Инструментальные средства информационных систем

Автор: Сафиуллина Н.Ф.

Лекция 1 Инструменты контроля версий

Содержание

Введение

В этой лекции мы узнаем о системах контроля версий и одной из самых популярных — *GIT*, будем создавать репозитории на своём компьютере и на онлайн-портале *GitHub*, узнаем о ветках, коммитах и прочем.

Git — это система управления версиями с открытым исходным кодом. Она упрощает совместную работу над проектами с помощью *распределенной системы управления версиями* файлов (бывают ещё локальные и централизованные), которые хранятся в *репозиториях*.

GitHub — это служба размещения в Интернете репозиториев Git. Некая платформа для размещения кода для контроля версий и совместной работы над одним проектов из любого места. С помощью GitHub можно создавать, распространять и поддерживать программное обеспечение.

С 1-го октября 2020 GitHub поменял название стандартной ветки с master на main. При создании нового репозитория главная ветка будет иметь название main.

Из-за этого возможны появление некоторых проблем — во всей документации и материалах главная ветка репозитория названа master; различное ПО считает ветку master главной и строит логику на этом.

Название главной ветки можно указать в настройках аккаунта. Мы этого делать не будем, просто имейте это в виду.

1 Системы контроля версий

Представьте работу команды разработчиков, проект программного продукта имеет множество функциональностей. С каждой итерацией разработки появляются новые функциональности, меняются старые. Над проектом одновременно работают множество разработчиков, тестировщиков и других. Вся система постоянно меняется. Накапливается множество версий модулей проекта и сопроводительного материала.

Помочь в управлении этим всем призваны системы контроля версий.

Основная задача систем контроля версий – это вести версионность продукта, т.е. вести историю изменений продукта. Можно отслеживать кто, когда, куда вносил изменения и, главное, какие изменения вносились. Основное преимущество версионности, это возможность откатить продукт на одну или несколько версий назад, если это потребуется.

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.

Применять системы контроля версий можно не только к программному коду, но и к любым типам файлов. Например, при совместном авторстве книги.

1.1 Виды систем контроля версий

Системы контроля версий обладают многими другими полезными свойствами, помимо версионности, необходимыми именно для разработки программного кода. Мы рассмотрим важность системы контроля версий на примере одной из них — Git.

Прежде чем мы перейдем к Git, рассмотрим преимущества и возможные недостатки некоторых видов систем контроля версий:

– локальной,

– централизованной,

– децентрализованной.

1.1.1 Локальные системы контроля версий

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

Среди **достоинств** таких систем нужно отметить:

– Возможность восстановления данных из определенной версии (точно определяется по времени записи).

– Высокая скорость выполнения восстановления (база данных четко 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)*. Опишем принципы ее 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 извлеченной 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.

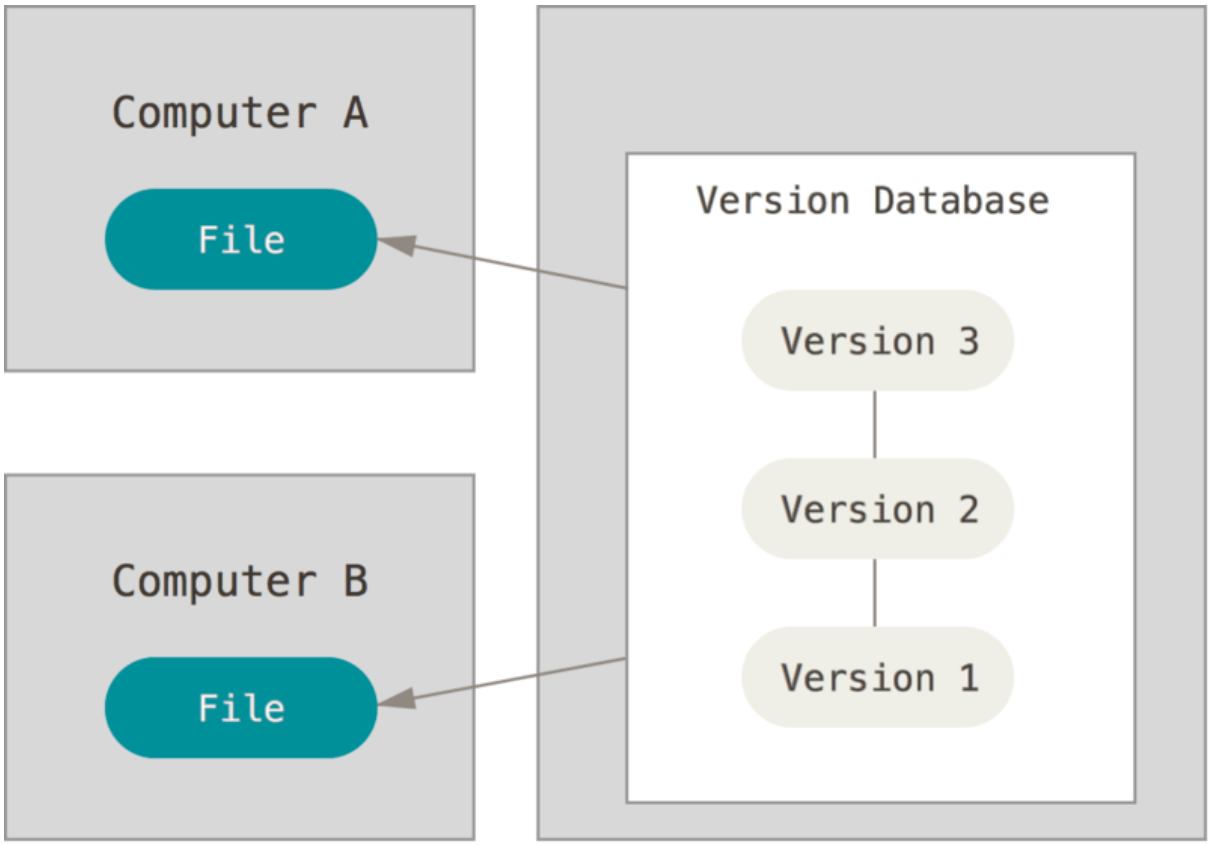
— Версии каждого отслеживаемого файла хранятся в соответствующем файле истории.

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.

1.1.2 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 распределенной 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 поврежден, а резервных копий не будет, то вся история проекта (за исключением единичных снимков репозитория, которые сохранились на локальных машинах разработчиков) будет потеряна.

1.1.3 Децентрализованные системы контроля версий

Представитель децентрализованной СКВ — *Git* (от англ. — *Global Information Tracker*) — это распределённая система управления версиями.

Дополним, *GIT* является распределенной, 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 развертывания.

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 измененной версии существующего файла.

1.2 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.

1.3 Основные состояния Git

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

1. Зафиксированное.
2. Измененное.
3. Подготовленное.

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

Для того чтобы зафиксировать файл, необходимо перевести его в подготовленное состояние. Это происходит при выполнении определенной операции, о которой речь пойдет далее. После того как файлы подготовлены, можно провести фиксацию изменений, что также выполняется специальной командой.

Диаграмма перехода от состояния к состоянию показана на рисунке ниже.

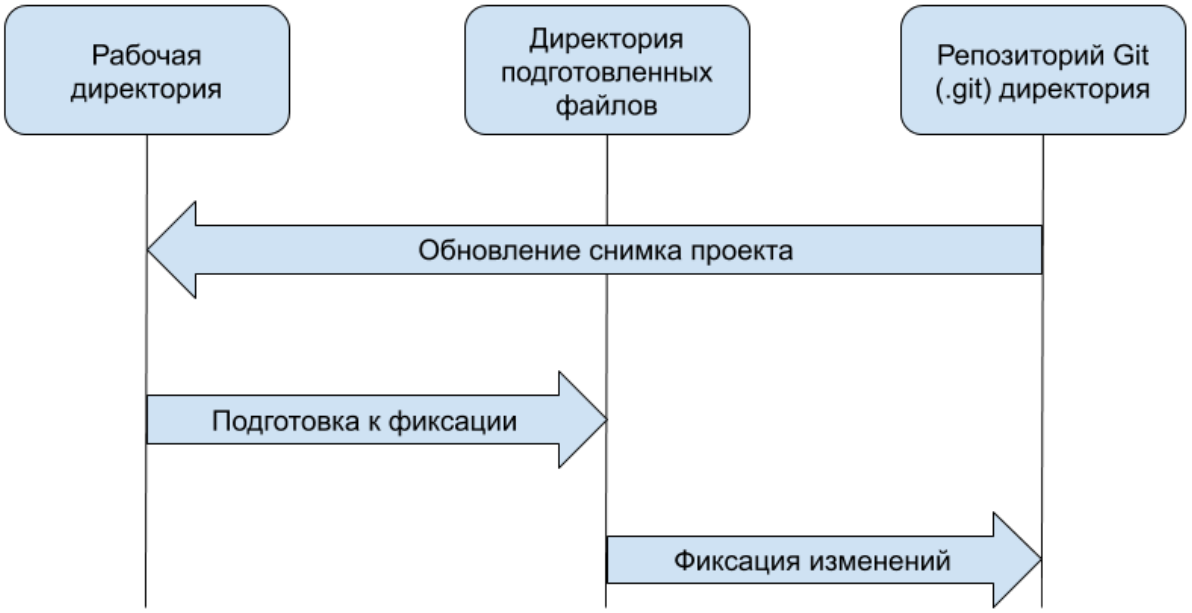


Рис. Диаграмма перехода от состояния к состоянию в *Git*

Технически, когда вы создаете *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 еще 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.

2 Сервис GitHub

2.1 Облачные решения для Git репозиториев

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

Есть множество облачных решений для развертывания *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 сервисом.

2.2 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.

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.

2.3 Как создать аккаунт на GitHub

Прежде чем перейти к знакомству с *GitHub*, необходимо создать аккаунт на этом сервисе. Выполните следующие шаги:

1. Есть ли у вас учетная запись на этом сервисе?

— Есть — пропустите этот шаг.

— Нет — перейдите по ссылке к форме регистрации.

2. Выберите имя пользователя, почтовый ящик, пароль и кликните на *Sign Up*. После проверки ваших учётных данных *Git* предложит решить простую задачу, чтобы удостовериться, что вы человек, а не робот.
3. Кликните на кнопку *Join free plan,* на открывшейся странице выберите ваши активности и то, как вы планируете использовать *GIT*.
4. Кликните Complete setup, и на следующей странице GitHub предложит вам верифицировать ваш email адрес.

5.

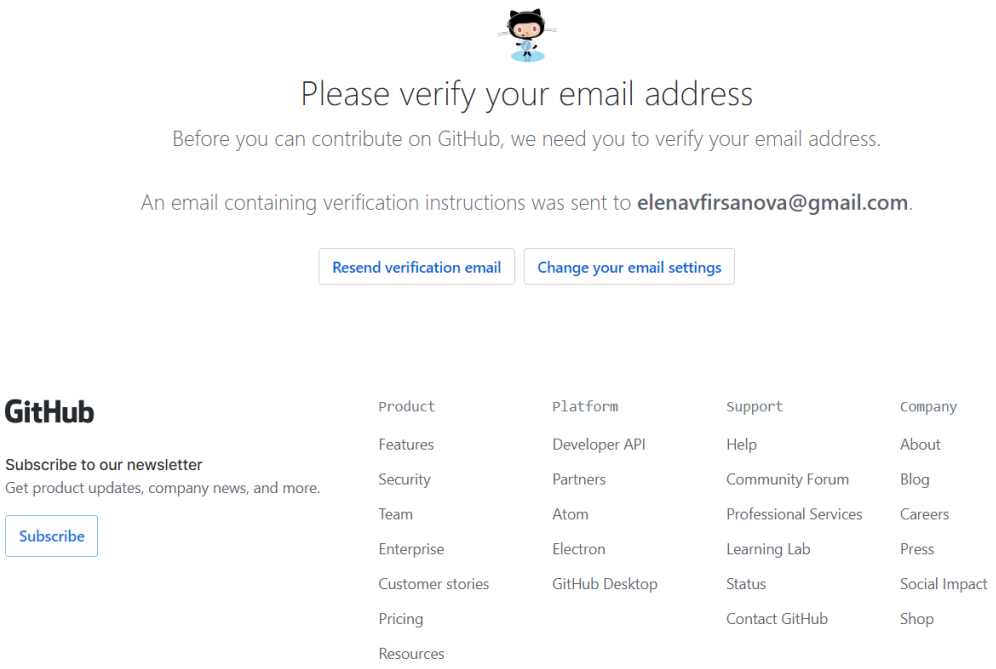


Рис. Верификация *email*

6. Откройте папку входящие своего email и найдите сообщение от GitHub. Откройте его и кликните в этом сообщении на кнопку Verify email address.

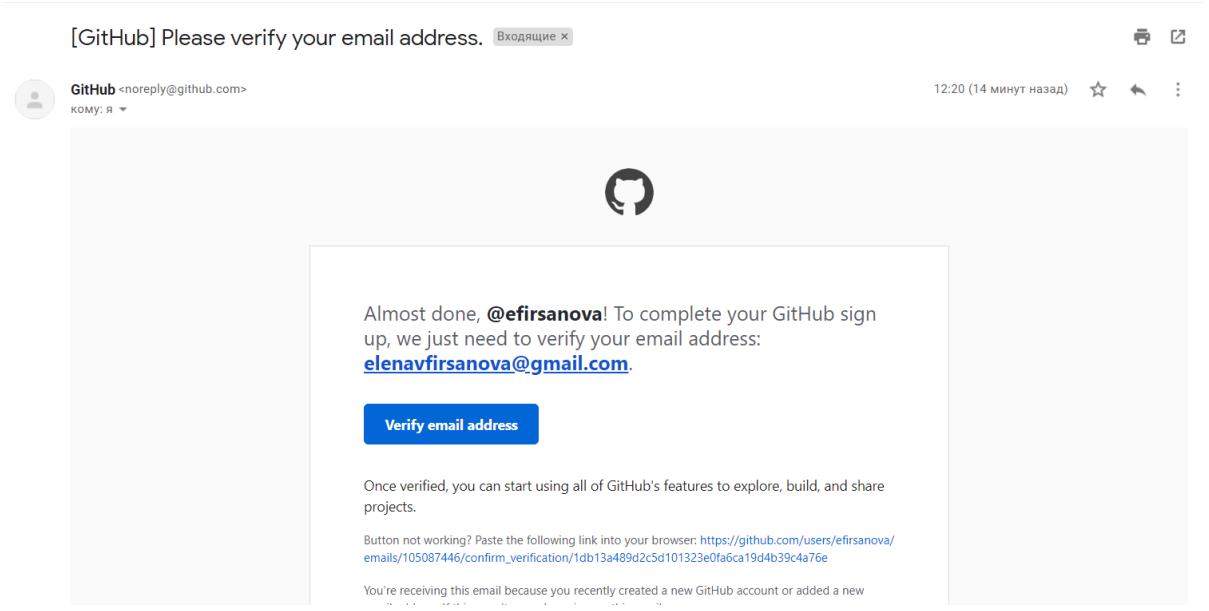


Рис. Сообщение от *GitHub* в электронной почте

7. После этого вы будете перенаправлены на свою страничку *GitHub*, где увидите сообщение об успешном подтверждении вашего *email* и предложение о первых шагах на *GitHub*.

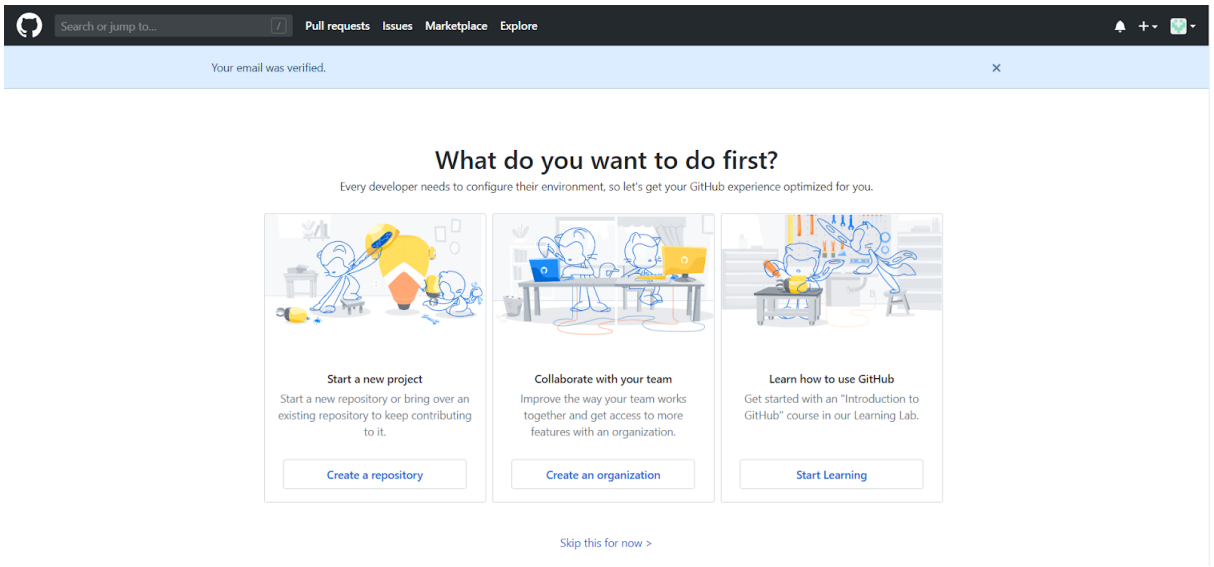


Рис. Первые шаги на *GitHub*. Что сделать в первую очередь?

Начните со своей домашней страницы *GitHub*. Для перехода на неё кликните по ссылке *Skip this for now*. Это страничка, на которую вы будете попадать каждый раз после авторизации в *GitHub*. С помощью команды *Read the guide* вы узнаете, как создать новый репозиторий, какие виды репозиториев могут быть созданы пользователями с бесплатными аккаунтами, какие еще функции есть у этого сервиса.

3 Основы работы с GIT

3.1 Устанавливаем приложение git-scm

В предыдущем разделе мы познакомились с сервисом *GitHub*. Но созданного репозитория на *GitHub* недостаточно для полноценного функционирования системы контроля версий. Необходимо установить приложение, которое бы управляло репозиторием локально и выполняло синхронизацию изменений с *GitHub*.

Есть несколько таких приложений. В этом курсе мы будем работать с одним из них — git-scm. Перейдите по ссылке, откройте раздел *Downloads* (загрузки) выберите операционную систему вашего компьютера и кликните соответствующую ссылку для загрузки приложения.

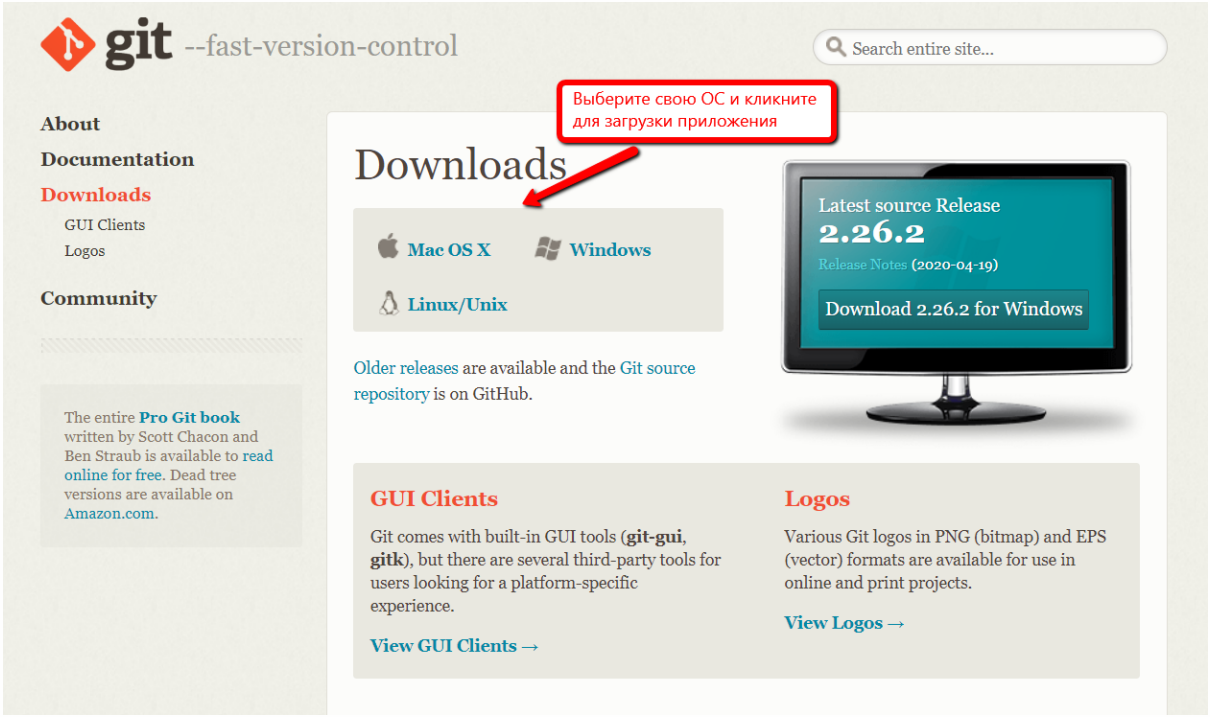


Рис. Загрузка приложения git-scm

После загрузки приложения установите его. Для пользователя Windows установщик последовательно запросит:

— Разрешение изменений на компьютере, согласие с положениями лицензии.

— Каталог для установки приложения, параметры установки, определяемые чекбоксами, название папки приложения в стартовом меню, редактор, который будет использован для редактирования комментариев, использование компонентов приложения (Git GUI и Git Bash — о них позднее), библиотеки SSL/TLS, опции преобразования окончания строки, специфичные для операционной системы и дополнительные опции.

На все эти запросы можно просто выбрать значения по умолчанию. (Подробнее процесс установки есть в лабораторной работе 1.)

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.

3.2 Клонирование репозитория

Работа с командной строкой даст более глубокое понимание того, как работает *Git*.

Создайте репозиторий *InformationSystemTools* с файлом *Readme* в своем аккаунте *GitHub*. Выполним его клонирование с *GitHub*, для этого:

1. 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*.
2. С помощью команды операционной системы *cd* перейдите к каталогу на своём компьютере, в который вы хотите клонировать репозиторий с *GitHub*. Например, *cd c:\education* (формат команды для *Windows*).
3. 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. Нажатием на клавишу *Enter* завершите ввод. Репозиторий с *GitHub* будет клонирован на компьютер.

Рассмотрим содержимое папки *education* на компьютере, в которую был клонирован репозиторий с *GitHub*. В этой папке находится ещё одна папка с именем клонированного репозитория *InformationSystemTools*. 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.

3.3 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.

3.4 Команды для работы с репозиторием

В предыдущем разделе мы рассмотрели первый и важный шаг в начале работы с репозиторием, который находится на GitHub. Клонирование репозитория, созданного на GitHub, на свои компьютеры. Таким образом, получают два синхронизированных между собой репозитория.

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

`git status` — первая команда, команда получения статуса файлов в репозитории.

Для выполнения команды сделайте следующее:

1. Запустите Git cmd.
2. Перейдите в папку с репозиторием, который вы клонировали.
3. Используйте команду операционной системы cd для перехода в эту папку.
4. Выполните команду git status.
5. Прочитайте сообщения, которые вы увидите на экране:

```
c:\>d:

d:\GIT>cd d:\GIT\Information-Systems-Tools

d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

d:\GIT\Information-Systems-Tools>
```

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 удаленный сервер.

— *Working tree clean* – означает, что рабочий каталог не содержит измененных 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 еще не были зафиксированы.

3.5 Алгоритм работы с файлами

Рассмотрим алгоритм работы с файлами, помещенными 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 удаленном сервере (в нашем случае с репозиторием на *GitHub*, с которого мы клонировали свой локальный репозиторий).

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

Добавим текстовый файл (например, testFile01.txt) 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.

Выполните команду **`git status`**. Убедитесь, что появился информационный блок, где была выведена информация о неотслеживаемых файлах (`Untracked files`). 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.`**

```
d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        testFile01.txt
```

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

d:\GIT\Information-Systems-Tools>
```

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 **`.`** (точка), что означает «добавить все измененные 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.

```
d:\GIT\Information-Systems-Tools>git add .

d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   testFile01.txt


d:\GIT\Information-Systems-Tools>
```

3.6 Команда git commit

Git хранит данные в виде набора легковесных «снимков», известных как *коммиты*. Они хранят состояние файловой системы в определенный момент времени, а также указатель на предыдущие коммиты.

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 ниже 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 *main* локального репозитория опережает ветку *main* на сервере на 1 коммит. И это не удивительно, потому как в удалённом репозитории на *GitHub* пока нет того файла, который мы добавили в репозиторий локально.

```
d:\GIT\Information-Systems-Tools>git commit -m "File for example"
[main 6442220] File for example
 1 file changed, 1 insertion(+)
 create mode 100644 testFile01.txt

d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

d:\GIT\Information-Systems-Tools>
```

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

3.7 Команда git push

Применим команду **`git push.`**

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

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 добавленный 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.

```
d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is up to date with 'origin/main'.

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:   testFile01.txt

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

d:\GIT\Information-Systems-Tools>
```

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

Введите команду **`git status`** и обратите внимание, что измененный 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*).

```
d:\GIT\Information-Systems-Tools>git add .

d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   testFile01.txt


d:\GIT\Information-Systems-Tools>
```

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.

3.8 Жизненный цикл состояний файлов

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

3.9 Команда git log

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

```
d:\GIT\Information-Systems-Tools>git log
commit 6442220a637e70f0011453f1aed0b9a648a9b90f (HEAD -> main)
Author: Natalia Safiullina <nfsafiullina@gmail.com>
Date:   Wed Jun 14 22:12:18 2023 +0700

    File for example

d:\GIT\Information-Systems-Tools>
```

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 -> main

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

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

3.10 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 скриншоте ниже показан процесс удаления файла. После команды **`git rm`** была выполнена команда **`git status`**, которая отметила файл как удаленный в информационном блоке о файлах для коммита. После команды **`git commit`** выполняется команда **`git push`** для синхронизации изменений с репозиторием на *GitHub*. Переход в репозиторий на *GitHub* подтвердит, что файл, который удаляли локально, отсутствует.

```
d:\GIT\Information-Systems-Tools>git rm testFile01.txt
rm 'testFile01.txt'

d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    testFile01.txt


d:\GIT\Information-Systems-Tools>git commit -m "File for example, removed"
[main 607cf24] File for example, removed
 1 file changed, 2 deletions(-)
```

```
 delete mode 100644 testFile01.txt

d:\GIT\Information-Systems-Tools>git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

d:\GIT\Information-Systems-Tools>
```

4 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.

Ветки нужны для добавления новой функциональности, не изменяя основную. Обычно перед тем как взяться за решение какой-то задачи, программист заводит новую ветку от последнего рабочего коммита *main* 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 *main*.

Слияние веток делается командой: **`git merge.`**

Слияние веток это однонаправленный процесс, если мы сливаем ветки, то сливаем одну ветку в другую, а не каждую с каждой.

Важно отметить, что ветка *main* — 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.

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 *main*, продолжается разработка по задачам для следующего релиза. После завершения релизного тестирования и исправления всех дефектов релизная ветка присоединяется к ветке *main*.

5 Командная работа

5.1 Команда git pull

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

Это команда: **`git pull.`**

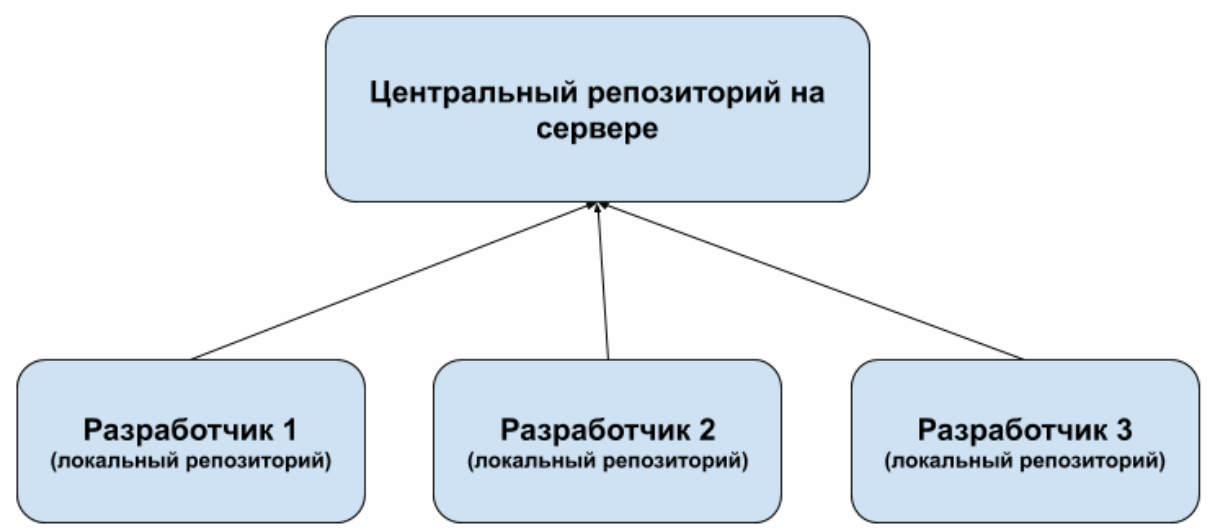
Вы уже знаете, как клонировать удалённый репозиторий на локальный компьютер, как отправить изменения в локальном репозитории в удалённый репозиторий на сервере. А как получить изменения с репозитория на сервере? Ведь с этим репозиторием могут работать несколько разработчиков. Они клонируют репозиторий себе на компьютер, выполняют свои задачи, периодически отправляя изменения на сервер. Команда **`git pull`** нужна чтобы каждый из них мог получить изменения в коде, которые сделали другие разработчики.

5.2 Распределенные рабочие процессы

Если над проектом работает команда разработчиков, то для работы с удалённым репозиторием на сервере требуется установить определенный рабочий процесс. Существуют различные распределенные рабочие процессы, которые предлагает *Git*, и мы рассмотрим два из них.

5.2.1 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.



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.

5.2.2 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.

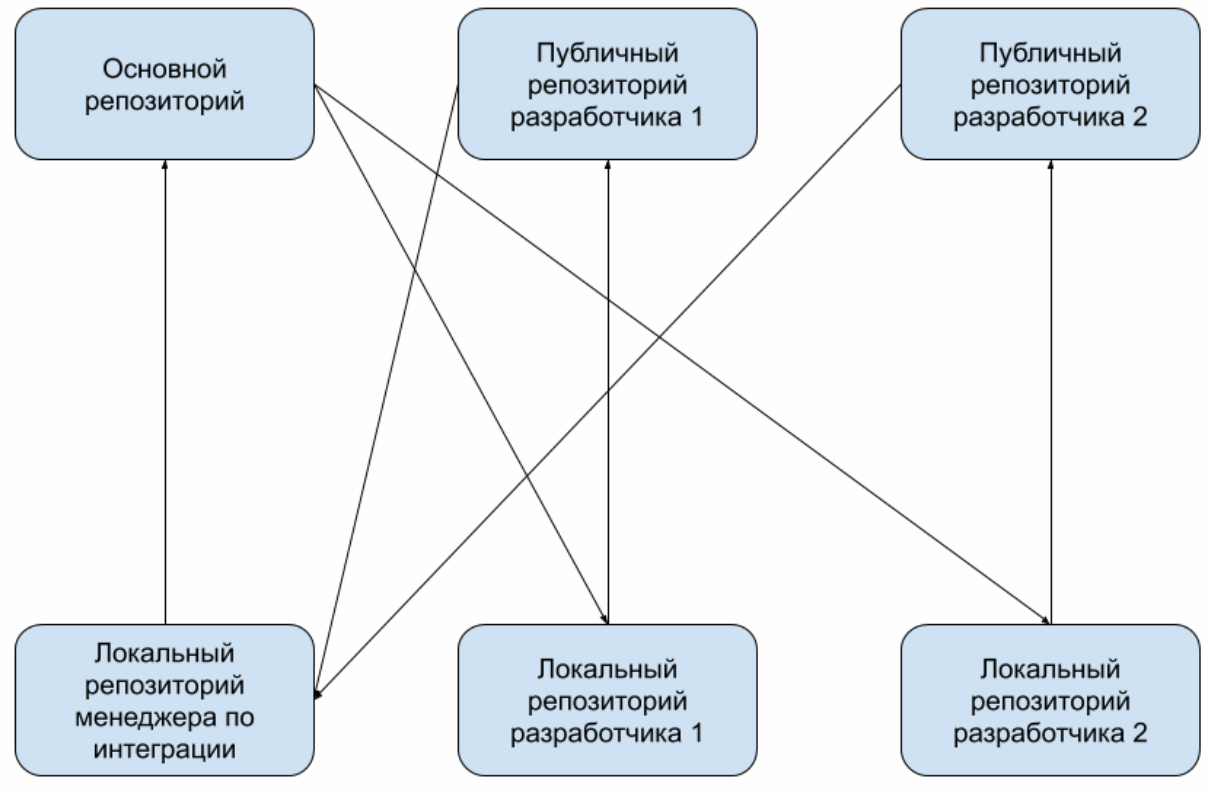
Этот сценарий часто включает в себя утверждённый (главный) репозиторий, представляющий «официальный» проект. Чтобы добавить свои изменения в этот проект, каждый из разработчиков создает свой собственный публичный клон проекта (свой собственный публичный репозиторий) и публикует свои изменения только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



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.