# Версионирование кода. Git

Белов Виталий, весна 2021

# Что это? 

Git - система контроля версий.

С его помощью мы можем гибко управлять разработкой программного обеспечения.

Другие системы контроля версий: SVN, Mercurial и пр.

Преимущства Git:

- OpenSource
- Криптографическая целостность. При каждом коммите генерируются контрольная сумма
- Удобная система ветвления
- Быстрый

# Зачем?

- Если больше одного разработчика
- Даже если один разработчик, имеет смысл пользоваться git
    - Возможность откатиться до рабочей версии, если что-то пошло не так
    - Разработка фич в разных ветках
    - Полная история изменений
    - Удаленный репозиторий - можно быстро продолжить работу с любого компьютера и передать код другим разработчикам
    - Портфолио разработчика

# Основы Git

# Установка

*Ubuntu*

```
$ sudo apt install git
```

*MacOS*

```
$ brew install git
```

*Windows*

Устанавливаем Git Bash:

```
http://git-scm.com/download/win
```

Регистрируемся на https://gitlab.com - будем использовать на курсе

Другие популярные варианты: GitHub, Bitbucket, etc

# Настройка

## 1. **Информация о пользователе**

```
$ git config --global user.name "username"
$ git config --global user.email mail@gmail.com
```

Каждый коммит будет снабжен справочной информацей: хеш, пользователь, дата коммита, описание коммита и пр.

2. **Доступ к удаленному репозиторию с помощью SSH ключа:**

   Создаем ключ (через терминал в MacOS или Linux, через Git Bush в Windows)

   - `ssh-keygen`
     Просмотреть ключи можно в `~/.ssh`

   Копируем содержимое файла <key_name>**.pub** в настройки GitLab.

   - User Settings -> SSH Keys -> Add key

# Создаем локальный репозиторий

Переходим в директорию проекта (если он уже есть):

```
$ cd /home/user/my_project
```

Инициализируем репозиторий

```
$ git init
```

Эта команда создаст директорию .git, содержащую настройку репозитория и объекты системы git, отражающие историю изменений и структуру проекта.

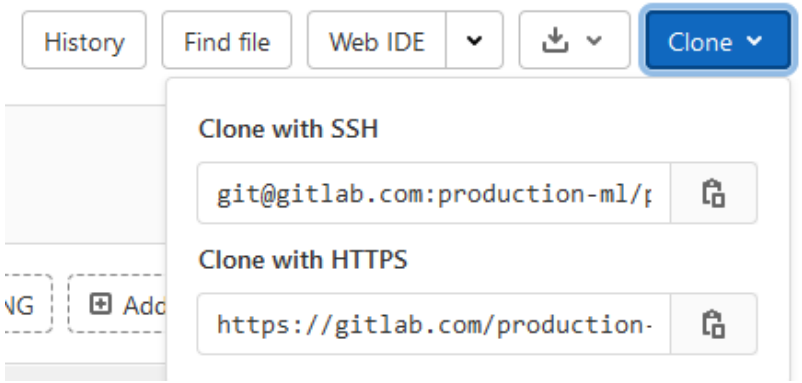
Более подробно, что внутри Git

# Клонируем репозиторий

Это создание идентичной копии удаленного репозитория Git на локальной машине.

Переходим в ветку, в которой планируем создать копию репозитория

```
$ git clone git@gitlab.com:production-ml/password_app.git
$ cd password_app
```




Лучше использовать SSH

# Ответвление проекта. Git Fork

Fork - копия на GitLab оригинального репозитория.

Используем эту опцию, если у нас нет прав на внесение изменений в оригинальный репозиторий.

С помощью merge request можем предложить владельцу оригинального репозитория внести изменения в код. (Open Source projects)

# Внесение изменений

Проверить состояние репозитория:

```
$ git status
```

Появится сообщение вида:

```
На ветке master
Ваша ветка обновлена в соответствии с «origin/master».
нечего коммитить, нет изменений в рабочем каталоге
```

Добавляем новый файл *new_file* и снова проверяем статус:

```
На ветке master
Ваша ветка обновлена в соответствии с «origin/master».

Неотслеживаемые файлы:
  (используйте «git add <файл>…»,
  чтобы добавить в то, что будет включено в коммит)
        new_file

ничего не добавлено в коммит, но есть неотслеживаемые файлы
(используйте «git add», чтобы отслеживать их)
```

Для того, чтобы проиндексировать изменения выполняем:

```
$ git add new_file
```

Проверяем:

```
На ветке master
Ваша ветка обновлена в соответствии с «origin/master».

Изменения, которые будут включены в коммит:
  (use "git restore --staged <file>..." to unstage)
        новый файл:    new_file

Файл теперь находится под версионным контролем.

Файл проиндексирован, но не закоммичен.
```

Чтобы зафиксировать изменения в проиндексированных файлах выполняем:

```
$ git commit -m 'описание'
```

```
[master 843b7c6] added new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 new_file
```

Советы:

- Один коммит должен решать одну проблему.
- Не следует вносить множество изменений в один коммит - это усложнит чтение репозитория и его поддержку.
- Индексация нужна, чтобы отделить изменения, которые принадлежат одному коммиту.

# Просмотр изменений

# Историю коммитов можно просмотреть командой:

```
$ git log
```

```
commit 08dfa75682c9c570640fb32d8341915452be8492
Merge: c6c1da8 3671364
Author: aguschin <1aguschin@gmail.com>
Date:   Mon Jan 18 12:53:21 2021 +0300

    Merge commit '367136443a8a5d67db1e95e59562bea69940edd7'

commit c6c1da883f58d3b514086e77ce33761ade5585d2
Author: aguschin <1aguschin@gmail.com>
Date:   Mon Jan 18 12:52:58 2021 +0300

    update ci
```

# История с графическим отображением веток (более наглядное представление):

```
$ git log --graph --oneline
```

```
* 006b9ce (HEAD -> feature1, origin/feature1) DevOps structure defined
* b3378b2 small edit in lib welcome.sh
* f405de0 modified the last edit date
* 578de10 Editing the common lib file
* 1c4c923 Topic branch: iss54 created as admin user
* 6e6e9de edited temp file as an admin user
*   c366419 Merge remote-tracking branch 'origin/master'
|\
| * 4920adc edited temp file from admin user
* | feeb30c (tag: database_r1.0) modified db.log file as greets user
|/
* 9975f27 edited temp file as greets user
```

# Правила исключений Git. Gitignore

Временные файлы, логи и т.д. мы бы не хотели добавлять в репозиторий. Имеет смысл заранее настроить правила, чтобы исключить их их попадание в индексацию.

Для этого в корневой директории нужно создать файл **.gitignore**

```
__pycache__/
*.egg
*.py[cod]
.pytest_cache/
```

Готовый набор шаблонов

# Просмотр изменений. Git diff

Просмотр всех изменений в непроиндексированных файлах

```
$ git diff
```

Если файлы уже были проиндексированы:

```
$  git diff --cached
```

Просмотр изменений в определенном файле

```
$ git diff app.py
```

Режим выделения измененных слов цветом:

```
$ git diff --color-words
```

Чтобы при этом git стал понимать python синтаксис, нужно создать **.gitattributes** в корневом каталоге и добавить в нем:

```
*.py diff=python
```

# Работа в команде

# Работа с удаленным репозиторием

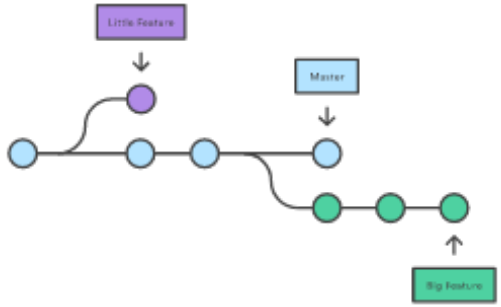
Для того, чтобы привести локальный репозиторий в соответствие с удаленным:

```
$ git pull
- равносильно: git fetch + git merge, где
- git fetch - подтягивает изменения с удаленного репозитория
- git merge - объединяет изменения
```

# Работа с ветками

Ветки (**Branches**) нужны для совместной работы над проектом.

Это дает возможность разрабатывать новые функции приложения, не мешая разработке в основной ветке (master).



## Основные команды:

Просмотр веток и активной ветки (будет помечена *):

```
$ git branch
```

Создать новую ветку:

```
$ git branch имя_ветки
```

---

Переключиться на ветку (обновляет указатель HEAD, чтобы он ссылался на указанную ветку или коммит):

```
$ git checkout имя_ветки
```

Удаление ветки:

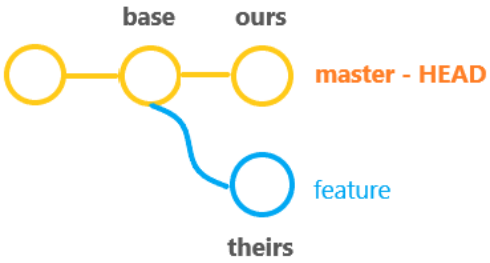
```
$ git branch -d <branch_name>
```

Сравнить две ветки:

```
$ git diff <branch_1> <branch_2>
```

---

# Слияние (Merge) и конфликты




Выполняется командой (находимся на ветке master)

```
$ git merge feature
```

Git пытается автоматически объединить изменения из двух веток. Если для одного участка кода внесены изменения в ветках и master и feature, возникает конфликт.

```
<<<<<<< HEAD
<code from branch master>
=======
<code from branch feature>
>>>>>>> feature
```

Разрешаем конфликт - выбираем желаемый код из ветки master или feature.

Другой вариант: выбрать версию кода одной из веток:

```
$ git checkout --ours <file> - выберет версию <file> из master
$ git checkout --theirs <file> - выберет версию <file> из feature
```

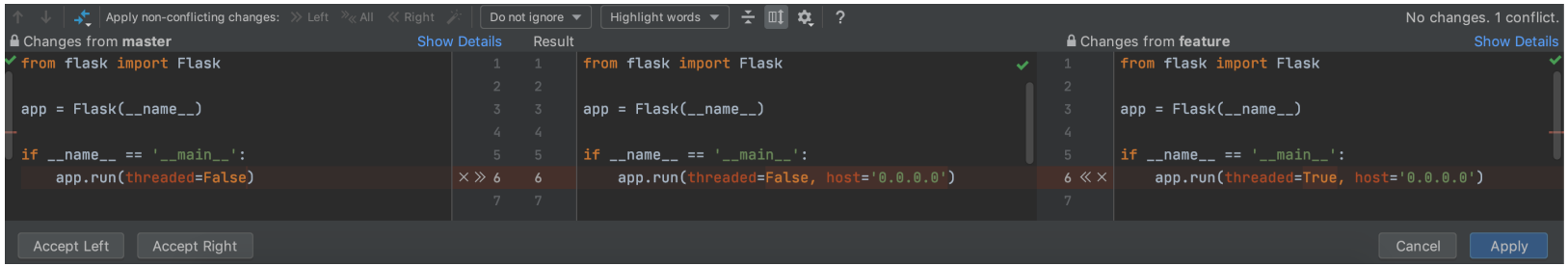
Завершаем слияние:

```
$ git add <file>
$ git merge --continue
```

Если передумали выполнять слияние:

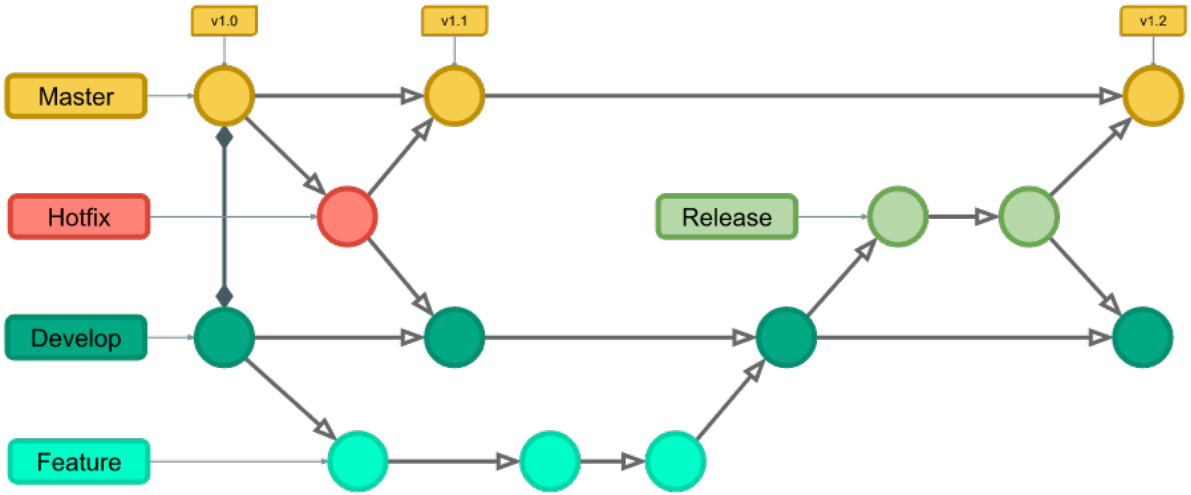
```
$ git merge --abort
```

Удобно разрешать конфликты в IDE, например PyCharm:



# Gitflow

Модель рабочего процесса Git для управления крупными проектами.
Gitflow определяет роли для веток и описывает характер и частоту взаимодействия между ними.



Инициализировать репозиторий со структурой gitflow: `$ git flow init` (требуется установка)

- Ветка **master** - хранится историю релизов
- Ветка **develop** - для объединения всех функций.
- Под каждую фичу содается отдельная ветка **feature/**, которая затем мерджится в develop.
- Ветка **release** создается на основе develop, когда версия продукта готова к выпуску.
- При готовности к выпуску **release** заливается в **master**.
- Ветки **hotfix** служат для исправлений в рабочие релизы. Ветки создаются от **master**, а после завершения исправлений вносятся в **master** и **develop**

# Git Hooks

Git Hooks — скрипты, которые запускаются автоматически до или после того, как Git выполняет Commit или Push или другую команду.

Git init создает примеры хуков, которрые можно посмотерть в **.git/hooks**

Писать Git hooks можно на любом скриптовом языке: Python, PHP, Bash.

Файл нужно сделать исполянемым: `chmod a+x .git/hooks/pre-commit`

## Готовый инструмент: pre-commit. Готовые хуки

```
$ install(brew) --user pre-commit
$ pre-commit sample-config > .pre-commit-config.yaml
```

.pre-commit-config.yaml:

```
repos:
-   repo: https://github.com/psf/black
    rev: 20.8b1
    hooks:
    -   id: black
```

Устанавляеваем pre-commit для репозитория:

```
$ pre-commit install
```

Проверить все файлы:

```
$ pre-commit run --all-files
```

# Артефакты

В процессе работы над проектом возникают артефакты - сборки, релизы, пакеты, отчеты, docker images и так далее.

Обычно, эти артефакты хранят в таких местах, как:

- Git (Git LFS)
- Object storage (s3)
- Artefactory (jFrog Artefactory, Nexus)

Для некоторых артефактов есть специальные хранилища:

- PyPi - Python package index
- Docker registry (Docker hub, private docker registry)
- Gitlab Package Registry, Github packages, Gemfury

# Git LFS

***Git Large File Storage**

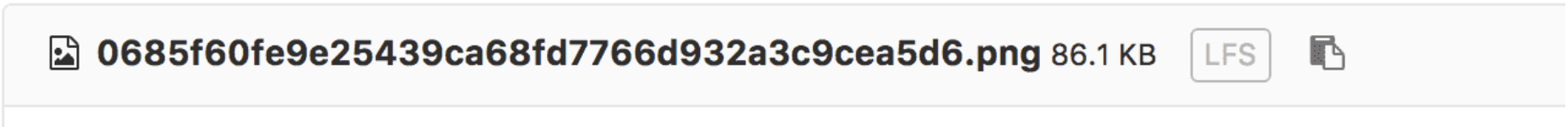
Расширение Git’a, предназначенное для версионирования больших файлов.

Если пользоваться классическим git, то каждое изменение больших файлов будет увеличивать репозиторий до значительных размеров.
Git LFS заменяет большие файлы текстовыми указателями.
Содержимое этих файлов сохраняется на удалённом сервере.

## Как использовать?

```
$ git lfs install
$ git lfs track "*.dat"
```

Последняя команда создаст файл .gitattributes (если его не создали до этого) и добавит в него запись *.dat filter=lfs diff=lfs merge=lfs -text



# Версионирование артефактов

Широко распространен подход Semantic Versioning

```
Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner,
3. PATCH version when you make backwards compatible bug fixes.
Additional labels for pre-release and build metadata are available as extensi
```

https://semver.org

# Семинар

1. Создаем новый проект (пустой)

- Создаем каталог проекта -> Переходим -> Создаем локальный репозиторий
- Настройки:

```
git config --global user.name "username"
git config --global user.email mail@gmail.com
# настройка редактора:
git config --global core.editor vim
# настроим alias - красивый лог:
git config --global alias.glog 'log --graph --abbrev-commit --decorate --
# Настроим LFS:
git lfs install
git lfs track "*.dat"
# добавим правила:
touch .gitignore
# Проверить конфиги:
git config --list --global

```

2. Простые операции на новом репозитории:
   - git add
   - git commit
3. Создаем ветку feature,
   - переключаемся, вносим изменения
   - возвращаемся в master
   - вносим конфликтующие изменения
   - пробуем git diff между ветками
   - git merge (на PyCharm или VSCode)
   - Отправляем его на свой аккаунт:

```
gitlab: git push --set-upstream https://gitlab.com/<your_gitlab_name>/new-pro
```

4. Клонируем репозиторий учебного проекта:
   - Добавляем SSH ключ к репозиторию

# Полезные ссылки

- https://git-scm.com/book/en/v2 - официальное руководство по Git (на русском)
- https://learngitbranching.js.org/ - интерактивный курс по основным операциям Git
- https://www.atlassian.com/git/tutorials - руководство от Atlassian