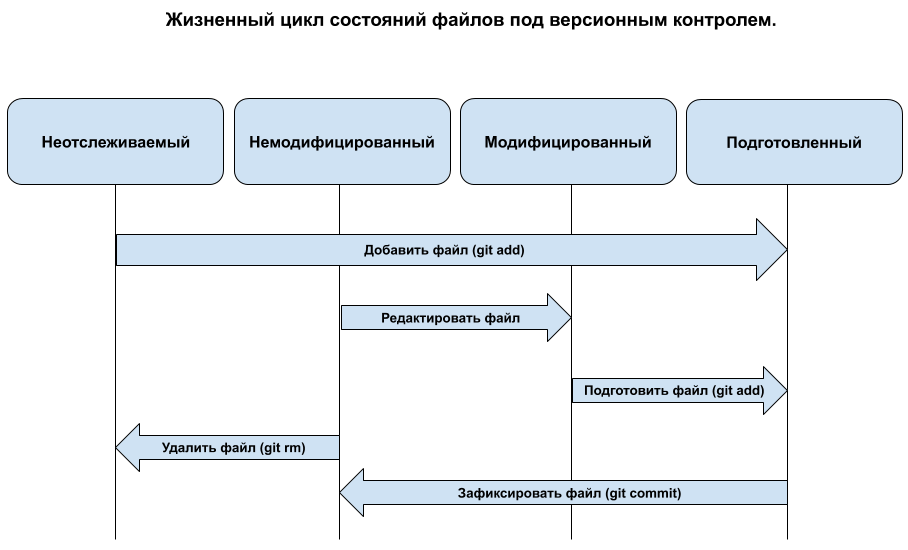
Введите команду git status и обратите внимание, что изменённый файл, добавляемый в подготовленные для коммита файлы, имеет признак «модифицированный» (modified). В случае, когда мы добавляли новый файл в репозиторий, он имел признак «новый» (new file).

Рис. 9.4.8. Сообщение о модифицированном (изменённом файле)

Выполните фиксацию изменений и отправку изменений на сервер, используя соответствующие команды git. Перейдите в ваш репозиторий на GitHub, убедитесь, что изменённый файл был успешно отправлен на сервер. Если вы использовали текстовый файл, то можете открыть его содержимое на GitHub и убедиться, что файл содержит все сделанные изменения.

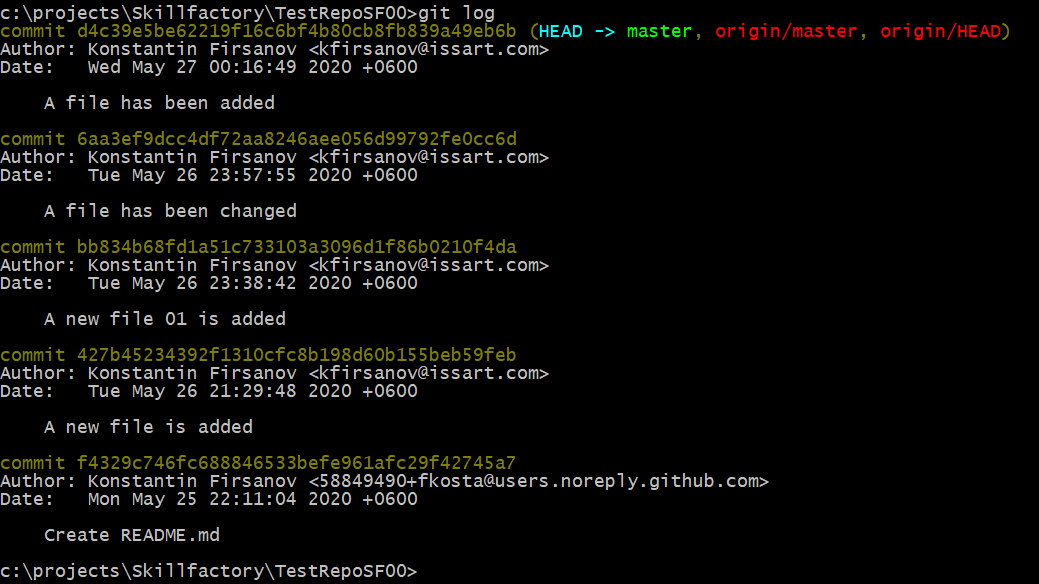
## ****Жизненный цикл состояний файлов****

Рассмотренный выше процесс можно представить следующей диаграммой, показывающей жизненный цикл новых и изменённых файлов в репозитории.

Рис. 9.4.9. Диаграмма жизненного цикла новых и изменённых файлов в репозитории

## ****Команда git log****

Познакомимся с командой, показывающей историю изменений. Эта команда git log. Выполним эту команду и посмотрим, что она выводит в консоль.

Рис. 9.4.10. Сообщения после выполнения команды git log

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

Из примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 контрольными суммами, именем и электронной почтой автора, датой создания и сообщением коммита. Напротив самого последнего коммита выводится информация о состоянии индекса:

**HEAD -> master, origin/master, origin/HEAD**

Это означает, что в настоящий момент индекс указывает на master ветку локального репозитория, который синхронизирован с master веткой удалённого репозитория.

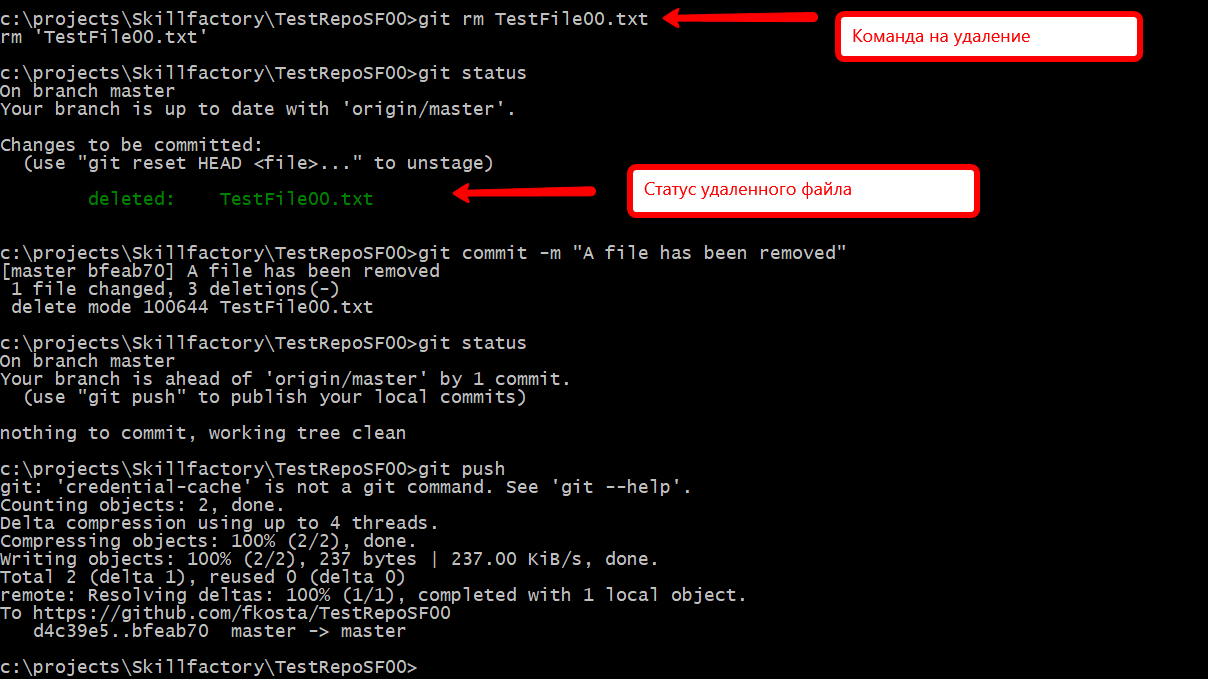
Команда git log имеет очень большое количество опций для поиска коммитов по разным критериям. Ознакомиться с опциями этой команды можно, прочитав [главу 2.3 книги «Pro Git»](https://git-scm.com/book/ru/v2/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git-%D0%9F%D1%80%D0%BE%D1%81%D0%BC%D0%BE%D1%82%D1%80-%D0%B8%D1%81%D1%82%D0%BE%D1%80%D0%B8%D0%B8-%D0%BA%D0%BE%D0%BC%D0%BC%D0%B8%D1%82%D0%BE%D0%B2).

## ****Команда git rm****

Что делать, когда необходимо удалить файлы из-под версионного контроля? Для того чтобы удалить файл из Git, необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса), а затем выполнить коммит. Это позволяет сделать команда git rm, которая также удаляет файл из вашего рабочего каталога.

Выберите один из файлов в репозитории и введите команду git rm, указав после неё имя файла. После этой команды необходимо зафиксировать изменения и опубликовать их на сервере. Просто так удалить файл, находящийся под версионным контролем, не получится, потому как есть снимок этого файла в репозитории и файл в рабочем каталоге будет восстановлен из этого снимка. Поэтому и необходимо использовать команду git rm для полного удаления файла.

На рисунке 9.4.11. показан процесс удаления файла. После команды git rm была выполнена команда git status, которая отметила файл как удалённый в информационном блоке о файлах для коммита. После команды git commit была выполнена команда git push для синхронизации изменений с репозиторием на GitHub. Переход в репозиторий на GitHub подтвердит, что файл, который удаляли локально, отсутствует.

Рис. 9.4.11. Процесс удаления файла

## 9.5. Ветвление кода

Для рассмотрения темы ветвления в *Git* давайте вернёмся ещё раз к материалу предыдущего урока. Мы рассмотрели команду git log. Давайте повторим процесс изменения файла в локальном репозитории.

Откройте *Git CMD*. Перейдите в каталог локального репозитория. Измените любой из файлов в локальном репозитории, добавьте его в индекс и выполните коммит. Теперь, не синхронизируя изменения с *GitHub,* выполните команду git log.

Обратите внимание на верхние строчки. В первой строке информация (HEAD -> master) означает, что индекс указывает на последний коммит ветки *master*. А строка (origin/master, origin/HEAD) указывает на то, что этот коммит — это последний коммит на удалённом сервере и то, что на этот коммит указывает индекс на удалённом сервере.

Рис. 9.5.1. Выполнение команды *git log*

Если теперь выполнить команду git push, то строка (HEAD -> master origin/master, origin/HEAD) будет указывать на последний коммит. Запомните пока эти указатели, мы к ним будем возвращаться по ходу урока.

**Веб-приложение PetFriends**

Рассмотрим ситуацию в команде разработки веб-приложения PetFriends. У вас есть набор чек-листов для тестирования приложения *PetFriends*. Для хранения чек-листов вы используете репозиторий, их много, приложение активно разрабатывается, появляются новые функциональности.

Представьте, что разработчики работают над определённой функциональностью. Постоянно добавляются новые требования, и вы решаете, что вам нужно создать ещё одну версию чек-листов, к которой вы перейдёте после того, как будет реализована новая функциональность. До того момента вы будете использовать текущую версию для тестирования приложения. Помочь эффективно решить эту задачу поможет концепция ветвления.

**Что такое ветка в Git**

**Ветка в Git** — это подвижный указатель на один из коммитов. Обычно ветка указывает на последний коммит в цепочке коммитов. Ветка берёт своё начало от какого-то одного коммита.

Создадим ветку в локальном репозитории. Откройте *Git CMD*. Перейдите в каталог локального репозитория. В настоящий момент вы находитесь в ветке *master*. Запомните, какие есть файлы в рабочем каталоге. Файлы, которые там есть, соответствуют ветке *master*.

Для создания новой ветки в *Git* используется команда git branch, после которой следует имя создаваемой ветки, например:

**git branch testbranch00**

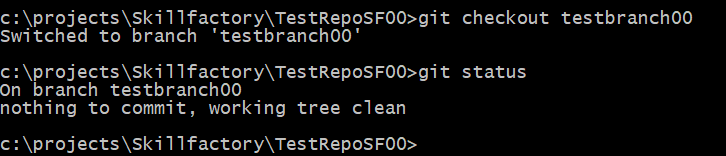
Вы можете использовать своё имя ветки. Далее по тексту будет использоваться имя ветки *testbranch00*. Если вы используете своё имя ветки, то вам необходимо заменить имя ветки *testbranch00* в примерах команд на имя ветки, которую создали вы.

Выполните следующие команды:

* 1. Для проверки состояния выполните команды git status и git log. Ничего необычного в выполнении этих команд вы не заметите.
  2. Команду git branch --list. Она выведет список веток, которые существуют в вашем локальном репозитории. Текущая ветка *master* будет выделена зелёным цветом.
  3. Команду git branch --list -v . Вы увидите информацию о последних коммитах каждой из веток. Так как коммитов в ветку *testbranch00* ещё не было, то она указывает на коммит в мастере, от которого была создана.

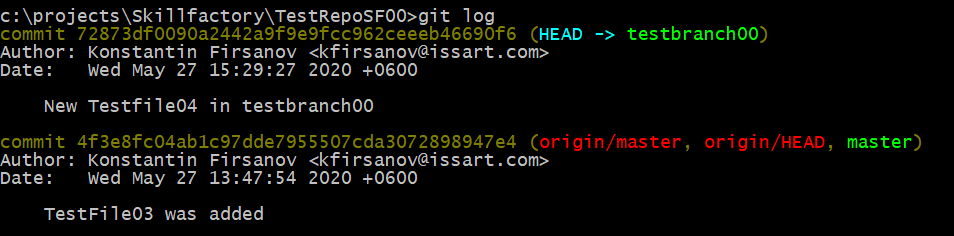
Рис. 9.5.2. Выполнение команд *git branch --list*

* 1. Команду git checkout необходимо выполнить для переключения на созданную ветку. Выполняя команду git checkout, укажите после неё имя ветки, на которую вы хотите переключиться. Например, git checkout testbranch00. Вы получите сообщение о переключении на ветку *testbranch00*.
  2. Выполните команду git status и вы увидите сообщение о том, что вы находитесь на ветке *testbranch00*.

Рис. 9.5.3. Выполнение команды *git checkout*

* 1. Выполним команду git log. Сейчас можно видеть, что индекс в локальном репозитории указывает на ветку *testbranch00*, а в удалённом — на ветку *master*. Давайте сделаем изменения в локальном репозитории, находясь в данный момент на ветке *testbranch00*.

1. Выполните команду git log и вы увидите, что на самый поздний коммит указывает индекс на ветке *testbranch00*. Следующий коммит — это коммит на удалённом сервере.

Рис. 9.5.4. Коммит показывает индекс на ветке *testbranch00*

1. Выполните команду операционной системы dir и запомните файлы, находящиеся в репозитории. Обратите внимание, что файл, который вы только что создали, находится в репозитории.

Команда git checkout сделала две вещи:

* 1. Она переместила указатель *HEAD* назад, на последний коммит ветки *master*.
  2. Вернула файлы в рабочем каталоге в то состояние, которое было сохранено в снимке, на который указывает ветка.

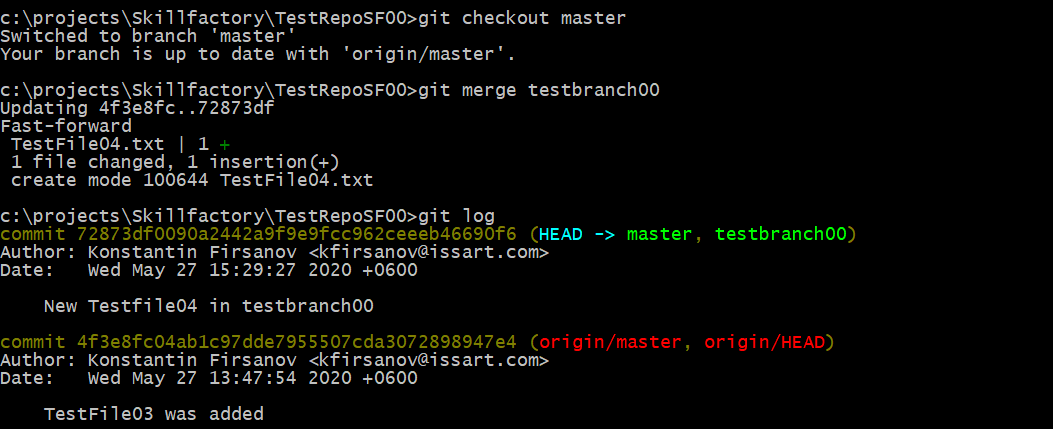
Это также означает, что все изменения, вносимые с этого момента в файлы, будут отнесены к предыдущей версии. Другими словами, откатилась вся работа, выполненная в ветке *testbranch00*, а вы можете продолжать работу с предыдущей версией файлов.

**Зачем нужна команда merge**

Ситуация получается достаточно запутанной:

* + В ветке *testbranch00* есть файлы, которых нет в ветке *master*.
  + В ветке *master* один из файлов отличается от аналогичного в ветке *testbranch00*.

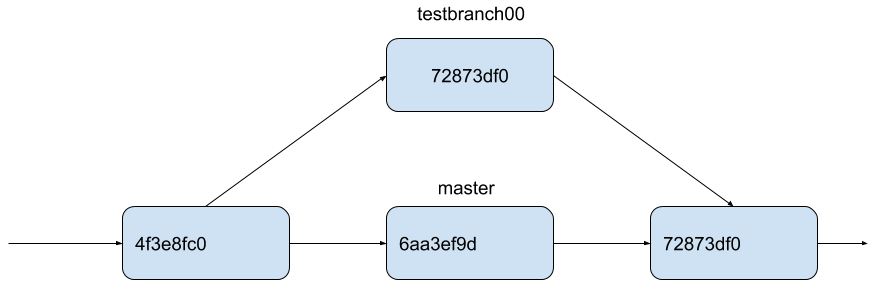
Как же всё привести к общему знаменателю? Очень просто. Переключимся обратно на ветку *master* и добавим изменения из ветки *testbranch00* в ветку *master*. Для добавления изменений используется команда merge. Параметром этой команды передаётся имя ветки, которую нужно присоединить. Выполним эту команду.

Рис. 9.5.5. Изменения в репозитории и списке коммитов после выполнения команды *merge*

**Что изменилось в репозитории и списке коммитов**:

* + Во-первых, в результате выполнения команды git merge был добавлен один файл в ветку *master*.
  + Во-вторых, в список коммитов добавился последний коммит из ветки *testbranch00*.
  + В-третьих, после выполнения команды dir вы увидите, что в ветке master появился файл, который вы добавляли в ветку *testbranch00*.

Графически действия, которые мы сейчас выполнили, могут быть изображены в виде диаграммы, показанной на рисунке 9.5.6.

Рис. 9.5.6. Изменения в репозитории после выполнения команд

Во всех случаях после того как одна ветка объединяется с другой, все коммиты, сделанные в ней, попадают в ветку, с которой она была объединена. В нашем примере объединение ветки *master* с веткой *testbranch00* добавило коммиты последней к первой.

Также важно понимать, что merge это не двунаправленная операция. Файл, который вы меняли в ветке *master*, остается неизменным в ветке *testbranch00*. Если нужно, чтобы это произошло, нужно слить ветку *master* в ветку *testbranch00*. Для этого нужно сначала переключиться на ветку *testbranch00*, а потом выполнить операцию слияния с веткой *master*, выполнив команду git merge master.

Выполним публикацию изменений на ваш удалённый репозиторий.

### **Задание 9.5.5**

1/1 point (graded)

Перейдите в консоль Git, переключитесь на ветку testbranch00. Выполните публикацию изменений этой ветки на сервер. Используйте команду git push --set-upstream origin testbranch00.

Проверьте, появилась ли ветка testbranch00 на удалённом сервере.

В заключение данного юнита хотелось бы рассказать о политике работы с ветками. Прежде всего, важно отметить, что ветка master — это такая же ветка, как и все другие, создаваемые в репозитории. Её отличие в том, что добавляется по умолчанию при создании репозитория.

Часто в процессе разработки, ветка master содержит стабильную версию программного кода или стабильную версию тестов. В других ветках содержатся изменения, например, код нового функционала или код тестов, написанных для нового разработанного функционала.

Обычно перед тем как взяться за решение какой-то задачи, программист заводит новую ветку от последнего рабочего коммита master ветки и решает задачу в этой новой ветке. В ходе решения он делает ряд коммитов, после этого тестирует код непосредственно в ветке задачи. После того как задача решена, делает слияние своей рабочей ветки с веткой master.

Такую схему работы часто используют с юнит-тестами и автоматизированным развёртыванием на сервере. Если юнит-тесты будут покрывать весь код, то можно настроить развёртывание так, что вначале будут выполняться все тесты в ветке задачи. Если тесты прошли успешно, будет происходить слияние и развёртывание. При такой схеме можно добиться полной автоматизации при тестировании и развёртывании.

Описанный процесс работы с ветками — это классический процесс. На практике политика работы с ветками может быть более сложная. Например, может существовать отдельная релизная ветка, в которой тестируется код очередного релиза и исправляются релизные дефекты. В это время в ветках, создаваемых от ветки master, продолжается разработка по задачам для следующего релиза. После завершения релизного тестирования и исправления всех дефектов релизная ветка присоединяется к ветке master.

В следующем уроке мы продолжим работу с ветками, сделав упор на организацию командной работы.

## 9.6. Командная работа

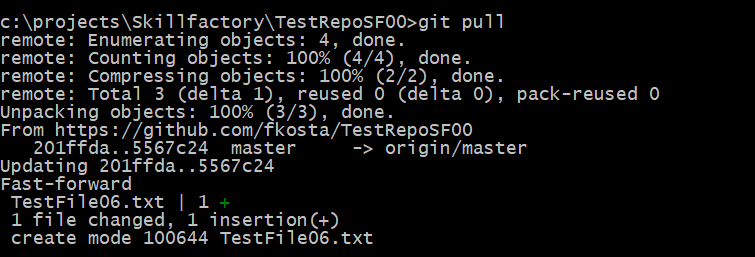
## ****Две концепции командной работы****

Прежде чем мы рассмотрим концепции командной работы, расскажем вам про ещё одну команду, которая получает изменения из репозитория.

Вы уже знаете, как клонировать удалённый репозиторий на локальный компьютер, как отправить изменения в локальном репозитории в удалённый репозиторий на сервере. А как получить изменения с репозитория на сервере? Ведь с этим репозиторием могут работать несколько разработчиков. Они клонируют репозиторий себе на компьютер, выполняют свои задачи, периодически отправляя изменения на сервер. Каждый из них должен каким-то образом получить изменения в коде, которые сделали другие разработчики.

Рассмотрим команду git pull, с помощью которой можно получить изменения, а потом рабочий процесс. Откройте свой репозиторий на GitHub, выберите ветку master и создайте новый файл в репозитории.

После того как файл появится в репозитории, перейдите в локальный репозиторий и выполните команду git pull. Сравните полученное вами с рисунком 9.6.1.

Рис. 9.6.1. Результат выполнения команды git pull

## ****Распределённые рабочие процессы****

Если на проекте работает команда разработчиков, то для работы с удалённым репозиторием на сервере требуется установить определённый рабочий процесс. Существуют различные распределённые рабочие процессы, которые предлагает Git, и мы рассмотрим два из них в теории и один на практике.

## ****Модель централизованного рабочего процесса****

Эта модель унаследована из централизованных СКВ. Существующий центральный общий репозиторий содержит код проекта. Каждый разработчик периодически синхронизируется с ним, публикуя свои изменения. Упрощённая диаграмма работы по этому процессу приведена ниже.

Рис. 9.6.2. Модель централизованного рабочего процесса

При работе по данному процессу есть существенная особенность. Если два разработчика копируют содержимое центрального репозитория и оба делают изменения, то первый разработчик, который отправит свои изменения обратно в центральный репозиторий, может сделать это без проблем. А вот второй разработчик должен добавить изменения, сделанные первым, прежде чем отправить свои изменения в центральный репозиторий, чтобы случайно не перезаписать изменения первого разработчика. Git поддерживает такой рабочий процесс лучше, чем SVN, потому что Git не позволит, чтобы пользователи перезаписали работу друг друга.

Всё, что нужно для работы по этому процессу — это настроить единый репозиторий, предоставляющий всем членам команды быстрый доступ. Например, разработчик 1 и разработчик 2 оба начинают работать в одно и то же время. Разработчик 1 заканчивает свои изменения и отправляет их на сервер, используя команду git push. Если разработчик 2 попытается отправить свои изменения, то сервер отклонит их, и команда git push, которую он использует для отправки изменений на сервер, вернёт ему предупреждение, что это сделать невозможно, пока изменения не получены с сервера по команде git pull.

Этот рабочий процесс прост и интуитивно понятен, и он используется не только в маленьких командах. С гибкой системой ветвления, которой обладает Git, этим процессом также успешно пользуются и достаточно крупные команды.

## ****Рабочий процесс «менеджер по интеграции»****

Поскольку Git позволяет работать с несколькими удалёнными репозиториями, возможно установить рабочий процесс, в котором каждый разработчик:

* + Может предоставить другим доступ «по записи» к своему публичному репозиторию.
  + Получить доступ «по чтению» ко всем остальным.

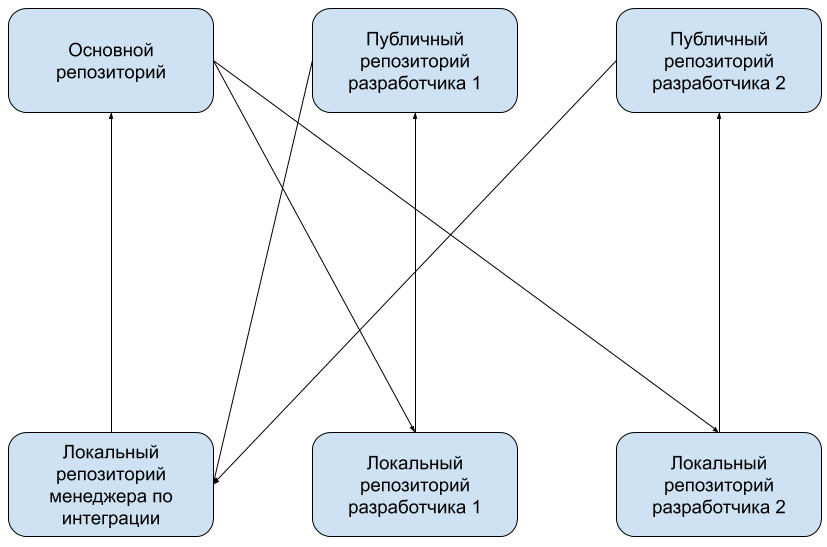
Этот сценарий часто включает в себя утверждённый (главный) репозиторий, представляющий «официальный» проект. Чтобы добавить свои изменения в этот проект, каждый из разработчиков создаёт свой собственный публичный клон проекта (свой собственный публичный репозиторий) и публикует свои изменения только туда.

Основной репозиторий доступен «по записи» только менеджеру по интеграции. Также менеджеру по интеграции доступны репозитории разработчиков только для чтения.

Шаги процесса по публикации изменений в основной репозиторий следующие:

* 1. Разработчики клонируют основной репозиторий, и каждый создаёт локальную и публичную копии.
  2. Разработчики отправляют изменения в свои публичные репозитории.
  3. Разработчики отправляют запрос менеджеру по интеграции, имеющему доступ по записи в основной репозиторий, чтобы тот загрузил их изменения в свой локальный репозиторий.
  4. Менеджер по интеграции загружает изменения, объединяет их с кодом в своём локальном репозитории.
  5. Менеджер по интеграции публикует изменения в основной репозиторий.

Диаграмма взаимодействия по этому процессу показана на рисунке.

Рис. 9.6.3. Диаграмма взаимодействия в рабочем процессе «менеджер по интеграции»

Такой процесс очень часто можно встретить, когда используется GitHub. В GitHub один пользователь легко может создать копию репозитория другого пользователя. Созданная копия будет доступна владельцу оригинального репозитория, из которой он сможет загружать изменения в свой оригинальный репозиторий.

**Важно!** Найдите напарника для выполнения заданий.

Для выполнения серии заданий ниже необходимо создать группы из двух студентов. Задания, помеченные **Студент 1**, выполняет первый, а **Студент 2**, соответственно, второй. Договоритесь в Slack с однокурсником, с которым сможете выполнить все следующие задания этого урока.

### **Задание 9.6.2**

**Студент 1**: Дайте ссылку на свой репозиторий GitHub коллеге. Пусть тот авторизуется в GitHub и перейдёт по этой ссылке.

**Студент 2**: Перейдите по предоставленной вам ссылке. В открывшемся окне предоставленного репозитория кликните на кнопку fork. Под вашим аккаунтом в списке репозиториев появится клонированный репозиторий вашего коллеги, который предоставил вам ссылку.

Согласно диаграмме процессов, пользователи скопированных репозиториев добавляют свои изменения только в них.

### **Задание 9.6.3**

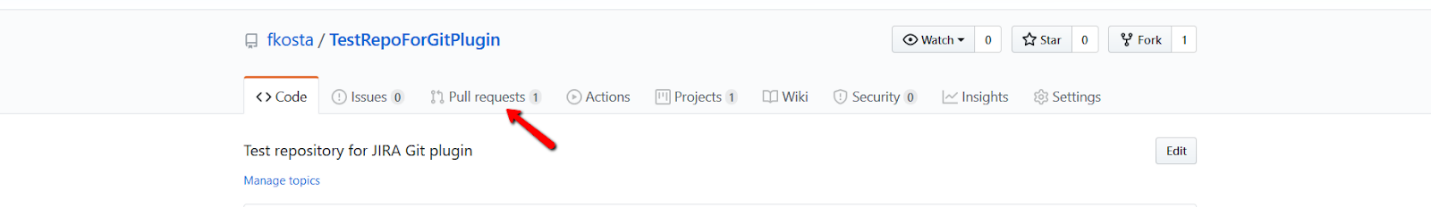
**Студент 2**: После клонирования репозитория перейдите в список репозиториев, найдите клонированный репозиторий и откройте его. Создайте файл в этом репозитории. Кликните на кнопку Create file и откроется форма для создания файла. Введите имя файла, его содержимое и ниже в форме описание коммита.

Каждый новый файл, который вы создаёте в GitHub или загружаете туда, проходит процедуру коммита, прежде чем вы увидите этот файл в списке файлов репозитория. Кликните на кнопку Commit new file.

Появился новый файл в вашей копии репозитория?

**Студент 2**: Нажмите на кнопку New pull request и в открывшейся форме кликните на кнопку Create pull request. Откроется форма для дополнительного описания. Можете пропустить её заполнение и нажать на кнопку Create pull request. Запрос создан.

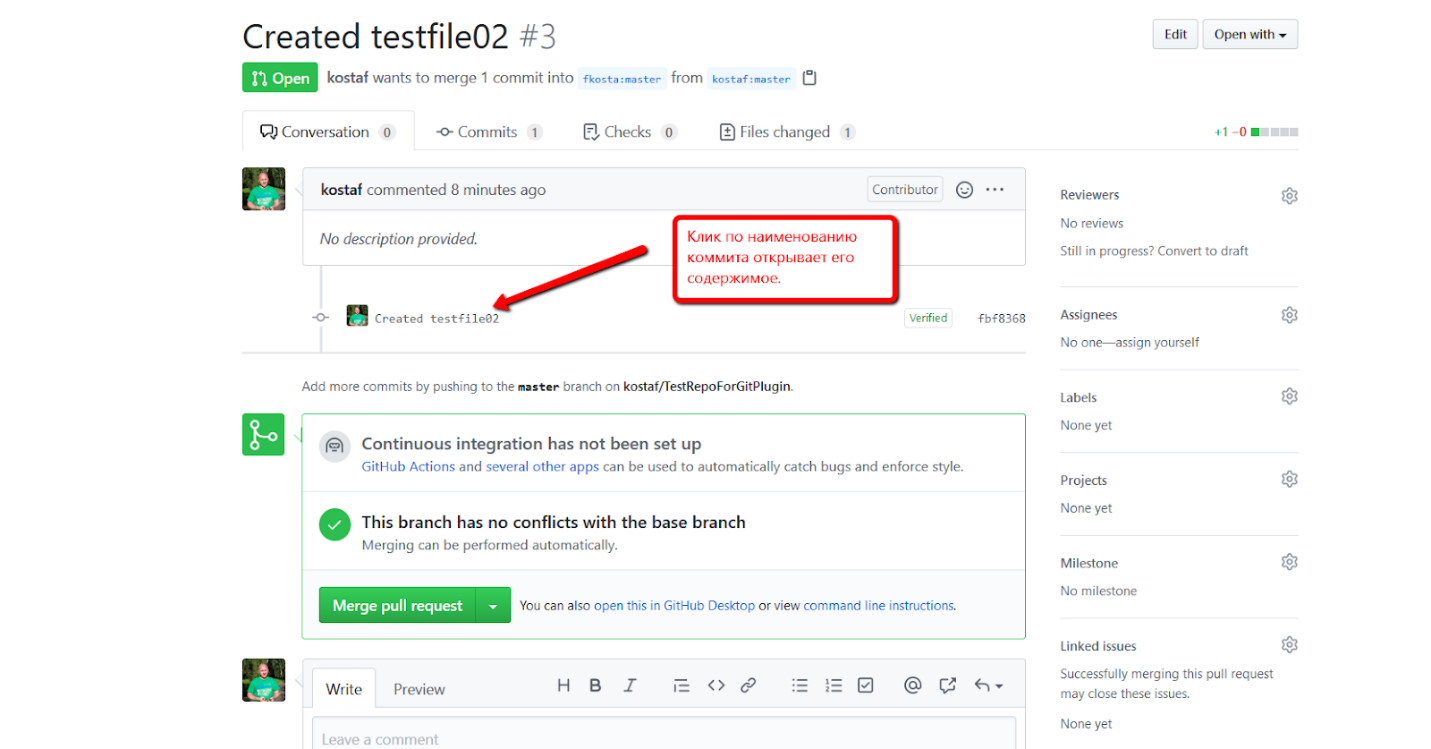
**Студент 1**: Откройте репозиторий, ссылку на который вы предоставили, и проверьте, появился ли у него запрос на загрузку изменений.



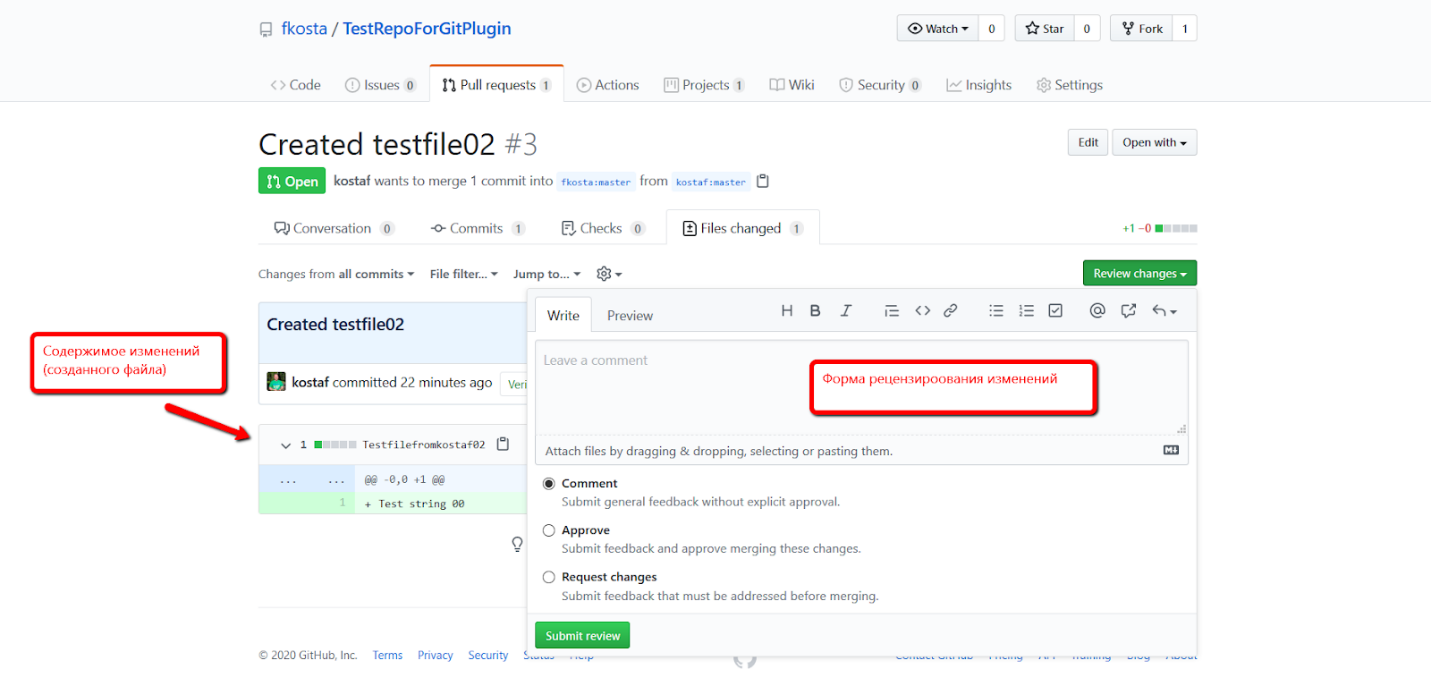
После этого владелец репозитория может начать обработку запроса. Клик на закладку *Pull request* откроет форму со списком запросов, из которой нужно будет выбрать нужный запрос. После выбора нужного запроса и клика по нему откроется форма обработки этого запроса.

Из этой формы можно:

* + Перейти на просмотр изменений, связанных с коммитом, их верификацию, одобрение или отклонение.
  + Подтвердить загрузку и добавление сделанных изменений.

Рис. 9.6.4. Работа с коммитами и списками запросов

По клику на изменения откроется форма со списком изменений. Клик на кнопку *Review changes* добавляет форму рецензирования, в которой можно оставить свой комментарий, одобрить или отклонить изменения.

Рис. 9.6.5. Форма рецензирования

Если изменения одобрены, то выбирается *approve* и производится клик по кнопке *submit review*. После этого форма просмотра изменений закрывается, и владелец репозитория может кликнуть на кнопку *merge pull request*. И новый файл, добавленный его коллегой в клон репозитория владельца, появится в репозитории у владельца.