



# Тема 7:

## Основы систем контроля версий (Git).

# Цель занятия:

Ознакомиться с системами  
контроля версий.





# План занятия:

1. Контроль версий. Git
2. Работа в локальном репозитории
3. .gitignore
4. Github
5. Markdown



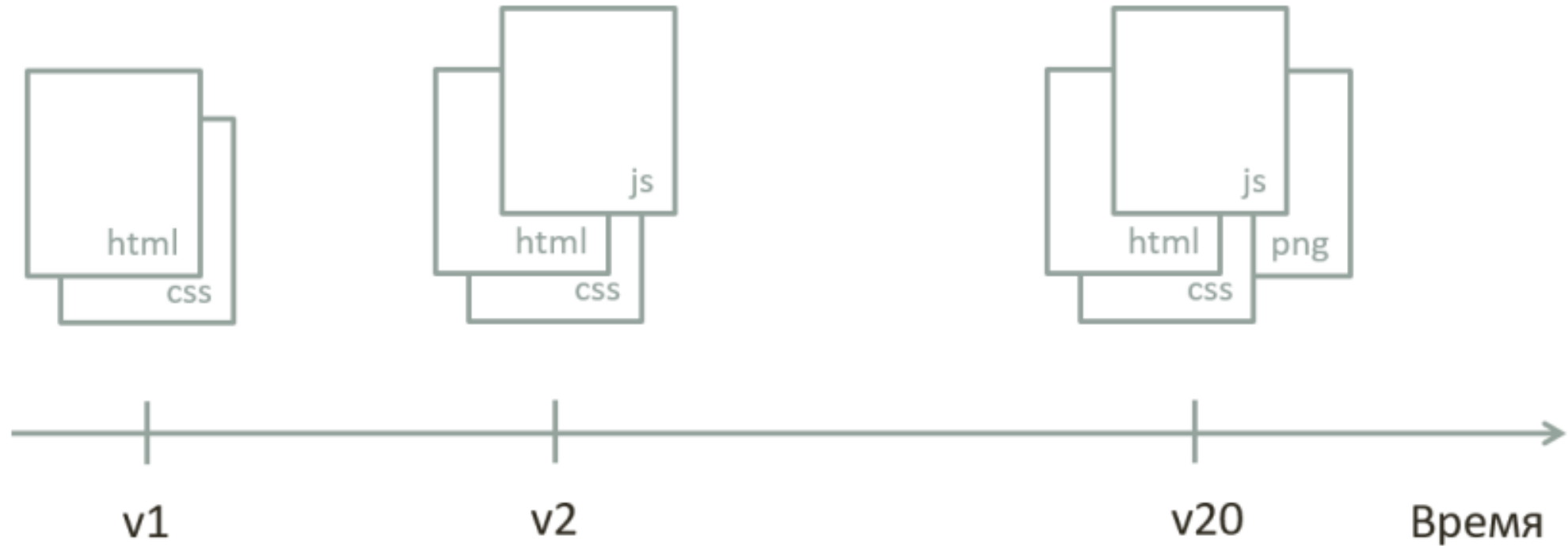
# 1. Контроль версий. Git

Система контроля версий (СКВ) - это программное обеспечение, которое помогает разработчикам отслеживать изменения в файловой системе и управлять версиями файлов и кода.

Она записывает все изменения, которые сделаны над файлами, позволяя вам вернуться к предыдущим версиям, сравнивать изменения, объединять различные версии и отслеживать историю изменений.

Основная цель СКВ заключается в предоставлении средства координации работы между разработчиками, позволяющего им эффективно сотрудничать над проектами.

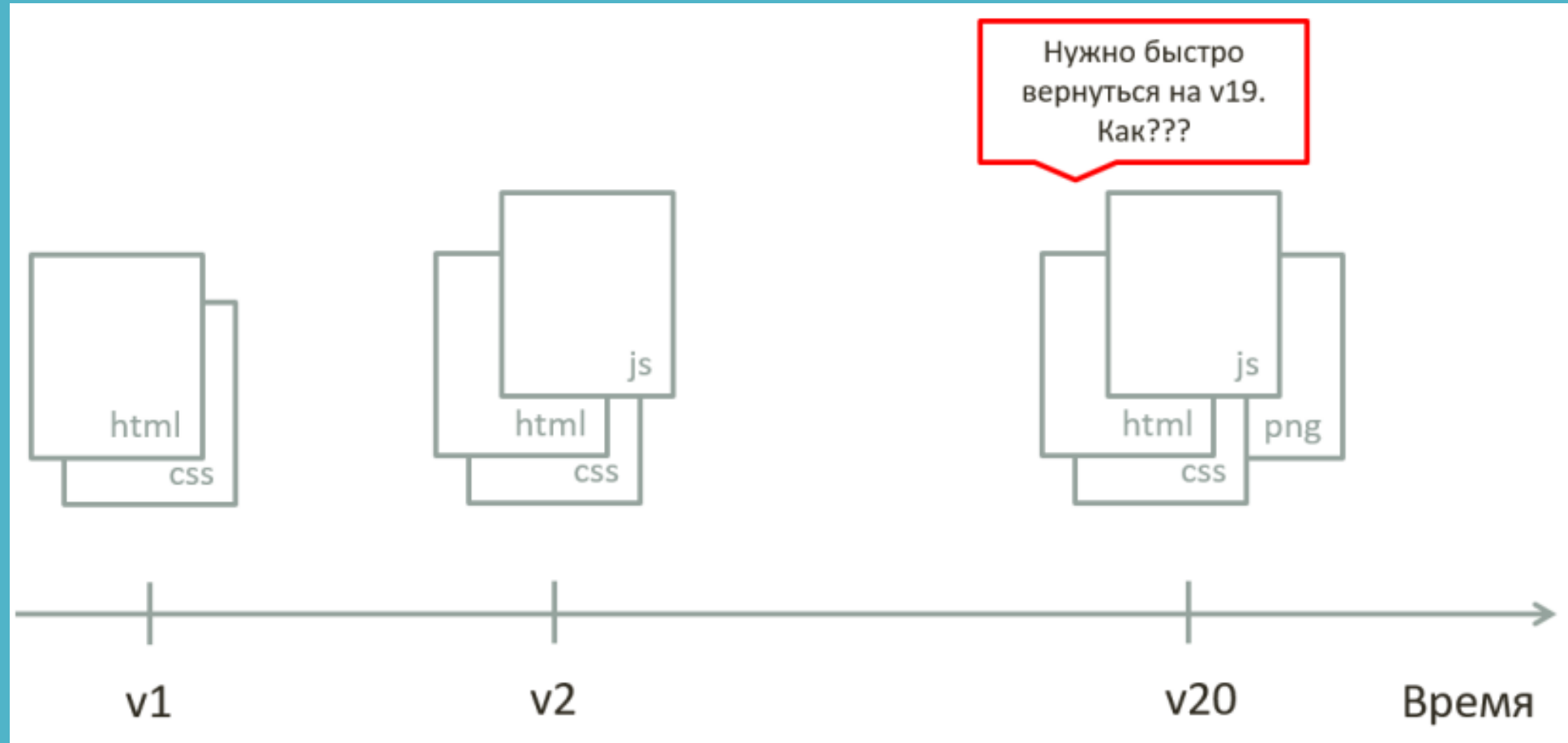
# 1. Контроль версий.



\* HTML / CSS / JS — это файлы для создания сайтов

Представим следующую ситуацию: вы создаёте посадочную страницу. Всё хорошо, и со временем вы туда добавляете новые блоки, специальные акции (распродажи), формы обратной связи и т.д.

# 1. Контроль версий.



И вдруг выясняется, что ваша версия содержит критическую ошибку, и нужно срочно вернуть всё так, как было в предыдущей версии. Что делать?

# 1. Контроль версий.

Можно, конечно, как студенты, хранить много версий файлов (или даже папок), например:

- /сайт\_v1
- /сайт\_v2
- ...
- /сайт\_v20

# 1. Контроль версий.

Можно, конечно, как студенты, хранить много версий файлов (или даже папок), например:

- /сайт\_v1
- /сайт\_v2
- ...
- /сайт\_v20



# 1. Контроль версий.

ПРОБЛЕМЫ ТАКОГО ПОДХОДА:

Случайно удалили каталог;

Случайно удалили файл в каталоге;

Случайно изменили файл в каталоге с предыдущей версией;

Параллельно с коллегой меняете один и тот же файл.

Поэтому нужно использовать специальные системы, которые защищают нас от этих проблем и позволяют сделать процесс контроля версий более удобным.

# 1. Контроль версий.

Системы контроля версий (VCS, от Version Control System) — системы, позволяющие удобно создавать новые версии файлов, отслеживать историю изменений и при необходимости возвращаться к предыдущим версиям, определять авторство изменений, смотреть различия и многое другое.

## Функции системы контроля версий:

- 1.История изменений: СКВ сохраняет полную историю изменений, позволяя вам видеть, какие изменения были сделаны, кто и когда их внес, и почему такие изменения были важны. Это обеспечивает прозрачность и отслеживаемость процесса разработки.
- 2.Восстановление и откат изменений: СКВ позволяет вам возвращаться к предыдущим версиям файлов или кода, в случае необходимости. Это полезно, если внесенные изменения вызвали проблемы или если вам нужно откатиться к стабильной версии.
- 3.Работа в команде: СКВ обеспечивает возможность совместной работы нескольких разработчиков над одним проектом. Она позволяет каждому разработчику работать над своей версией файлов и эффективно объединять изменения, минимизируя конфликты и обеспечивая целостность проекта.

4.Отслеживание ошибок: СКВ часто интегрируется с системами отслеживания ошибок (например, системами управления проблемами или задачами), позволяя связывать изменения с конкретными ошибками или задачами. Это упрощает отслеживание и управление исправлениями ошибок и новыми функциями.

5.Ветвление и слияние: СКВ позволяет создавать ветви (branches), что позволяет разрабатывать новые функции или исправления ошибок независимо от основной версии проекта. Позднее ветви могут быть слиты (слияние) в основную ветвь, позволяя объединить изменения.

6. Анализ истории и авторства изменений;

7. Резервная копия проекта.

## Классификация СКВ:

1. Локальные
2. Централизованные
3. Распределенные



# GIT.

## Требования:

- Опыт работы с Back-end (PHP7+, Mysql, Memcache, Redis, Sphinx, RESTFull, Git\Gitlab, Clickhouse)
- Опыт работы с Front-end (HTML5, CSS3, JavaScript, VUE, WebPack, etc.)
- Наличие опыта в разработке интернет-проектов на фреймворках (Laravel)
- Навыки работы в команде
- Понимание принципов SEO
- Умение разбираться в чужом коде
- Умение настраивать web-сервер на Linux будет плюсом (Nginx+PHP+MySQL+Memcache+Redis)
- приветствуется знание Go
- Ответственность

## Условия:

- График 5-ти дневная рабочая неделя, режим работы: с 9:00 до 18:00.

## Ключевые навыки

PHP

MySQL

HTML5

JavaScript

Git

Laravel

VueJS

CSS3

# GIT. Введение.

**Git** — это распределённая система контроля версий, разработанная Линусом Торвальдсом, создателем ядра Linux.

В 2005 году Линус Торвальдс столкнулся с проблемами в другой системе контроля версий, которую он использовал для разработки ядра Linux. Он решил создать свою собственную систему контроля версий.

В апреле 2005 года Линус Торвальдс выпустил первую версию Git. Он разместил исходный код системы на сервере и пригласил разработчиков ядра Linux присоединиться к проекту.

Git быстро набрал популярность среди разработчиков благодаря своей скорости, эффективности и гибкости. В 2008 году компания GitHub была создана на основе Git, предоставляя облачное хранилище и социальную платформу для совместной разработки.

Git стал стандартом в индустрии разработки программного обеспечения. Многие крупные проекты и компании, такие как Google, Microsoft, Facebook и многие другие, используют Git для управления своим кодом.

В 2018 году Microsoft приобрела GitHub за 7,5 млрд долларов. На тот момент сервисом пользовались более 28 миллионов программистов.

# **Git. Базовые понятия**

Репозиторий (Repository) - это хранилище, где хранятся все файлы и история изменений проекта. Он может быть локальным (на вашем компьютере) или удаленным (на сервере или в облачном хранилище).

Коммит (Commit) - состояние репозитория в определенный момент времени. Каждый коммит содержит информацию о внесенных изменениях, авторе коммита и времени его создания. Коммиты являются основными строительными блоками истории проекта.

Ветка (Branch) представляет собой отдельную линию разработки, основанную на определенном коммите. Ветки позволяют работать над разными функциональностями или исправлениями ошибок независимо друг от друга. Они позволяют изолировать изменения и вносить исправления без влияния на основную ветку разработки.

Слияние (Merge) - это процесс объединения изменений из одной ветки с другой. Когда вы хотите включить изменения из одной ветки в другую (например, объединить ветку функциональности в основную ветку), вы выполняете операцию слияния.

# **GIT. Базовые понятия**

Откат (Revert) позволяет отменить изменения, сделанные в определенном коммите, создавая новый коммит, который отменяет эти изменения. Он полезен, когда вам нужно исправить ошибку или отменить нежелательные изменения.

Удаленный репозиторий (Remote Repository) - это репозиторий, расположенный на сервере или в облачном хранилище. Он служит для совместной работы и обмена изменениями между разработчиками. Наиболее популярными хостинг-провайдерами для удаленных репозиторий являются GitHub, GitLab и Bitbucket.

Клонирование (Clone) - это процесс создания локальной копии удаленного репозитория. Клонирование позволяет вам получить полную историю изменений и все файлы проекта для работы с ними на вашем компьютере.

Ветка "master" (или "main"): Ветка "master" (или "main") представляет основную ветку разработки проекта. Она обычно содержит стабильную версию кода и считается основной веткой, от которой отводятся другие ветки для разработки новых функциональностей или исправления ошибок.

# Git. Базовые понятия

Ответвление (Fork): Ответвление - это создание копии удаленного репозитория в вашем профиле на хостинг-провайдере, таком как GitHub. Оно позволяет вам свободно вносить изменения в проект без прямого доступа к исходному репозиторию. После внесения изменений вы можете предложить их включить в исходный проект через процесс запроса на слияние (Pull Request).

Ветка "develop": Помимо ветки "master", часто используется ветка "develop", которая представляет собой ветку для активной разработки новых функциональностей. Она может служить промежуточным этапом перед объединением изменений в ветку "master".

Конфликт слияния (Merge Conflict): Конфликт слияния возникает, когда Git не может автоматически объединить изменения из двух веток. Это происходит, когда одна и та же часть файла была изменена в обеих ветках. Разрешение конфликта требует ручного вмешательства для выбора правильных изменений.

Ветвление и слияние по модели GitFlow - популярная модель ветвления и слияния, которая предлагает определенную структуру для управления разработкой проекта. Она использует основные ветки "master" и "develop", а также дополнительные ветки, такие как "feature" (для разработки новых функциональностей) и "release" (для подготовки релизов).



# **Git. Базовые понятия**

Индекс (Index) и рабочая директория (Working Directory): Индекс представляет собой промежуточную область, где вы можете подготовить изменения перед коммитом. Рабочая директория - это ваша локальная файловая система, где вы вносите изменения в файлы проекта.

Ветка "hotfix" - используется для быстрого исправления критических ошибок в продакшн-версии проекта. Она позволяет отклониться от нормального рабочего процесса и сразу внести исправление без влияния на остальную разработку.

.gitignore - содержит список файлов и папок, которые Git должен игнорировать при отслеживании изменений. Это позволяет исключить временные файлы, конфиденциальные данные, скомпилированный код и другие файлы, которые не должны попадать в репозиторий.

## 2. GIT. Работа в локальном репозитории.

Решим следующую задачу: у нас есть проект веб-сайта, состоящий из нескольких файлов и каталогов. Задача состоит в том, чтобы перевести работу с этим сайтом в Git.

### ОБЩАЯ СХЕМА РАБОТЫ:

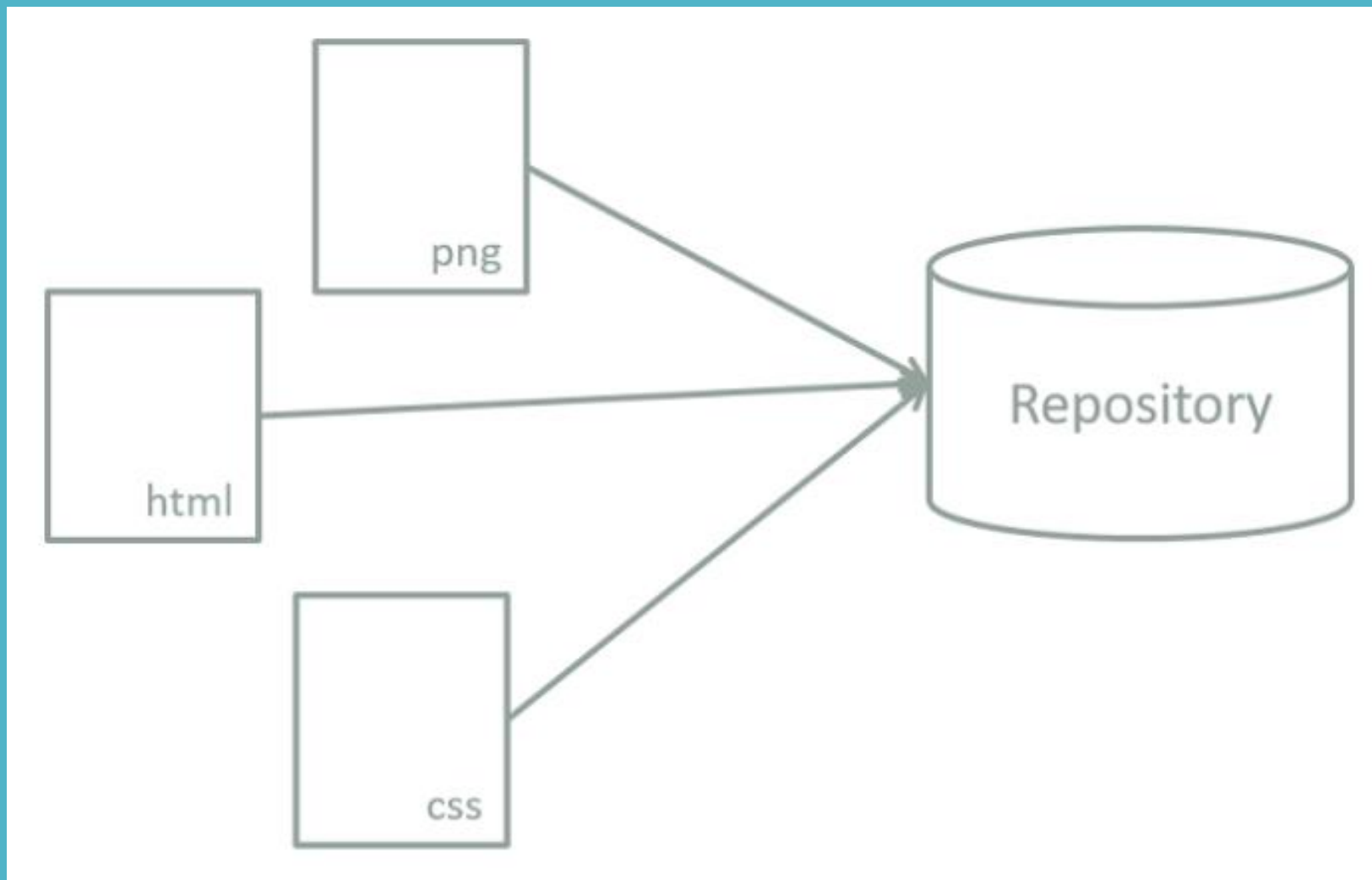
1. Создание локального репозитория для проекта;
2. Добавление файлов (изменённых) в список отслеживания (stage area или index);
3. Фиксация изменений в файлах (commit).

Операции 2 и 3 выполняются в течение всего развития проекта.

# **GIT. Установка.**

- Windows: <https://git-scm.com/download/win>
- Mac: <https://git-scm.com/download/mac>
- Ubuntu: apt-get install git
- Fedora: dnf install git
- Другие версии Linux: <https://git-scm.com/download/linux>

Репозиторий — это хранилище истории и данных (файлов, каталогов) вашего проекта.



# СОЗДАНИЕ РЕПОЗИТОРИЯ

Репозиторий создаётся в конкретном каталоге с помощью команды:

```
git init
```

Если команда сработала без ошибок, то вы увидите ответ:

```
Initialized empty Git repository in Диск:/папка/.git/
```

В результате выполнения команды `git init` появится подкаталог `.git` в котором и будут храниться служебные настройки git'a и сам репозиторий.



# ДОБАВЛЕНИЕ ФАЙЛОВ

Git будет следить только за теми файлами, которые вы добавили в «список отслеживаемых».

Сделать это можно с помощью команды:

```
git add index.html
```

где `index.html` — это имя добавляемого файла.

# МАСКИ ФАЙЛОВ

Есть ряд специальных символов, которые позволяют упростить процесс работы с файлами:

Специальный символ `*` означает любую последовательность символов.

- `git add *` добавит в список отслеживаемых файлов все файлы в текущем каталоге и подкаталогах;
- `git add *.js` добавит в список отслеживаемых файлов все файлы с расширением `.js` в текущем каталоге и подкаталогах.

# ТЕКУЩИЙ СТАТУС

Команда `git status` позволяет отследить текущий статус нашего репозитория:

```
git status
```

В результате выполнения команды вы увидите вывод:

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
    new file:   index.html
```

# ФИКСАЦИЯ ИЗМЕНЕНИЙ (КОММИТ)

Для фиксации изменений используется команда `git commit` (после этого откроется редактор для указания комментария). В редакторе nano нужно будет нажать клавиши `Ctrl + O`, `Ctrl + X` для записи и выхода из редактора (см. сокращения).

```
GNU nano 3.1 C:/project/.git/COMMIT_EDITMSG
|
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch
#
# Initial commit
#
# Changes to be committed:
#   new file:   README.md
#   new file:   css/style.css
#   new file:   index.html
#   new file:   js/app.js
#
```

# ПОСЛЕ ФИКСАЦИИ

После коммита мы получаем «чистое» состояние, готовое к новому добавлению изменений и фиксации. Проверим это командой `git status`:

```
git status
```

Результат:

```
On branch master  
nothing to commit, working tree clean
```



working  
directory

staging area  
(index)

repository

создание, изменение  
файлов



Добавление в Stage



Commit



## Добавим изменения в файл.

```
git status
```

Результат:

```
On branch master
```

```
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
    modified:   index.html
```

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

## Добавим изменения в файл.

Теперь согласно схеме, мы должны:

1. Добавить файл в stage с помощью команды `git add index.html`
2. Закоммитить изменения с помощью команды `git commit`.

```
git add index.html  
git commit
```

```
hint: Waiting for your editor to close the file...  
[master e2e7c64] Добавлены Google Fonts  
1 file changed, 13 insertions(+)
```

# КОГДА ДЕЛАТЬ `git add`

**Важно запомнить следующий момент:** команда `git commit` фиксирует только те изменения, которые были добавлены в staging area через `git add`.

Поэтому если вы сделаете `git add` для файла, а затем измените его и сделаете `git commit`, то ваши последние изменения не зафиксируются, так как вы не сделали `git add`.

# СОКРАЩЕНИЯ

Поскольку операции добавления в stage файлов и их фиксации очень частые, то в Git для этого есть специальное сокращение, позволяющее сделать всё одной командой:

```
git commit -a -m "Добавлены Google Fonts"
```

```
[master 075a656] Добавлены Google Fonts  
1 file changed, 1 insertion(+)
```

# СОКРАЩЕНИЯ

Поскольку операции добавления в stage файлов и их фиксации очень частые, то в Git для этого есть специальное сокращение, позволяющее сделать всё одной командой:

```
git commit -a -m "Добавлены Google Fonts"
```

```
[master 075a656] Добавлены Google Fonts  
1 file changed, 1 insertion(+)
```

Флаг `-a` (сокращение от `--all`) говорит о том, что мы добавляем в stage все удалённые/изменённые файлы (но не новые, новые нужно добавлять отдельно).

Флаг `-m "Сообщение коммита"` (сокращение от `--message="Сообщение коммита"`) позволяет не открывать редактор, а указывать сообщение прямо в командной строке.

# ИСТОРИЯ

Посмотреть историю коммитов можно с помощью команды `git log`.

```
git log
```

Результат:

```
commit 075a656ac42d7b892e12f309cdc547603ad53577 (HEAD -> master)
```

```
Author: Vasya Pupkin <vasya@localhost>
```

```
Date:   Fri Oct 26 19:42:48 2018 +0400
```

Добавлены Google Fonts

```
commit 4ad7e09ee8d76ce6c6f25e233cf186689b3b55b2
```

```
Author: Vasya Pupkin <vasya@localhost>
```

```
Date:   Fri Oct 26 19:26:05 2018 +0400
```

# ИДЕНТИФИКАТОР КОММИТА

Перед коммитом файла Git вычисляет контрольную сумму, которая является идентификатором коммита.

```
git commit
```

```
[master (root-commit) 4ad7e09] Первоначальная версия  
4 files changed, 14 insertions(+)  
create mode 10064 README.md  
create mode 10064 css/style.css  
create mode 10064 index.html  
create mode 10064 js/app.js
```

```
git log
```

```
commit 4ad7e09ee8d76ce6c6f25e233cf186689b3b55b2 (HEAD -> master)
```

```
Author: Vasya Pupkin <vasya@localhost>
```

```
Date:   Fri Oct 26 19:26:05 2018 +0400
```



# ИДЕНТИФИКАТОР КОММИТА

Посмотреть полную информацию о коммите мы можем с помощью команды `git show id-коммита`. ID не обязательно писать целиком, достаточно первых нескольких символов:

```
git show 4ad7e0
```

```
commit 4ad7e09ee8d76ce6c6f25e233cf186689b3b55b2
```

```
Author: Vasya Pupkin <vasya@localhost>
```

```
Date:   Fri Oct 26 19:26:05 2018 +0400
```

Первоначальная версия

```
diff --git a/README.md b/README.md
```

```
new file mode 100644
```

```
index 0000000..e69de29
```

# ИСПРАВЛЯЕМ ОШИБКИ

Если вы случайно добавили в stage area файл, который добавлять не нужно, то удалить его можно командой:

```
git rm --cached stuff.txt
```

где `stuff.txt` — имя файла.

Если вы случайно зафиксировали коммит с ошибочным комментарием, то исправить комментарий последнего коммита можно с помощью команды:

```
git commit --amend -m "Roboto Font"
```

где `"Roboto Font"` — новое сообщение коммита.

# ИСПРАВЛЯЕМ ОШИБКИ

Если вы залили коммит с ошибкой, то можно создать «зеркальный» коммит, который отменит действие предыдущего:

```
git revert id-коммита
```

где `id-коммита` — идентификатор коммита.

Идентификатор коммита можно посмотреть с помощью команды `git log`.

# СПРАВКА

Вся справка по командам предоставляется самим Git:

- `git` – краткая справка по git;
- `git help команда` – справка по конкретной команде git.

Например, `git help add` покажет справку для команды `git add`.

### 3. .gitignore

Часто бывает, что в проектах бывают какие-то файлы, которые не нужно хранить в репозитории, например:

- файлы проектов тестовых редакторов;
- файлы операционных систем;
- генерируемые данные (результаты сборки, компиляции и т.д.);
- файлы с учётными данными;
- и любые другие, которые вы не хотите хранить в репозитории.

**Помните, что всё, что хранится в репозитории будет доступно всем.**

# ИГНОРИРОВАНИЕ ФАЙЛОВ

Для того, чтобы игнорировать подобные файлы в Git есть специальный настроечный файл, который называется `.gitignore` (точка в начале имени файла обязательна!).

В нём мы можем перечислить файлы и каталоги, которые Git должен игнорировать при работе.

Это обычный текстовый файл, где на каждой строке размещается одно правило игнорирования.

Обычно этот файл располагается в самом верхнем каталоге проекта и хранится в репозитории.

Строки, начинающиеся с символа `#` являются комментарием и не воспринимаются как правила.

# ИГНОРИРОВАНИЕ ФАЙЛОВ

```
# будут игнорироваться все файлы и каталоги Thumbs.db
# вне зависимости от того, в каком каталоге они находятся
Thumbs.db

# будет игнорироваться каталог tmp
# вне зависимости от того, в каком каталоге он находится
# слэш в конце указывает, что это каталог
tmp/

# будет игнорироваться относительно файла .gitignore
# чаще всего относительно всего проекта
/tmp/

# будут игнорироваться все файлы и каталоги с расширением .txt
# вне зависимости от того, в каком каталоге они находятся
*.txt
```



Microsoft



GitHub



## 4. GITHUB

Достаточно опасно хранить всю историю работы с нашим проектом только на нашем жёстком диске (локально) – при возникновении какой-либо проблемы с ним или компьютером мы можем на достаточно долгое время потерять доступ к исходным кодам проекта (если не навсегда).

Конечно, есть сервисы вроде Dropbox, Google Drive, Яндекс.Диск и другие, но гораздо удобнее использовать специализированные решения, интегрированные с Git.

# GITHUB. РЕГИСТРАЦИЯ.

Смотри мануал:

[https://github.com/ShViktor72/Education/blob/main/software%20design/lesson\\_8/GIT\\_Registration.md](https://github.com/ShViktor72/Education/blob/main/software%20design/lesson_8/GIT_Registration.md)

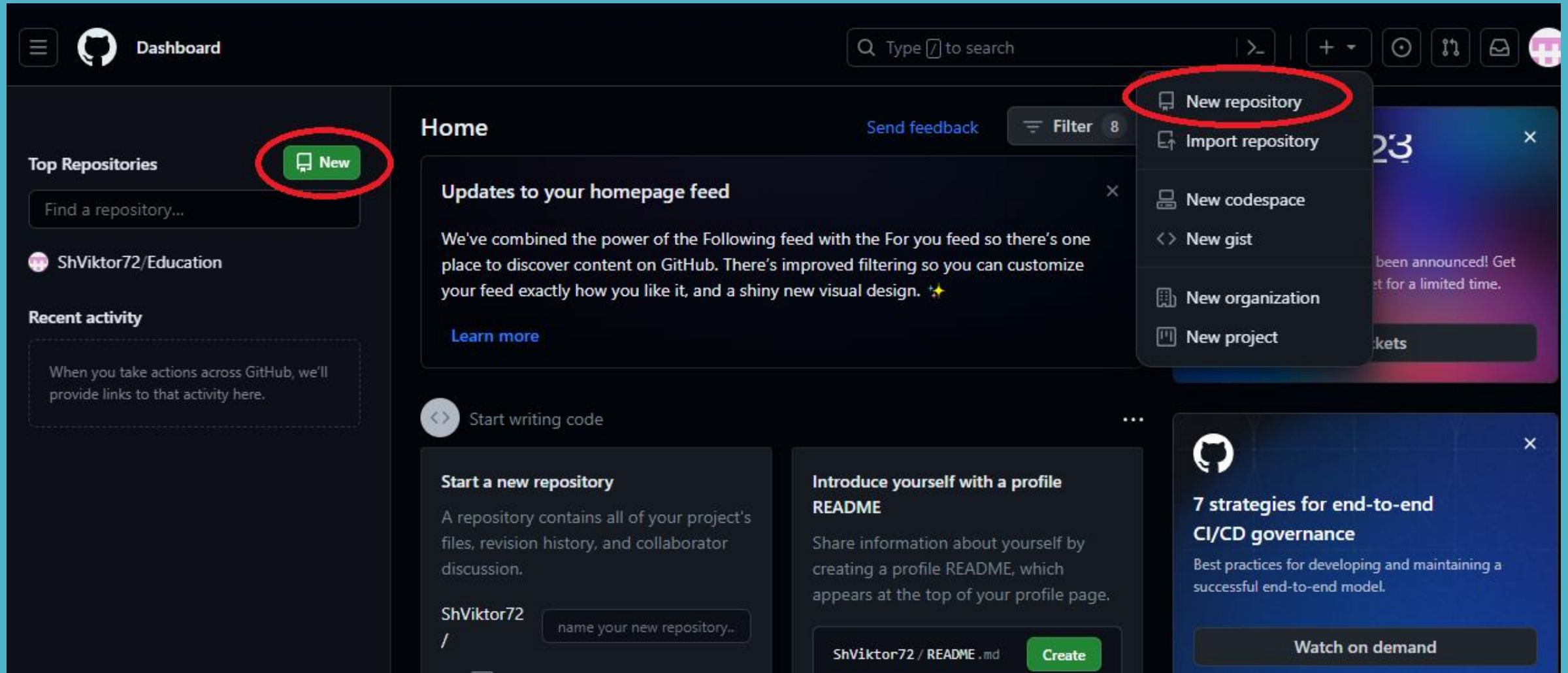
# НАЧАЛО РАБОТЫ

В качестве базовых сценариев начала работы мы на данной лекции выделим два:

1. У вас есть локальный репозиторий с проектом, вы хотите к нему привязать репозиторий на GitHub;
2. У вас нет локального репозитория, вы хотите начать новый проект.



На самом деле, и в том, и в другом случае, нам понадобится репозиторий на GitHub, поэтому посмотрим как его создавать.

# НАЧАЛО РАБОТЫ



На домашней странице github нажмите кнопку «Новый репозиторий»

# НАЧАЛО РАБОТЫ


 New repository

Create a new repository

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

Required fields are marked with an asterisk (\*).

Owner \*


 ShViktor72

Repository name \*


✔ new\_project is available.

Great repository names are short and memorable. Need inspiration? How about [fictional-octo-doodle](#) ?

Description (optional)


☒  Public


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

☐  Private

Выберите имя.

# НАЧАЛО РАБОТЫ

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

☐  **Private**  
You choose who can see and commit to this repository.

---

**Initialize this repository with:**

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more about READMEs.](#)

**Add .gitignore**

.gitignore template: **None** ▾


Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

**Choose a license**

License: **None** ▾

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

---

 You are creating a public repository in your personal account.

---

Create repository

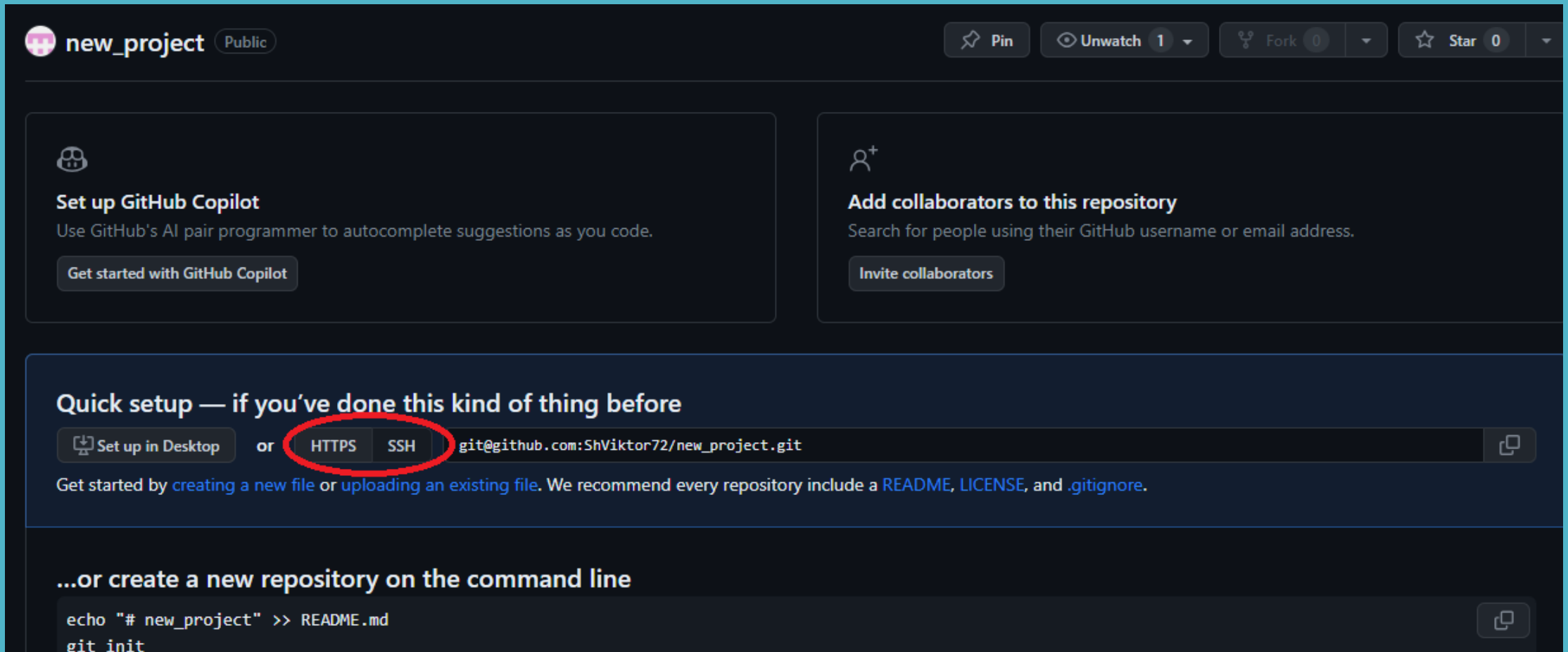
Далее остается нажать на кнопку «Create Repository»

# НАЧАЛО РАБОТЫ

1. **Name** — имя репозитория, вводится английскими буквами;
2. **Description** — описание репозитория (опционально);
3. **Include README** — автоматическое создание `README.md` (будем обсуждать чуть позже);
4. **Add .gitignore** — добавление преднастроенного `.gitignore`;
5. **Add a license** — добавление информации о лицензии.

В рамках этой лекции нам нужны только пункты 1 и 2, остальное всё оставьте пустым.

# НАЧАЛО РАБОТЫ



После этого открывается страница быстрых настроек репозитория.  
Нужно выбрать метод аутентификации (рекомендуется SSH)

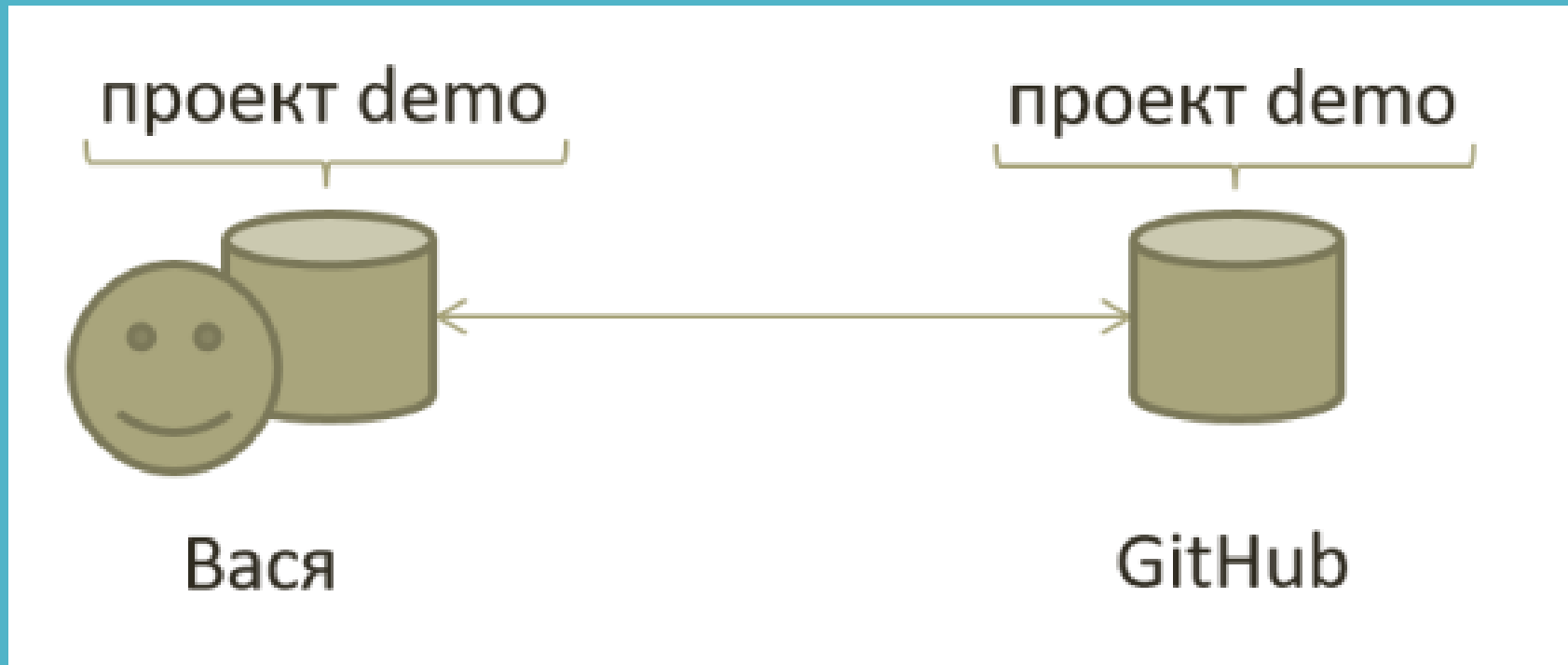


# Добавление ssh-ключа на GitHub

Смотри мануал:

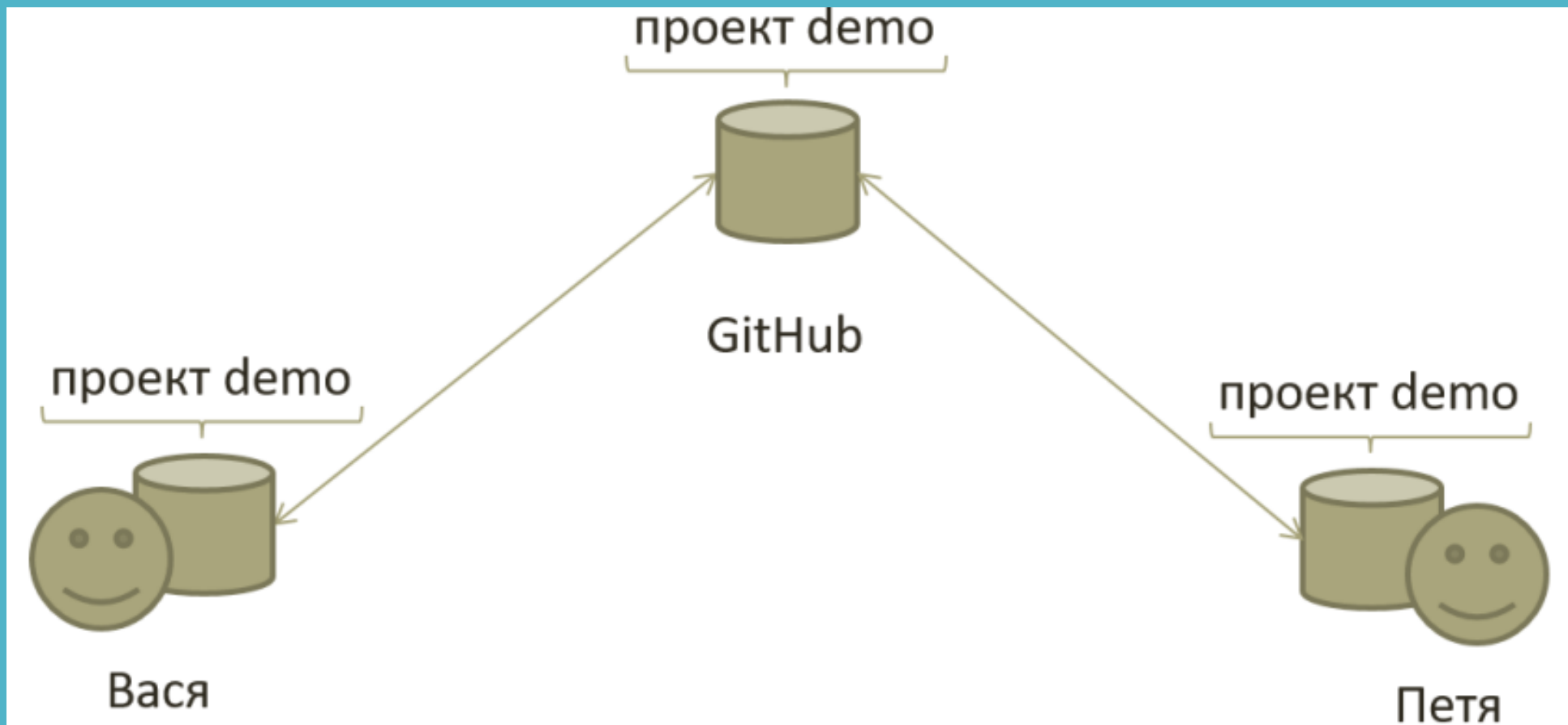
[https://github.com/ShViktor72/Education/blob/main/software%20design/lesson\\_8/Manual\\_Git\\_SSH\\_Windows.html](https://github.com/ShViktor72/Education/blob/main/software%20design/lesson_8/Manual_Git_SSH_Windows.html)

# УДАЛЁННЫЕ РЕПОЗИТОРИИ



Git позволяет нам привязать к нашему репозиторию удалённый репозиторий, так, чтобы мы могли туда отправлять изменения из своего репозитория и получать обновления с удалённого репозитория (если туда их ещё кто-то отправляет).

# УДАЛЁННЫЕ РЕПОЗИТОРИИ



Удалённые репозитории могут использоваться для совместной работы и привязать их можно много.

# КЛОНИРОВАНИЕ РЕПОЗИТОРИЯ

Операция `git clone` позволяет нам клонировать удалённый репозиторий, т.е. буквально взять почти всю информацию, хранящуюся в удалённом репозитории и создать копию на нашем компьютере – это очень здорово, потому что теперь даже если с удалённым репозиторием что-то случится, вся история разработки проекта со всеми версиями будет на нашем компьютере.

Чтобы его клонировать, нам необходим URL, по которому располагается этот репозиторий.

# КЛОНИРОВАНИЕ РЕПОЗИТОРИЯ

После того, как у нас есть URL мы можем в командной строке выполнить команду `git clone`:

```
git clone https://github.com/netology-git/demo.git
```

```
Cloning into 'demo'...
```

```
warning: You appear to have cloned an empty repository
```

Переходим в появившуюся папку:

```
cd demo/
```

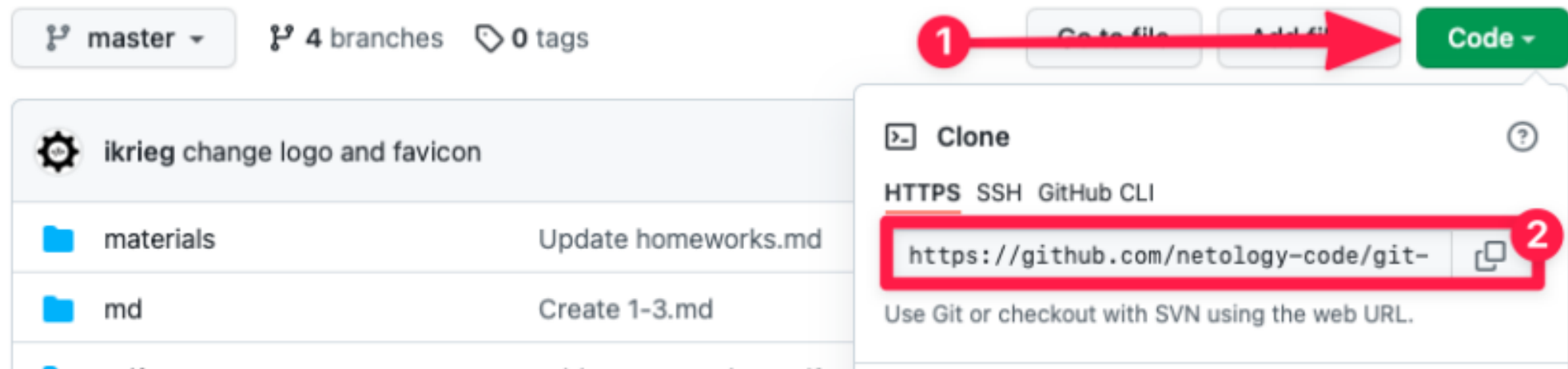
\* команда `cd demo/` нужна чтобы зайти в папку скопированного репозитория.

После успешной операции клонирования, создастся каталог `demo`, в котором будет наш репозиторий (уже привязанный к удалённому).

# КЛОНИРОВАНИЕ РЕПОЗИТОРИЯ

*Важно: необязательно клонировать пустой репозиторий, вы можете клонировать любой репозиторий, зная его URL.*

URL вы можете найти на странице проекта:



# git remote

Команда `git remote` позволяет нам управлять удалёнными репозиториями (добавлять, удалять, просматривать).

Например, мы можем в только что клонированном репозитории посмотреть remote:

```
git remote -v
```

```
origin  https://github.com/netology-git/demo.git (fetch)  
origin  https://github.com/netology-git/demo.git (push)
```

Общепринято, что первый удалённый репозиторий называют `origin`.

# РАБОТА С РЕПОЗИТОРИЕМ

Дальше мы работаем с нашим репозиторием так же, как с обычным локальным ровно до тех пор, пока не захотим отправить изменения, которые мы внесли, на удалённый репозиторий.



## ДОБАВЛЕНИЕ remote

Рассмотрим второй вариант, когда у нас уже есть локальный репозиторий с проектом, и мы хотим к нему подключить удалённый.

Для этого используем также команду `git remote`:

```
git remote add origin https://github.com/netology-git/demo.git
```

Проверим, что локальный репозиторий успешно связался с удалённым:

```
git remote -v
```

```
origin  https://github.com/netology-git/demo.git (fetch)
origin  https://github.com/netology-git/demo.git (push)
```

# ОТПРАВКА ИЗМЕНЕНИЙ

После того, как мы поработали локально, необходимо отправить наши изменения в удалённый репозиторий. Для этого используется команда `git push`, для первой отправки: `git push -u origin master`.

```
git push -u origin master
```

```
Username for 'https://github.com': ваш-логин
```

```
Enumerating objects: 10, done
```

```
Counting objects: 100% (10/10), done.
```

```
Compressing objects: 100% (7/7) done.
```

```
Writing objects: 100% (10/10), 1.13 KiB | 580.00 KiB/s, done.
```

```
Total 10 (delta 2), reused 0 (delta 0)
```

```
remote: Resolving deltas: 100% (2/2), done.
```

```
remote:
```

```
remote: Create a pull request for 'master' on GitHub by visiting:
```

```
remote:      https://github.com/netology-git/demo/pull/new/master
```

```
remote:
```

```
To https://github.com/netology-git/demo.git
```

```
* [new branch]      master -> master
```

```
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

# ОТПРАВКА ИЗМЕНЕНИЙ

При отправке изменений вас попросят ввести логин и пароль. При вводе пароля в командной строке вводимые символы не будут отображаться. Это нормальное поведение, не пугайтесь.

При последующих отправках достаточно использовать команду `git push`.

*Важно: мы с вами ещё не проходили ветки и конфликты, поэтому не вносите параллельные изменения с разных компьютеров. До следующей лекции делайте одно клонирование с одного репозитория и все изменения отправляйте с одного локального репозитория.*

Посмотреть, что изменения отправились, можно на веб-страничке проекта

## 5. MARKDOWN

Облегчённый язык разметки, который позволяет форматировать текст и затем преобразовывать его в другие форматы, например, HTML.

Чаще всего, документ, оформленный с использованием языка Markdown, хранится в текстовом файле с расширением `.md`.

На сервисе GitHub принято описание, содержащееся в специальном файле с именем `README.md` выводить в качестве описания проекта.

# ЗАГОЛОВКИ

Заголовки – так же, как и в HTML поддерживается несколько уровней заголовков: от 1 до 6

- 1    **# Заголовок первого уровня**
- 2    **## Заголовок второго уровня**
- 3    **### Заголовок третьего уровня**
- 4    **#### Заголовок четвертого уровня**
- 5    **##### Заголовок пятого уровня**
- 6    **##### Заголовок шестого уровня**

# ТЕКСТ: ОБЫЧНЫЙ И СТИЛИЗОВАННЫЙ

Обычный текст никакой специальной разметкой не оформляется.

Жирный, наклонный и перечёркнутый текст:

- 1 | **\*\*Жирный текст\*\***
- 2 | *\*Наклонный текст\**
- 3 | ~~Перечеркнутый текст~~

# СПИСКИ

- |   |  |                                  |
|---|--|----------------------------------|
| 1 |  | * Элемент списка                 |
| 2 |  | * Элемент списка                 |
| 3 |  | * Вложенный элемент списка       |
| 4 |  | * Вложенный элемент списка       |
| 5 |  | 1. Элемент упорядоченного списка |
| 6 |  | 1. Элемент упорядоченного списка |
| 7 |  | 1. Вложенный элемент списка      |
| 8 |  | 1. Вложенный элемент списка      |



- Элемент списка
- Элемент списка
  - Вложенный элемент списка
  - Вложенный элемент списка
- 1. Элемент упорядоченного списка
- 2. Элемент упорядоченного списка
  - 1. Вложенный элемент списка
  - 2. Вложенный элемент списка

# ГИПЕРССЫЛКИ

Гиперссылки оформляются в формате [Текст ссылки](url-адрес) :

[Текст ссылки](http://localhost)



Текст ссылки



# КОД

Есть два варианта оформления кода:

- Внутри строки: код заключается в бэктики (backticks): ``строка кода``
- Отдельным блоком:

```
```javascript  
console.log("");  
```
```

Для подсветки синтаксиса после верхних бэктиков укажите технологию, на которой написан кусок кода. Например, в блоке выше код написан на JavaScript. Этот язык мы и указываем.

# ИТОГИ:

Сегодня мы разобрали вопросы:

1. Предназначение системы контроля версий Git;
2. Работа с локальным репозиторием;
3. Привязка удалённого репозитория и GitHub;
4. Язык разметки Markdown.