

Версионирование кода и артефактов

Борисенко Глеб
ФТиАД2022

IN CASE OF FIRE 



1. git commit



2. git push



3. git out!

Что это?

- Git - система контроля версий.
- С его помощью мы можем гибко управлять разработкой программного обеспечения.
- Другие системы контроля версий: SVN, Mercurial и пр.
- Преимущества Git:
 - OpenSource
 - Криптографическая целостность
 - При каждом коммите генерируется 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

- `sudo apt install git`
- ЛИБО `brew install git`
- ЛИБО <http://git-scm.com/download/win>
- Варианты хранилища вашего кода (удаленного репозитория):
 - Github
 - Gitlab
 - Bitbucket
 - etc

Информация о пользователе

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

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

Способы подключения

- https – копируете ссылку из поля git clone https, и каждый раз вводите креды. Можно делать трюк, чтобы не приходилось, но он не рекомендуется. А еще я этот трюк не делаю, но VSCode сам как-то за меня делает магию, и я не ввожу каждый раз креды
- ssh – более «прозрачный» способ, при котором можно использовать публик и прайват кей и забыть креды

Подключение через ssh

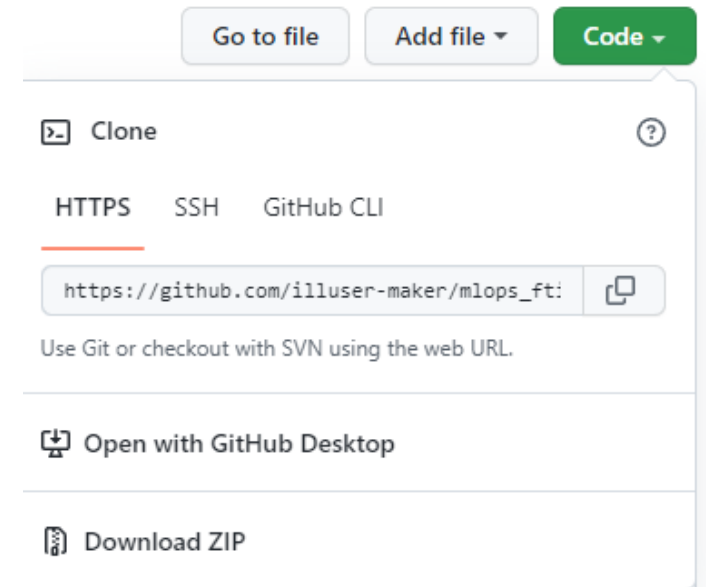
- Создаем ключ (через терминал в MacOS или Linux, через Git Bush в Windows)
 - `ssh-keygen`
 - Просмотреть ключи можно в `~/.ssh`
- Копируем содержимое файла `.pub` в настройки Github/GitLab.
 - User Settings -> SSH Keys -> Add key

Создание локального репозитория

- Переходим в директорию проекта (если он уже есть) и инициализируем репозиторий
- `git init`
- Эта команда создаст директорию `.git`, содержащую настройку репозитория и объекты системы `git`, отражающие историю изменений и структуру проекта.

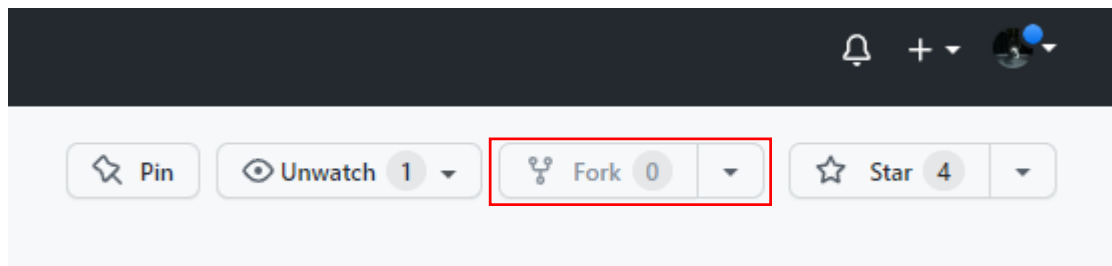
Клонируем репозиторий

- Это создание идентичной копии удаленного репозитория Git на локальной машине.
- Заходим на репозиторий нам нужный, там есть кнопка Code/Clone, там есть выбор способа клонирования
- Копируем ссылку (в зависимости от способа вами выбранного)
- В терминале выполняем команду:
 - `git clone <your_https_or_ssh>`
 - `git clone git@gitlab.com:production-ml/password_app.git`



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. (для Open Source projects)



Командочки

- Проверить состояние репозитория: `git status`

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

- Добавляем новый файл `new_file` и снова проверяем статус:

```
1 На ветке master
2 Ваша ветка обновлена в соответствии с «origin/master».
3
4 Неотслеживаемые файлы:
5   (используйте «git add <файл>...»,
6    чтобы добавить в то, что будет включено в коммит)
7     new_file
8
9 ничего не добавлено в коммит, но есть неотслеживаемые файлы
10 (используйте «git add», чтобы отслеживать их)
```

Командочки

- Для того, чтобы проиндексировать изменения, выполняем:
- `git add new_file`

```
1 На ветке master
2 Ваша ветка обновлена в соответствии с «origin/master».
3
4 Изменения, которые будут включены в коммит:
5   (use "git restore --staged <file>..." to unstage)
6     новый файл:    new_file
7
8 Файл теперь находится под версионным контролем.
9
10 Файл проиндексирован, но не закоммичен.
```

Командочки

- Чтобы зафиксировать изменения в проиндексированных файлах выполняем:
- `git commit -m 'описание'`

```
1 [master 843b7c6] added new file
2 1 file changed, 0 insertions(+), 0 deletions(-)
3 create mode 100644 new_file
```

- 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.

Командочки

- Историю коммитов можно просмотреть командой: `git log`

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

    Merge commit '367136443a8a5d67db1e95e59562bea69940edd7'

commit c6c1da883f58d3b514086e77ce33761ade5585d2
Author: [redacted] <[redacted]@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`

Gitignore

- Временные файлы, логи и т.д. мы бы не хотели добавлять в репозиторий. Имеет смысл заранее настроить правила, чтобы исключить их попадание в индексацию.
- Для этого в корневой директории нужно создать файл **.gitignore**

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

- Есть куча шаблонов в сети
- (например, <https://github.com/github/gitignore>)

Просмотр изменений

- 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`

А теперь работаем в команде

Работа с удаленным репозиторием

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

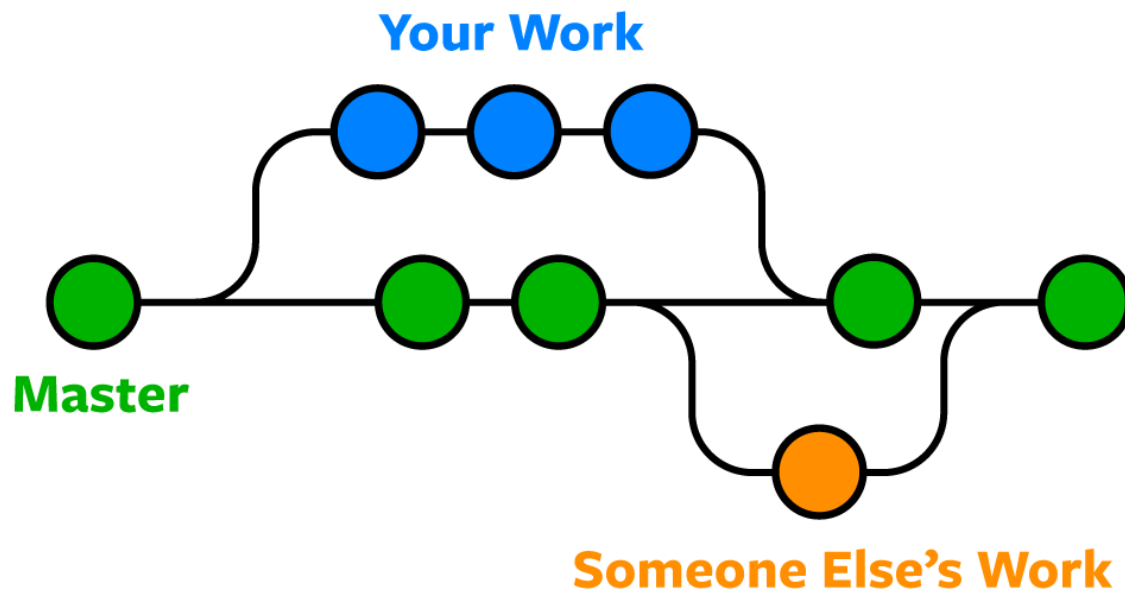
```
$ git pull
```

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

- Отправлять свои изменения на удаленный репозиторий можно через push:
 - `git push origin main`

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).



Работа с ветками

- 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`
- 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
```

Слияние (Merge) и конфликты

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

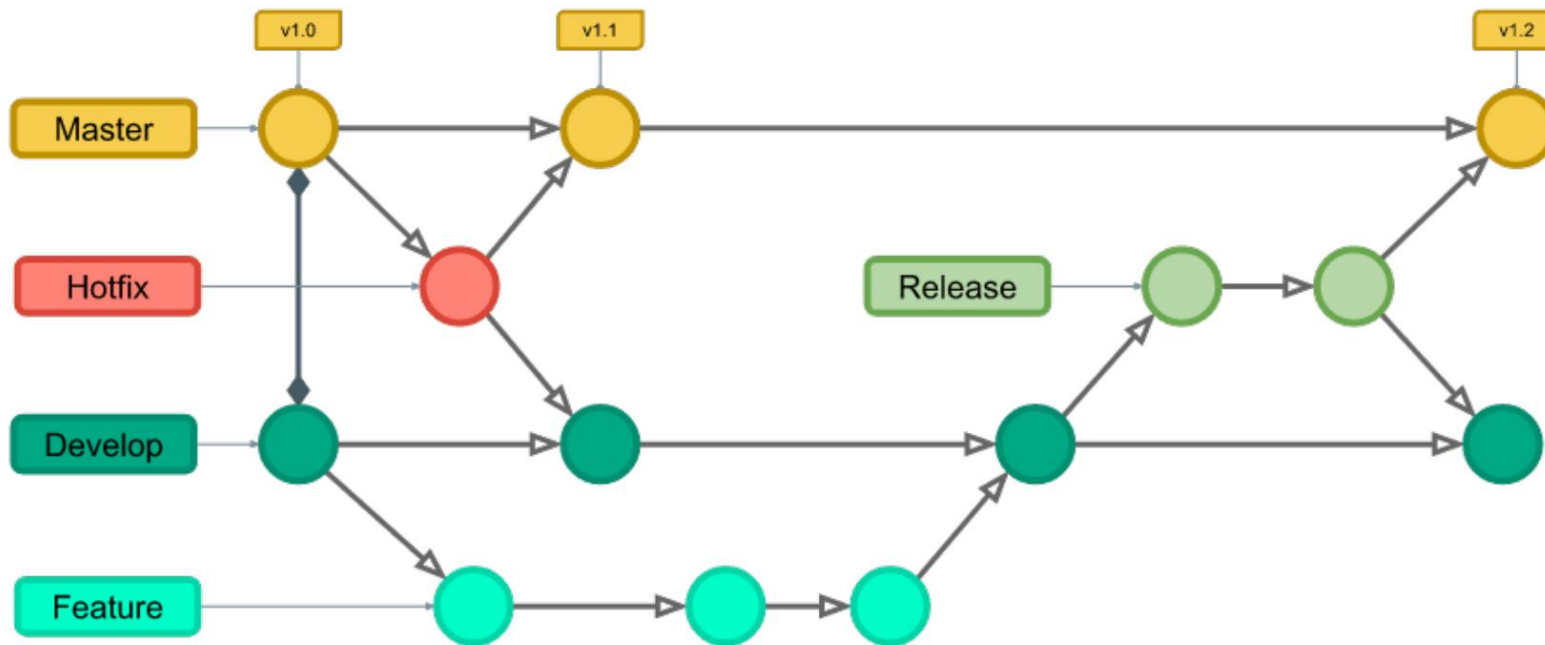
- Завершаем слияние:

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

- Если передумали выполнять слияние: `git merge --abort`

Gitflow

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



- Подробнее про:
- гит флоу
- код ревью
- хорошие практики при работе в команде
- ребэйз

Gitflow

- Ветка master - хранит историю релизов
- Ветка develop - для объединения всех функций.
- Под каждую фичу создается 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

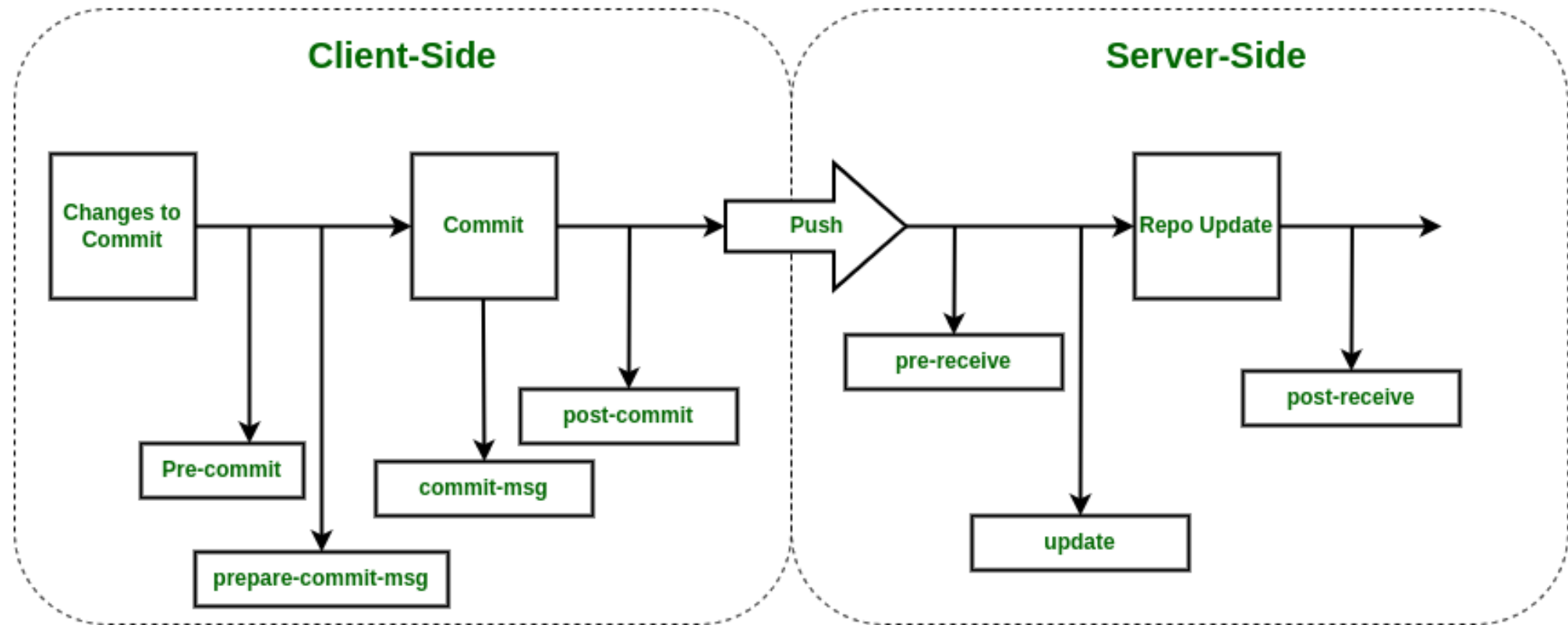
А что по гитфлоу для дс-ов?

- Тут сложнее
- Чаще всего, разные эксперименты – разные ветки одного репозитория, выделенного для задачи
- Из общего и нормального – код обучения, вычисления метрик, работы с данными и т.п., что и хранится в мастере/мэйне; как вариант – обернуть в пакет. Здесь классический флоу
- Но есть проблема – много экспериментов, много веток. Соответственно, ветки мы либо вливаем в мастер, либо удаляем.
- Еще вариант – хранить эксперименты в отдельной репе по папочкам

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, которые можно посмотреть в `.git/hooks`
- Писать Git hooks можно на любом скриптовом языке: Python, PHP, Bash. Файл нужно сделать исполняемым:
 - `chmod a+x .git/hooks/pre-commit`

Схема хуков



ГОТОВЫЙ ИНСТРУМЕНТ: pre-commit

```
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
```

Устанавливаем pre-commit для репозитория:

pre-commit install

Проверить все файлы:

pre-commit run --all-files

Артефакты

Что это

- В процессе работы над проектом возникают артефакты - сборки, релизы, пакеты, отчеты, docker images и так далее.
- Обычно, эти артефакты хранят в таких местах, как:
 - Artefactory (jFrog Artefactory, Nexus)
 - Object storage (s3)
 - Git (Git LFS)
- Для некоторых артефактов есть специальные хранилища:
 - PyPi - Python package index
 - Docker registry (Docker hub, private docker registry)
 - Gitlab Package Registry, Github packages, Gemfury

Nexus

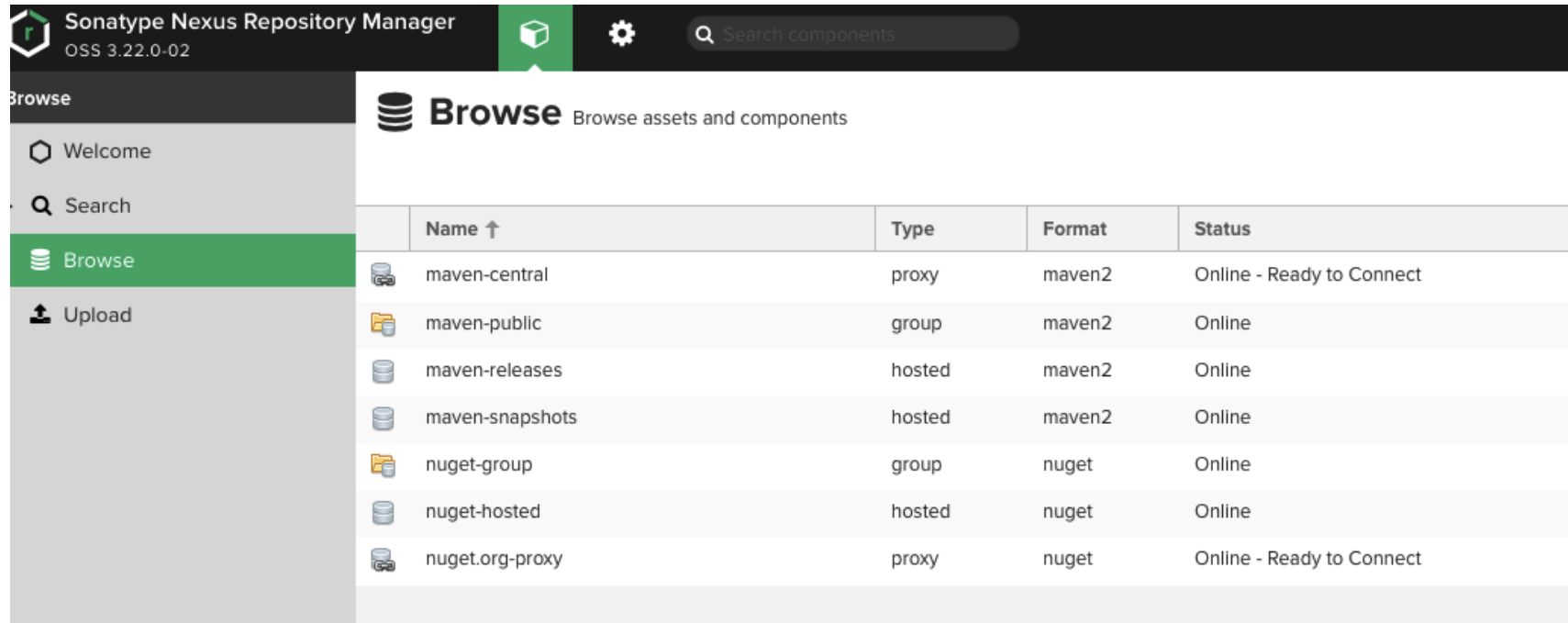
- Одно из распространенных хранилищ артефактов – Nexus. В качестве альтернативы есть jFrog.
- Если есть бабки, jFrog лучше, но бесплатная версия Nexus весьма обширна, нередко хватает ее, и она лучше чем фришная версия jFrog.
- По факту, Nexus не сколько хранилище артефактов, сколько **менеджер репозиториев**








Nexus

- Nexus по факту является единой точкой управления и общения кучи репозиторий-хранилищ
- Так, через него можно удобно и просто управлять docker образами, сборками кода, своим PyPI, да в принципе чем угодно
- Само хранение артефактов настраивается и может происходить где угодно, например, в s3
- Место, где хранятся артефакты, в терминах Nexus – **blob**. Настроить в качестве блоба можно что угодно.

Nexus: как взаимодействовать

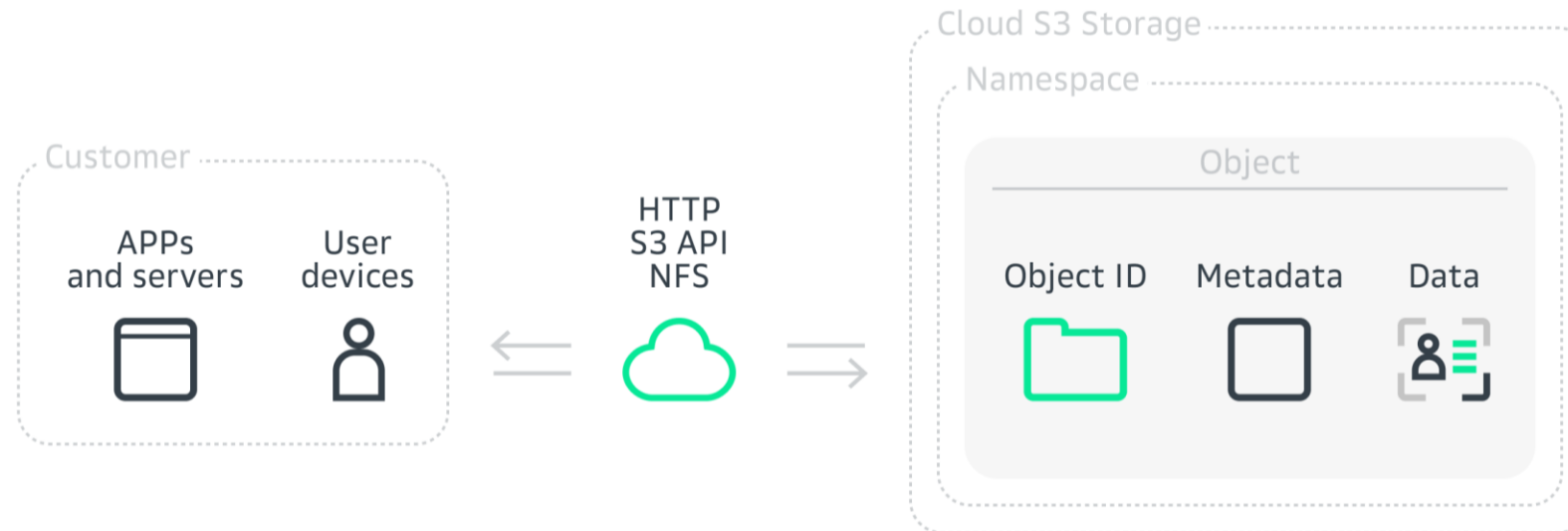
- Есть удобный UI, через который можно всем этим управлять
- Из кода/терминала управлять очень просто через API, то есть простыми запросами
- То есть это curl в терминале и python-nexus в питоне



	Name ↑	Type	Format	Status
	maven-central	proxy	maven2	Online - Ready to Connect
	maven-public	group	maven2	Online
	maven-releases	hosted	maven2	Online
	maven-snapshots	hosted	maven2	Online
	nuget-group	group	nuget	Online
	nuget-hosted	hosted	nuget	Online
	nuget.org-proxy	proxy	nuget	Online - Ready to Connect

S3

- S3 – объектное хранилище, обычно на базе какого облака
- Пошло (вроде как) от AWS
- Общение происходит через S3 API
- По факту, просто удобная объектная хранилка в облаке
- Локально поднять можно с помощью Minio



Особенности S3

- Объектное ☺
- Плоское адресное пространство, есть только бакеты
- Бакет - это логическая сущность, которая помогает организовать хранение объектов.
- В облаке куча еще особенностей:
 - Георепликация
 - Масштабирование
 - Некоторые возможности вроде группировки и сортировки на стороне облака по простому запросу
- Быстрый поиск
- Можно настроить версионирование

Как общаться

- В Python общаться с s3 можно с помощью библиотеки **boto3**

```
import boto3
session = boto3.session.Session()
s3 = session.client(
    service_name='s3',
    endpoint_url='https://<endpoint>'
)

# Создать новую корзину
s3.create_bucket(Bucket='bucket-name')

# Загрузить объекты в корзину из строки
s3.put_object(Bucket='bucket-name', Key='object_name', Body='TEST')

# Загрузить объекты в корзину из файла
s3.upload_file('this_script.py', 'bucket-name', 'py_script.py')
s3.upload_file('this_script.py', 'bucket-name', 'script/py_script.py')

# Получить список объектов в корзине
for key in s3.list_objects(Bucket='bucket-name')['Contents']:
    print(key['Key'])
```

Как общаться

- Есть еще всякие специальные программки для управления S3 вроде **S3 Browser** и т.п.

Minio – бесплатный локальный S3

- The short and simplified answer is “It’s like Amazon S3, but hosted locally.”
- Удобно развернуть, например, для себя либо на команду
- Есть большинство фишек как и у AWS
- Работает по дефолту над файловой системой, поэтому если будет куча объектов – важно выбрать правильную файловую систему
- Есть некоторые проблемки при большом количестве объектов (миллионы)

Создание питоновских пакетов

- Есть два ~~стуса~~ способа:
 - Через poetry
 - Через setup.py

Setup.py

```
1 from distutils.core import setup
2 from setuptools import find_packages
3
4 setup(
5     name='NeteasyDownloader',
6     version='0.01',
7     keywords=('neteasy', 'music', 'downloader', 'download'),
8     description='Neteasy Music/Ablum/Playlist downloader.',
9     long_description='neteasy -m 29414460 [-m music, -a ablum, -p playlist]',
10    author='MyFaith',
11    author_email='faith0725@outlook.com',
12    url='https://github.com/MyFaith/neteasy-downloader',
13    license='MIT',
14    packages=find_packages(),
15    entry_points={
16        'console_scripts': [
17            'neteasy = NeteasyDownload.main:main'
18        ]
19    }
20 )
```

python setup.py bdist_wheel

Poetry

- poetry build 😊
- Пакет собирается по вашему томлику