Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Российская академия народного хозяйства и государственной службы при Президенте Российской Федерации»

Нижегородский институт управления

Кафедра Информатики и информационных технологий

ОТЧЕТ

НА ТЕМУ

«GitHub»

Выполнил: студент группы: ИБ-321

Родионов Роман Александрович

Преподаватель: Окулич Виктор Иванович

Нижний Новгород

2023 г.

Оглавление

[**История GitHub** 3](#_Toc127717533)

[**Работа с GitHub** 4](#_Toc127717534)

[**Клонирование репозитория** 5](#_Toc127717535)

[**Отображение изменений файлов** 5](#_Toc127717536)

[**Сохранение изменений и отправка изменений на сервер** 6](#_Toc127717537)

[**Откат изменений** 7](#_Toc127717538)

[**Командная работа** 8](#_Toc127717539)

[**Процесс разработки** 9](#_Toc127717540)

[**Слияние веток** 10](#_Toc127717541)

# **История GitHub**

Идею, заложенную в основе Git, выдвинул создатель ядра ОС Linux Линус Торвальдс. Выбирая название технологии, Торвальдс взял сленговое слово *git*, что можно перевести на русский как «[презренный, неприятный субъект](https://www.urbandictionary.com/define.php?term=Git)». Линус предложил расшифровывать его как аббревиатуру двумя способами:

* Global Information Tracker (англ. «глобальный информационный трекер»), когда всё работает;
* Goddamn Idiotic Truckload of shit (англ. «чёртов идиотский самосвал дерьма»), когда всё плохо.

По своей сути Git — это просто программа, консольная утилита, которая управляется командами через терминал.

Когда вы начинаете работать со своим проектом через Git, папка с проектом становится **репозито́рием**. В широком смысле репозиторий — это хранилище с данными. Если же говорить о нём в ключе системы контроля версий, того же Git, — это не просто хранилище данных, это хранилище версий файлов проекта.

Репозитории бывают локальными и удалёнными.

Локальные репозитории хранятся на вашем компьютере, а удалённые вне вашего компьютера, например в облачном сервисе.

С локальным репозиторием удобно работать, если вы пишете проект самостоятельно, без помощи других разработчиков. Но как правило современная разработка — это групповая разработка, где вы с коллегами трудитесь над одним и тем же проектом совместно, в одно и то же время. Для организации подобной работы обычно используется не локальный, а удалённый репозиторий, доступ к которому разработчики получают через специальные веб-сервисы.

Есть несколько таких сервисов; среди них — GitHub, GitLab и BitBucket. Каждый из этих сервисов имеет свои достоинства и недостатки, но базовые принципы работы с ними одинаковые. Дальше по курсу вы будете использовать GitHub, например отправлять через него домашние работы на проверку.

По своей сути GitHub — социальная сеть для разработчиков. С момента своего возникновения в 2008 году GitHub объединил [десятки миллионов человек](https://octoverse.github.com/), дав им возможность реализовывать свои идеи и свободно обмениваться информацией.

Здесь можно завести аккаунт и размещать свой код, совместно работать над собственными или *open source* проектами (англ. «открытое программное обеспечение»), а на сервисе [GitHub Pages](https://pages.github.com/) можно даже бесплатно разместить свой собственный сайт.

Доступ к репозиториям, которые хранятся на GitHub обеспечивается интерфейсом командной строки Git и Git-команд.

# **Работа с GitHub**

Место, где хранится и обновляется код проекта, чаще всего в виде файлов, называют **репозито́рием**. Как говорилось ранее, репозиторий может быть как локальным, на вашем компьютере, так и удалённым, на сервере GitHub или на других подобных сервисах.

Основной репозиторий — удалённый, в вашем случае тот, что находится на GitHub. Дополнительный репозиторий — локальный. Он нужен для того, чтобы вносить изменения в код и доставлять эти изменения в основной репозиторий.

Мы создали для вас тестовый репозиторий, который называется *backend\_test\_homework*. Ваша задача — скопировать его в свой аккаунт.

Операция копирования чужого репозитория в свой аккаунт называется «форк» (англ. *fork* — «вилка»), этот процесс происходит на сервере GitHub.

После форка вы получаете полную независимую копию чужого репозитория в свой аккаунт и можете работать с ним, как со своим.

# **Клонирование репозитория**

«Клонировать репозиторий» — значит создать на компьютере его копию. При клонировании вы получаете не только рабочие файлы, но и историю их изменения.

Через терминал перейдите в папку со своими проектами, например, *Dev/*. Используйте команду cd.

В интерфейсе GitHub зайдите в репозиторий *backend\_test\_homework* и скопируйте ссылку на него; эта ссылка потребуется для клонирования репозитория на локальный компьютер. В прошлом уроке вы настроили соединение с GitHub по SSH, так что ссылку берите со вкладки SSH.

Напечатайте в терминале команду git clone, после неё поставьте пробел, вставьте скопированный адрес и выполните команду:

git clone git@github.com:ваш-аккаунт-на-гитхабе/backend\_test\_homework.git

Git запросит пароль перед тем, как сравнить публичный и приватный ключи, введите его. После того как вы введёте пароль, репозиторий загрузится с удалённого сервера на ваш компьютер, в директорию, из которой была выполнена команда клонирования.

# **Отображение изменений файлов**

При работе над проектом нужно сохранять его состояния на разных этапах разработки. У программистов есть отличный инструмент для контроля версий своего проекта — это Git. В Git все версии проекта связаны между собой и разработчик может посмотреть полную историю изменений.

Контроль версий (или версионный контроль) в Git предполагает, что любой файл локального репозитория находится в одном из четырёх состояний:

1. **Неотслеживаемый** (англ. *untracked*).
2. **Отслеживаемый,** *staged*, добавленный в *Staging Area* (англ. «плацдарм», «временное хранилище»). Иначе в Git это называют «добавить в индекс».
3. **Изменённый** (англ. *modified*).
4. **Боевой**, на жаргоне разработчиков «закоммиченный» (англ. *committed*, «брошенный в бой»). [В документации](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository) такие файлы называются **неизменёнными** (от англ. *unmodified*). Иными словами, это файлы, в которых не было никаких изменений с момента последнего коммита (с момента, как разработчик «бросил их в бой»).

Git всегда следит за изменениями, которые происходят в локальном репозитории, будь то создание нового файла или внесение изменений в уже существующий файл. Новый файл в репозитории получает статус «Неотслеживаемый», а файл, содержимое которого разработчик изменил, получает статус «Изменённый».

Чтобы указать Git, что изменения должны быть «сохранены», нужно проиндексировать файлы, то есть перевести их в статус «Отслеживаемый». Далее изменения можно зафиксировать: файлы перейдут в состояние «Неизменённый», а Git начнёт отслеживать уже новые изменения.

# **Сохранение изменений и отправка изменений на сервер**

Чтобы зафиксировать изменения в файле, сначала нужно эти изменения внести и отправить их в индек, выполнив команду git add .. Значит теперь вы можете зафиксировать эти изменения, то есть сделать первый коммит.

Каждый коммит сохраняет актуальное состояние файлов, и вы можете сравнивать разные состояния от коммита к коммиту. Пока что коммиты будут сохранены локально, на вашем компьютере, и доступны только вам.

Отправим коммит и сопроводим его комментарием. Для этого есть команда git commit. Ключ -m (от англ. *message*, «послание») даёт возможность добавить к коммиту комментарий, который пишется в кавычках после ключа:

git commit -m "Add first commit"

*# Сделали первый коммит.*

*# Текст комментария: Add first commit.*

В комментариях описывайте, какие изменения были сделаны в коммите, иначе через неделю вы уже не вспомните, что и зачем было написано. Git понимает кириллицу, но комментировать по-английски — хороший тон.

Все сделанные коммиты сохранены локально, на вашем компьютере. Чтобы ваша работа стала доступна, нужно отправить изменения на сервер.

Для отправки локальных изменений на сервер GitHub, выполните команду из любой директории проекта:

git push

Git запомнил, откуда вы клонировали репозиторий, поэтому не нужно указывать, куда именно отправлять изменения. На сленге эту операцию называют «запушить» (от англ. *push*, «протолкнуть»).

# **Откат изменений**

Когда программист работает над кодом, он вносит в него немало изменений и фиксирует их в Git. Может случиться такое, что после определённых изменений в работе программы что-то пойдёт не так, например одна из функций начнёт возвращать неверный результат, хотя до изменений всё было в порядке. В таком случае можно, используя возможности Git, вернуться к той версии проекта, которая работала корректно. Для этого вам нужно:

1. Получить информацию обо всех коммитах и найти тот коммит, после которого в работе кода возникли проблемы.
2. Просмотреть изменения, которые были сделаны в этом коммите.
3. Вернуться к коммиту, при котором в работе кода было всё в порядке.

Для того чтобы получить информацию о коммитах проекта backend\_test\_homework, находясь в любой папке проекта, выполните команду git log (англ. log, «журнал»)

Просмотреть изменения, внесённые в определённом коммите, можно командой git show (англ. show, «показать»). Она продемонстрирует не просто лог, а конкретные изменения в коде. Введите эту команду, а после неё укажите семь первых символов идентификатора коммита, который вас интересует

Git — это машина времени. Git может вернуть ваш код в любое предыдущее состояние, если оно сохранено в коммите. Для этого есть команда git reset (англ. *reset*, «сброс в исходное состояние»).

Чтобы вернуться к определённому коммиту, нужно выполнить команду git reset и через пробел указать первые семь символов идентификатора нужного коммита, точно так же, как с командой git show

# **Командная работа**

Итак, команда определена, тимлид выбран, роли и задачи поделены, график встреч согласован, но что делать дальше? Как и с чего начать писать код?

Дальше в дело вступает тимлид; как правило, это такой же разработчик, как и другие участники команды, но обладает чуть большим опытом. Именно этот опыт и достанется в качестве приза тимлиду вашей команды.

Первое, что должен сделать тимлид — добавить соразработчиков в свой репозиторий.

В своём репозитории на GitHub тимлид должен перейти во вкладку Settings, затем в левом боковом меню выбрать Collaborators и на открывшейся странице нажать кнопку Add people.

В открывшемся окне необходимо указать данные разработчика, которому предоставляется доступ к репозиторию в качестве соразработчика: его никнейм, полное имя или адрес электронной почты пользователя GitHub.

Эту операцию необходимо повторить для каждого участника команды.

Соразработчики должны принять приглашение, перейдя по ссылке в письме. Ссылка будет активна 7 дней; если за это время приглашенный не воспользуется ссылкой-приглашением, то эту процедуру придётся повторить. Когда все примут приглашение — репозиторий тимлида будет готов к совместной разработке.

# **Процесс разработки**

Ветка — это изолированный поток разработки, в котором можно делать коммиты так, что они не повлияют на код в других ветках проекта.

Основная ветка репозитория называется **master** или [main](https://github.com/github/renaming#new-repositories-use-main-as-the-default-branch-name). Эта ветка создаётся автоматически, когда в проекте инициализируется Git и создаётся первый коммит. В репозитории тимлида сейчас должна быть только эта ветка.

Разработчики проекта могут создавать новые ветки и переключаться между существующими. Обычно в ветке **master** хранят «продуктовую» версию кода — отлаженную и работающую; разработку же ведут в другой, отдельной ветке, например в **develop.** Это хорошая практика, будем придерживаться именно её.

Создать новую ветку можно прямо в репозитории на GitHub, но чаще всего разработчики делают это локально, в склонированном репозитории; в вашей команде создать ветку **develop** должен тимлид.

Можно посмотреть на все ветки, которые есть в склонированном репозитории и узнать, в какой ветке вы находитесь в данный момент. Если вы предпочитаете работу через терминал, то воспользуйтесь командой git branch

Ветка создана, но разработчик останется в ветке master. Чтобы переключиться в ветку develop, нужно ввести команду git checkout develop

Но ветка **develop** создана и сохранена лишь в локальном репозитории тимлида, а в репозитории на GitHub её пока нет. Чтобы сделать её доступной для других разработчиков, тимлид должен выполнить команду:

git push --set-upstream origin develop

После этого новая ветка будет доступна на GitHub, а когда соразработчики склонируют репозиторий тимлида себе, то обе ветки станут локально видны и им.

Если остальные участники команды уже склонировали репозиторий, в котором тимлид еще не успел создать новую ветку, то чтобы увидеть добавленную ветку в удалённом репозитории им необходимо будет выполнить команду git fetch.

Обе ветки сейчас абсолютно идентичны, но скоро всё изменится. Настало время переходить к задачам.

# **Слияние веток**

После решения задачи вашу изолированную ветку нужно объединить с веткой **develop**, залить в неё результаты вашей работы. А когда вся работа над проектом будет завершена, рабочую ветку **develop** тимлид «вольёт» в главную и отправит на проверку ревьюеру. Этот процесс называется «слияние веток», или **merge** («мёрдж», «мёрж», «мердж», «мерж» — как только ни произносят это слово в русскоязычных командах).

Чтобы объединить («смержить») ветки, нужно переключиться в ветку, **куда** должны попасть изменения; из этой ветки нужно выполнить команду на слияние.

Чтобы залить код из ветки **feature/email-validation** в ветку **develop**, нужно

* переключиться на ветку, **в которую** будут залиты изменения: git checkout develop *# Переключились в develop.*
* переместить все коммиты из **feature/email-validation** в ветку **develop** — смержить ветки: git merge feature/email-validation

Если всё прошло хорошо — Git сообщит, сколько строк кода изменилось и в каких файлах.