1531.23
Рейтинг

Timeweb Cloud
То самое облако

Подписаться

kesn
21 сен 2024 в 11:05

Программирование — это вообще не просто!

19 мин 52K

Блог компании Timeweb Cloud, Python*, Веб-разработка*, Карьера в IT-индустрии, Программирование*

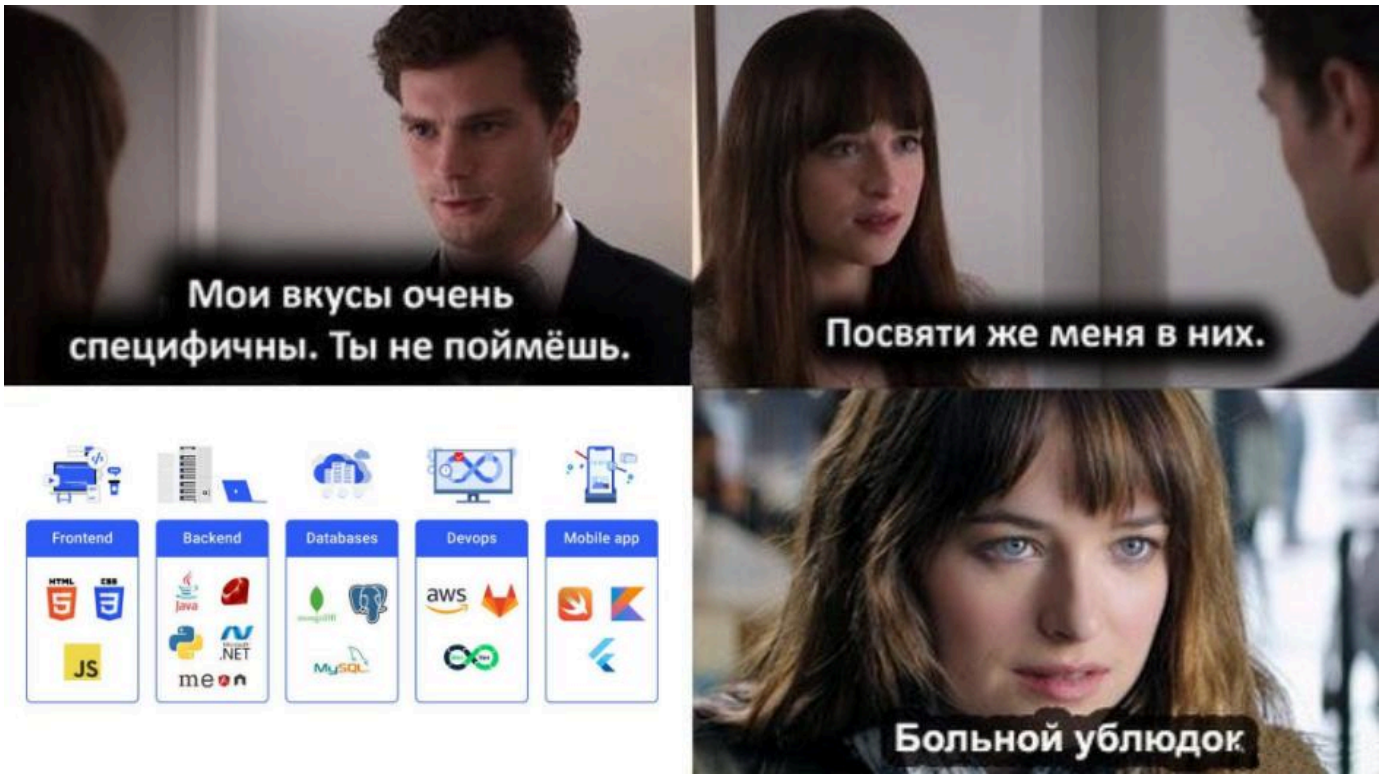
Привет, Хабр!

Идея статьи появилась, когда я начал повсюду замечать якобы подтверждения мифа, что «программирование — это просто».

В новостях «восьмилетняя девочка, которая второй раз в жизни занимается программированием, наклепала чат-бота за 45 минут» (ага, да!).

Курсы предлагают мне за 10 месяцев с нуля стать миддл+ (ага, да!).

Но я-то знаю, как оно на самом деле. Мы, программисты, обычно решаем проблемы и двигаемся дальше, но я решил запротоколировать всё как есть, и в течение пары месяцев скрупулёзно записывал всю ту хрень, что происходила со мной и моими коллегами, чтобы показать программирование без прикрас. Поехали!

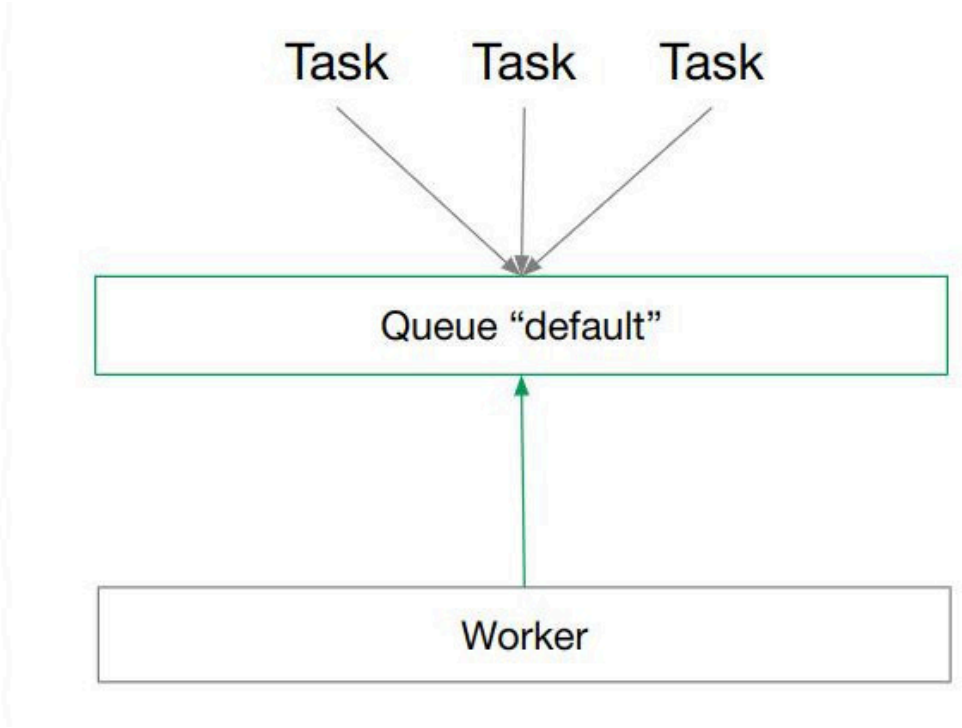


› Миф 1: Это программирование, поэтому можно немного ошибиться, и ничего за это не будет

Блин, да даже если ошибиться в одном символе — всё может сломаться! Вот вам примеры.

Celery

Celery — это такая библиотечка для запуска задач в питоне. Я её хотел использовать наибанальнейшим образом: складывать задачи в очередь «default» (крутое название, да?), и потом специальный воркер их бы оттуда забирал и выполнял одну за другой. Это самый простой сценарий, который только можно придумать.

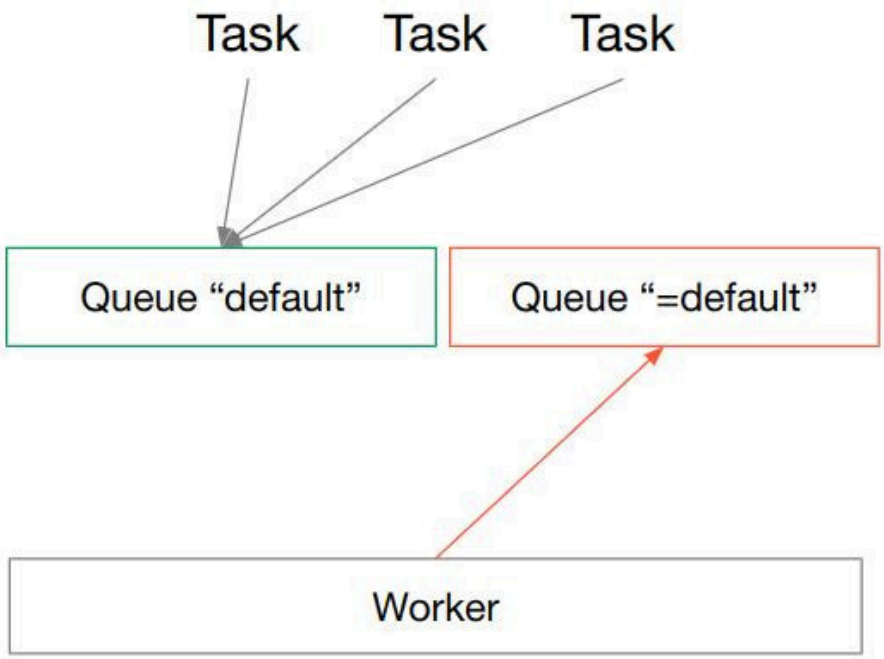


Запустить воркер можно очень просто, вуаля:

```
celery -A project worker -Q=default --loglevel=info
```

Запускаю — не работает! Задачи в очереди накапливаются, а воркер чиллит в сторонке и ничего не делает.

Оказывается, воркер слушал очередь, которая называлась « `=default` »! Ведь так я и написал: `-Q=default` .



Я решил разобраться, какого чёрта параметр `-Q` так странно обрабатывается — я такое вижу впервые. Вот симуляция воркера на питоне:

```
from argparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument('--queues', '-Q')
args = parser.parse_args()

print(args.queues)
```

Я потестировал вручную разные комбинации, в т.ч. ту, на которой у меня всё сломалось — но так и не смог воспроизвести баг:

```
> test.py --queues value
value
> test.py --queues=value
value
> test.py -Q value
value
> test.py -Qvalue
value
> test.py -Q=value
value  # это мой случай, но значение нормальное, без "равно"
> test.py -Q=”=value”
=value
> test.py -Q"=value"
value
```

Я почувствовал нотку безумия и полез в исходники celery. Выяснилось, что у них используется не встроенный в питон `argparse.ArgumentParser`, а библиотека click. Хорошо, перепишем нашу программу на `click` и потестируем...

```
import click

@click.command()
@click.option('--queues', '-Q')
def echo(queues):
    print(queues)
```

```
echo()
```

```
> test.py --queues value
value
> test.py --queues=value
value
> test.py -Q value
value
> test.py -Qvalue
value
> test.py -Q=value
=value  # хоба!
> test.py -Q=”=value”
==value  # хоба!
> test.py -Q"=value"
=value  # хоба!
```

Вот так я узнал, что `click` и `argparse` работают по-разному, но чтобы это узнать, нужно поставить куда-нибудь знак «=». Люблю программирование!

Django settings

Написал приложение на Django и больше 100 тестов на все случаи жизни. Запускаю — работает, но как-то странно, с неправильными значениями. Никаких ошибок в логах, конечно, нет. Проверяю код несколько раз и нахожу это:

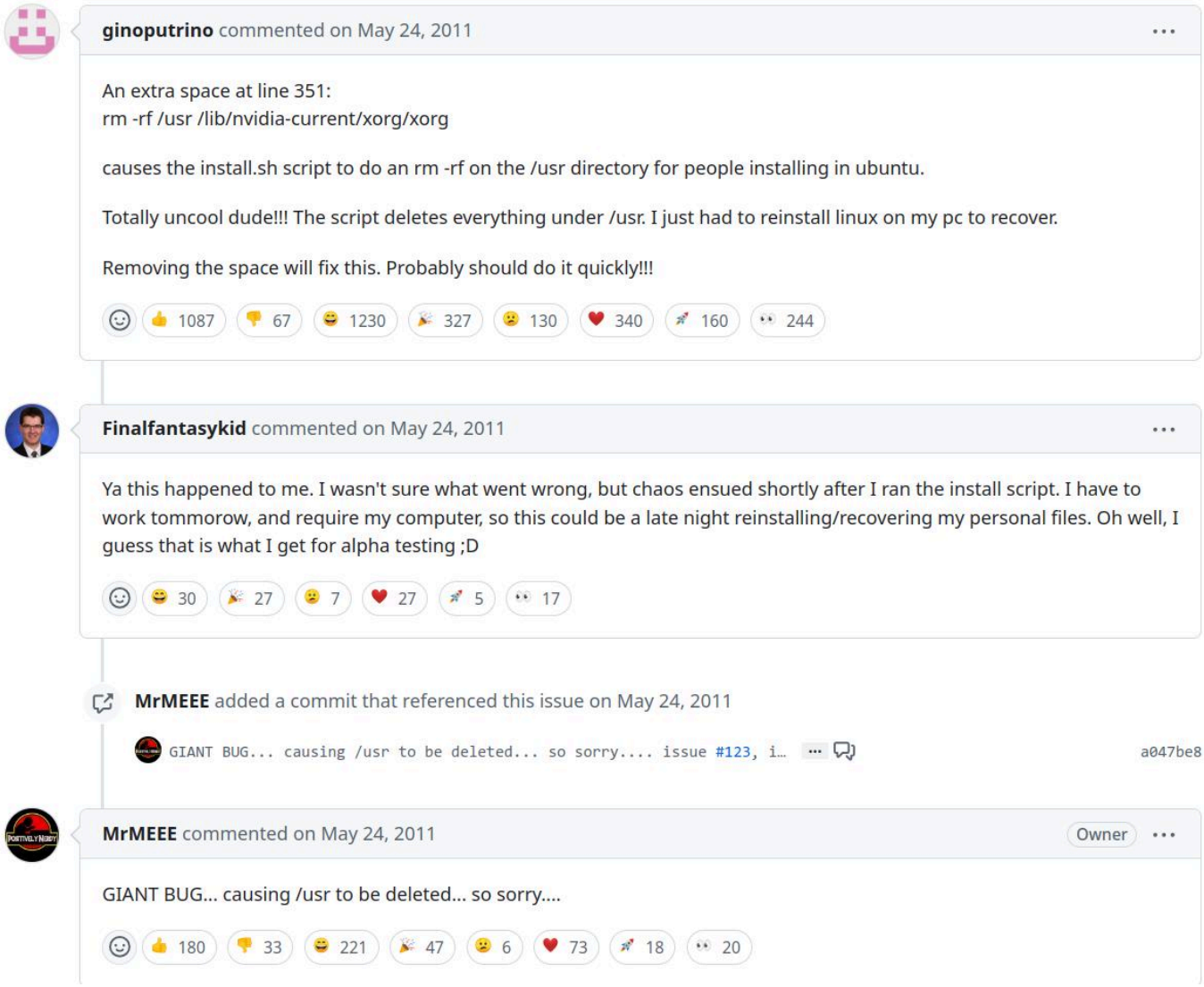
```
# settings.py
SUBSCRIPTION = {
  'duration': timedelta(days=30),
  'price': Decimal(100),
}

# service.py
subscriptions = getattr(
  settings,
  'SUBSCRIPTIONS',
  DEFAULT_SETTINGS
)
```

Так уж устроен Django: настройки — это питон-файл с выражениями типа `<КЛЮЧ> = <значение>`. Так как это просто файл, а не какая-то хитрая структура данных, то ключи могут быть любыми. В данном случае я указал `SUBSCRIPTION = ...`, а нужно было `SUBSCRIPTIONS = ...`. Получается, забыл одну букву «S» — и все мои настройки идут коту под хвост и вместо них используются настройки по умолчанию! Ура!

Bumblebee

Это не моё, но одно из моих любимых. Как-то один разработчик в установочном скрипте случайно поставил пробел посреди строки, и скрипт сносил папку `/usr` на компах пользователей, делая систему нерабочей.

Мелочь, а приятно!

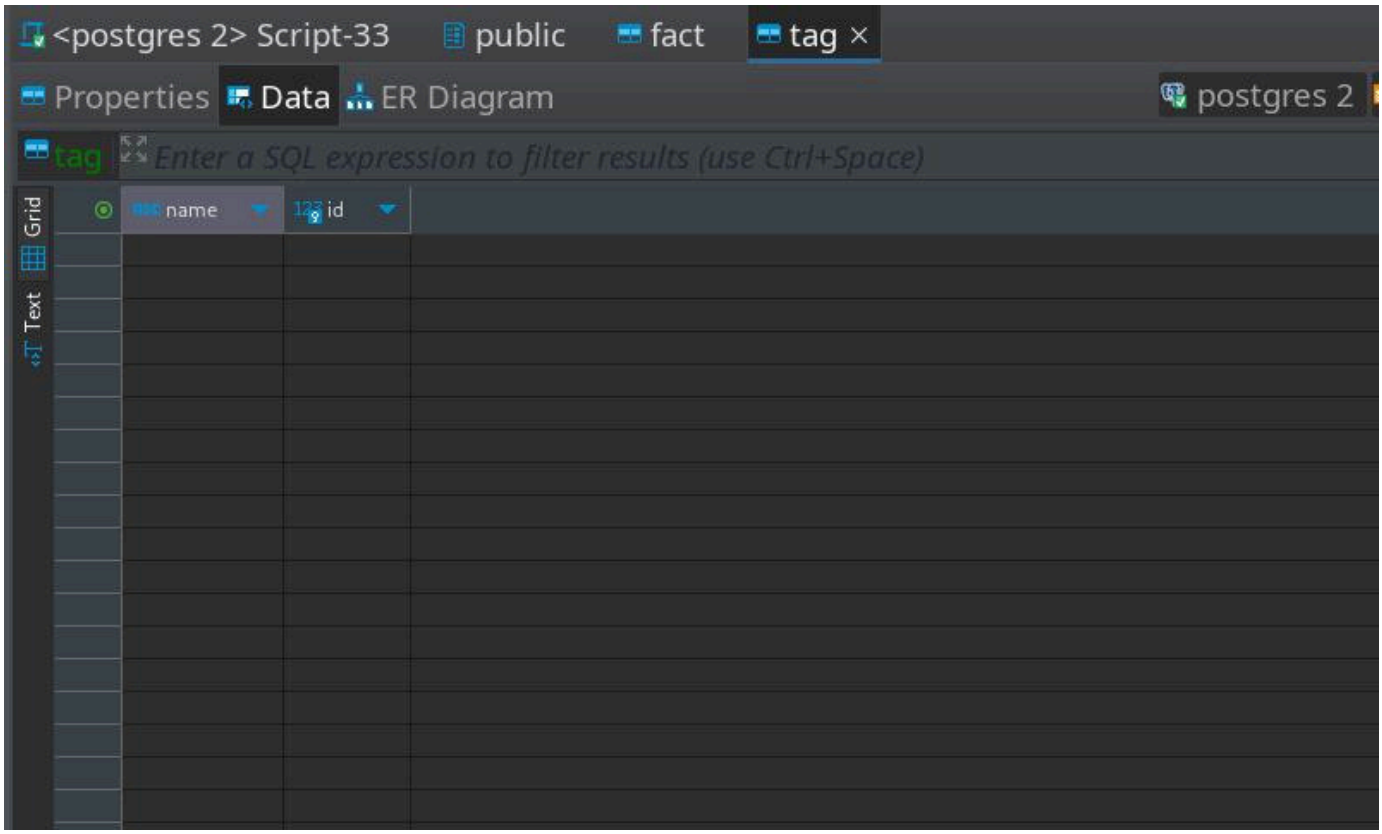


SqlAlchemy

Пытаюсь как-то записать несколько строк в Postgres. SqlAlchemy ругается на меня:

```
(psycopg.errors.UniqueViolation) duplicate key value violates unique constraint "ix_tag
DETAIL:  Key (lower('name'::text))=(name) already exists.
[SQL: INSERT INTO tag (name, id) VALUES (%(name)s::VARCHAR, nextval('tag_id_seq')) RETU
[parameters: {'name': 'ai'}]
```

В переводе на русский это значит «чел, ты пытаешься записать одно и то же имя, а сам сказал, что имена должны быть разными». Проблема в том, что я пытаюсь записать **разные** имена. Хм, наверно, в базе уже есть какие-то записи, и я пытаюсь добавить дубликат. Иду в базу...



… а она пустая!

После некоторого расследования я нашёл это:

```python
class Tag(BigIntBase):
    name: Mapped[str]

    __table_args__ = (
        Index("ix_tag_name", func.lower('name'), unique=True),
    )
```

Я хотел, чтобы индекс (и уникальность) брались из имени в нижнем регистре (`func.lower('name')`), вот только это не джанго, а алхимия, и строка «name» здесь — это просто строка, а не значение из колонки «name». Поэтому БД для каждой новой строки честно брала строку `'name'` , переводила её в нижний регистр и говорила, что я такую уже пытался записать только что!

Мини-рефакторинг

Жил-был вот такой код. Тут ничего сложного — есть «эпоха» (интервал значений), и нужно отфильтровать объекты, которые из этой эпохи.

```
epoch = (100, 200)
jobs = Jobs.objects.filter(
    block__gte=epoch[0],
    block__lt=epoch[1],
)
```

Расслабьтесь — тут меня ещё не было, так что всё работает. Но вот я посмотрел на это и подумал: не по-сеньорски! Если подумать, то эпоха — это диапазон чисел, а для диапазонов в питоне есть встроенный тип — `range` . Его можно в type hints писать, и `epoch: range` явно лучше, чем `epoch: tuple[int, int]` . Поэтому мини-рефакторинг:

```
epoch = range(100, 201)
jobs = Jobs.objects.filter(
    block__gte=epoch[0],
    block__lt=epoch[1],
)
```

Я даже не ошибся в границах (ведь чтобы число 200 было включено в интервал, нужно передать в range 201). Но всё равно все тесты поломались. Почему? А потому что `epoch[0]` всё так же равнялось `100` , но вот `epoch[1]` стало значить не «конец интервала», а «второй элемент», то есть `101` . В итоге мой фильтр скукожился до `[100, 101)` :) А как правильно? А вот так: `epoch.start` и `epoch.stop` .

```
epoch = range(100, 201)
jobs = Jobs.objects.filter(
    block__gte=epoch.start,
    block__lt=epoch.stop,
)
```

Флаги bash

Этот случай утащил с работы. Простенький скрипт:

```
#!/bin/bash -e

false
rm -rf ~
```

В первой строчке мы говорим, что для его запуска нужно использовать `bash` с флагом `-e`. Команда `false` возвращает код ошибки, а так как у нас `bash -e`, то выполнение скрипта прерывается, и до `rm -rf ~` не доходит. Надёжно? Надёжно.

Но только пока мы запускаем скрипт вот так: `./script.sh`. Однажды приходит другой разработчик и запускает скрипт через `bash script.sh`. Башу плевать на все эти `#!...` в начале файла, он это считает комментарием и пропускает. Поэтому он выполняет команду `false`, она возвращает ошибку, но по умолчанию (чья это была идея, блин?) баш игнорирует ошибки и выполняет команды дальше — а дальше `rm -rf ~`. Удобно? Удобно.

Решение — ставить флаги так, чтобы их нельзя было проигнорировать:

```
#!/bin/bash

set -e
false
rm -rf ~
```

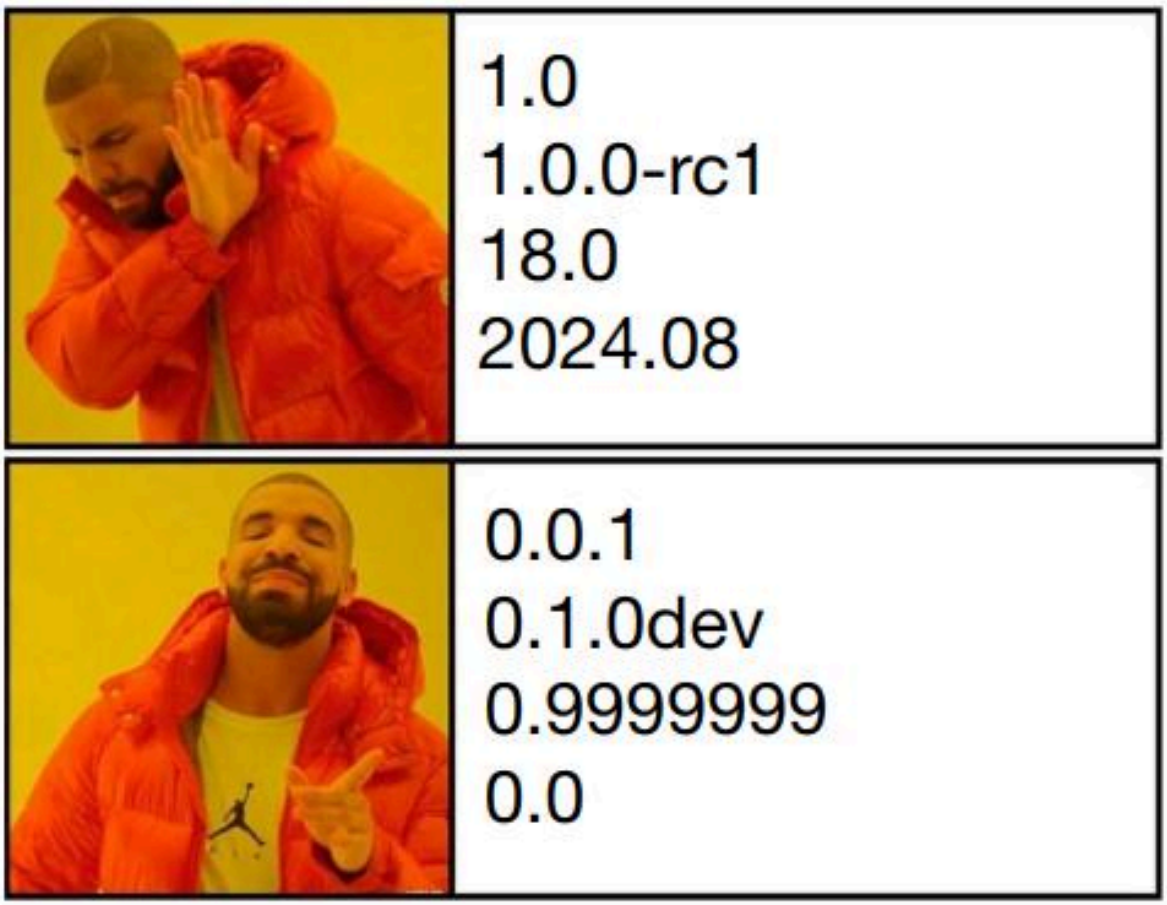
› Миф 2: У нас есть версии и системы контроля версий, так что ничего не ломается и всё отслеживается

ZeroVer

Да, у нас есть SemVer, которое позволяет отделить простые патчи от изменений, ломающих совместимость. У нас есть Changelogs. У нас есть LTS релизы, наконец.

Но всё равно находятся те, кто любит идти перпендикулярно best practices и добавляют перчинку в программирование. Вы же слышали про ZeroVer? Это когда разработчик

говорит «чуваки, я не несу никакой ответственности за свою программу и могу её сломать хоть прям сейчас!».

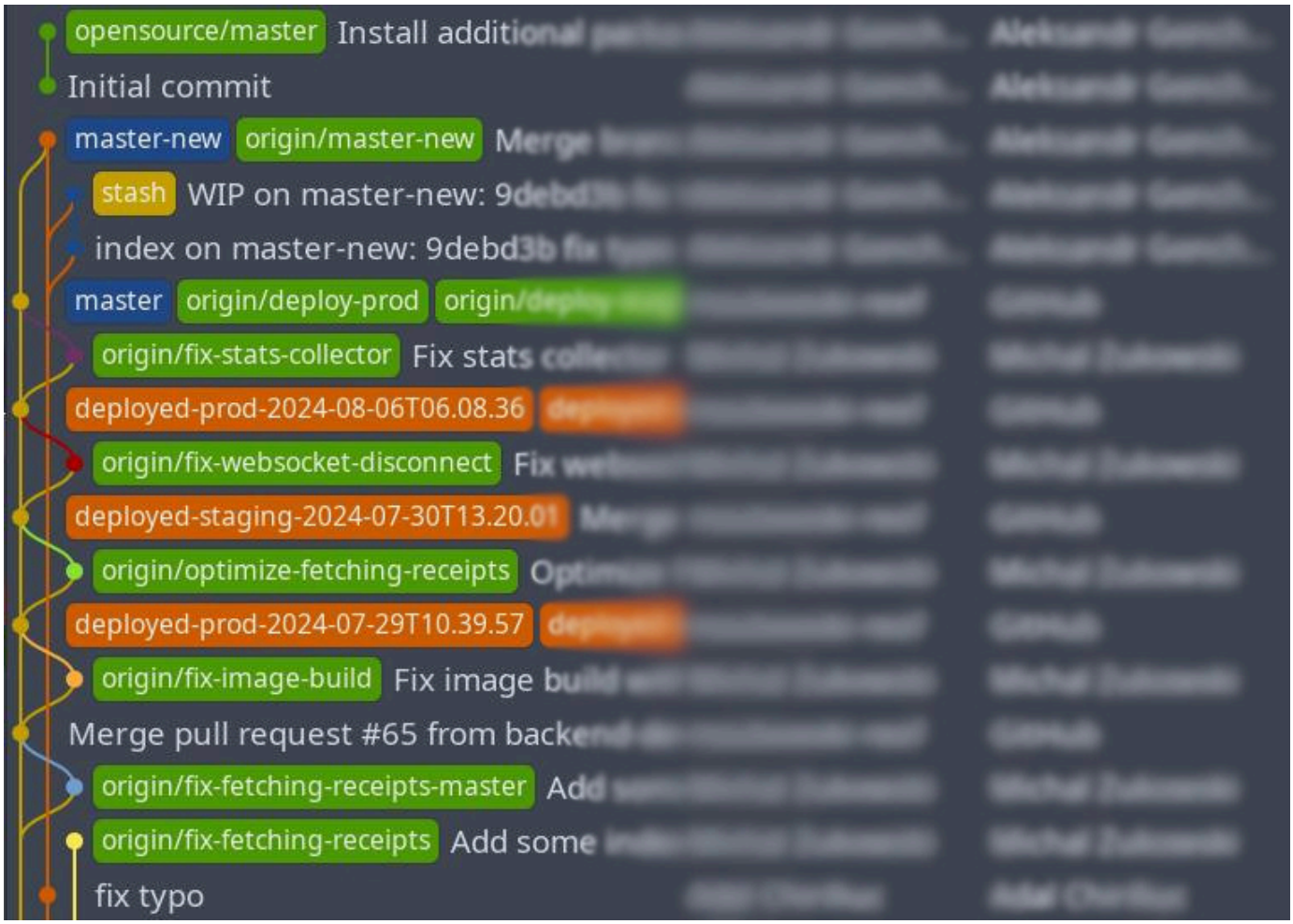


Самые популярные ZeroVer библиотечки — это, например, FastApi (на данный момент v0.115.0), react native (v0.75.3), за полным списком можно зайти на сайт https://0ver.org. Некоторые софтины живут в нулевой версии по 10+ лет!

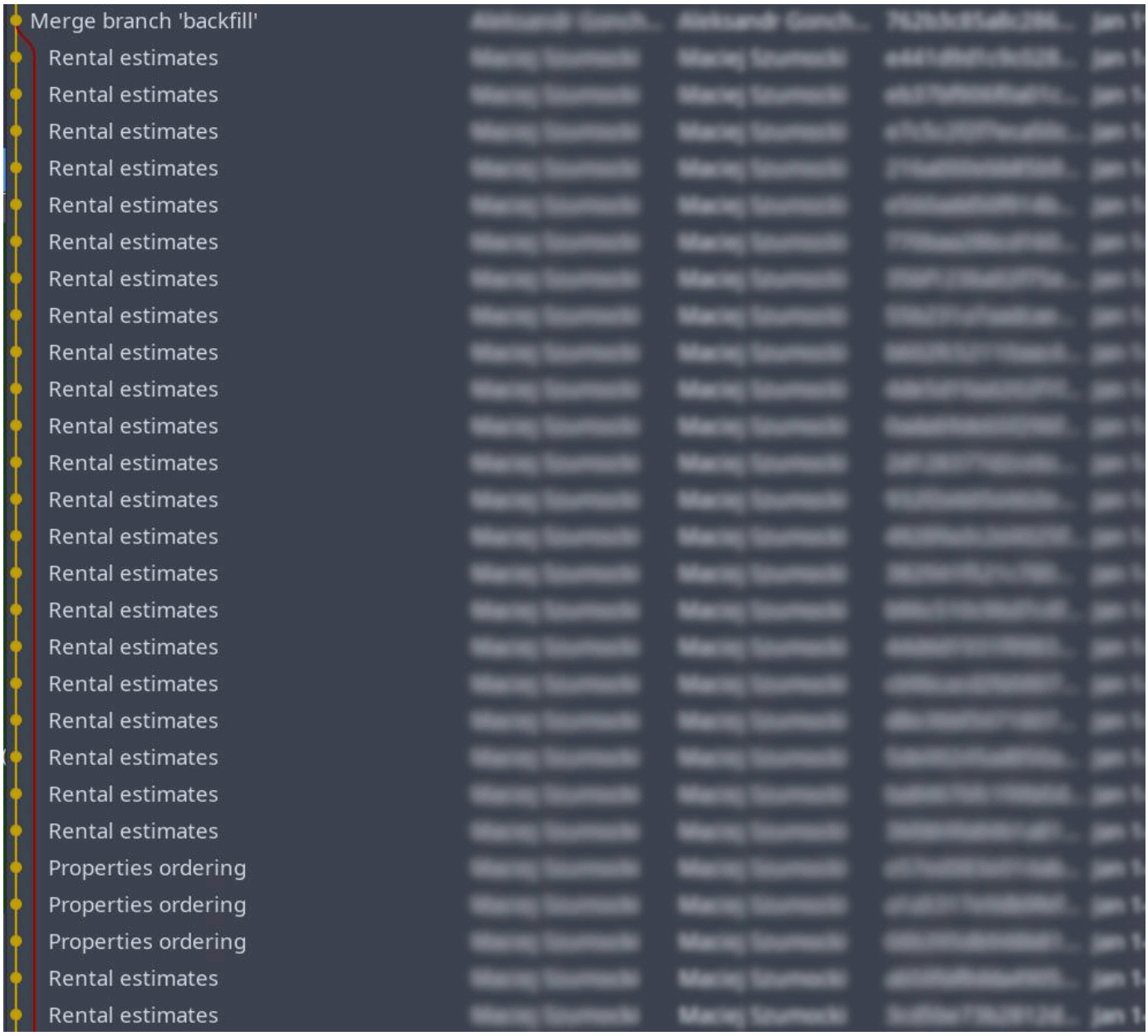
Git

У нас есть git, мы можем откатываться, работать в ветках и вообще наглядно видеть всю историю, да?

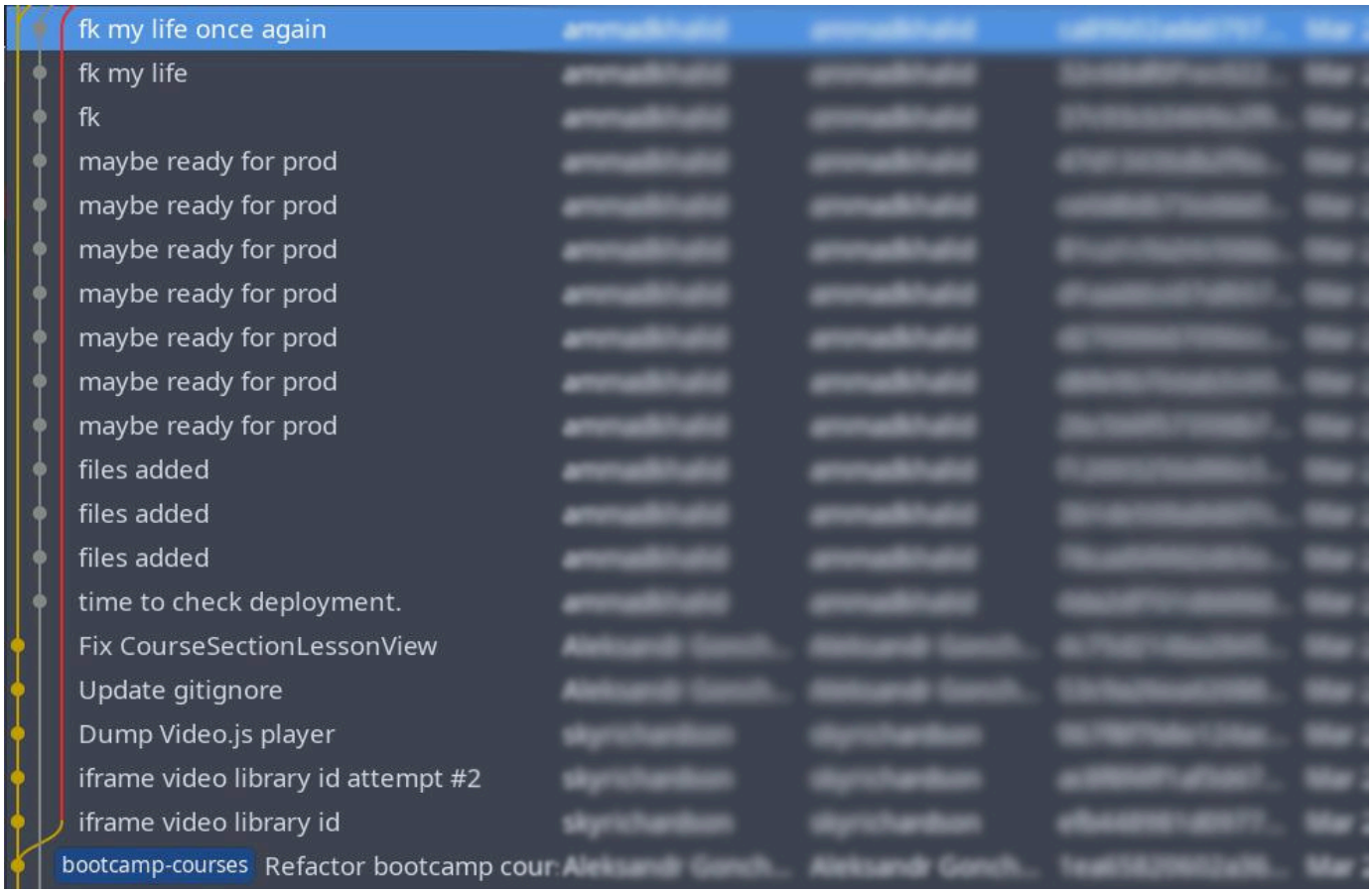
Ну вот вам наглядная история, пожалуйста. Тут какие-то фиксы, постоянные мерджи, есть ещё stash, а в конце вообще начинается отдельная история, никак не связанная с основной!



Или вот, например, коллега пишет понятные сообщения, чтобы я не запутался:



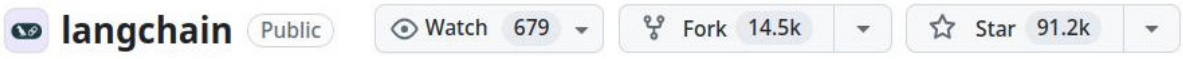
Иногда git хранит душераздирающие истории релизов, мне даже захотелось снять фильм по этому сценарию (читать снизу вверх):



› Миф 3: Это программирование, поэтому все умные и пишут классный код

Langchain: sync vs async

Langchain — это фреймворк для создания приложений с использованием LLM. 14k форков и 91к звёзд на гитхабе говорят сами за себя.



Кто я такой, чтобы судить о проектах такого масштаба, да? Вот только я обнаружил, что этот замечательный фреймворк блокировал event loop, вызывая синхронную функцию вместо асинхронной.

Вообще langchain хочет уметь и в синхронный, и в асинхронный питон, и поэтому использует популярную тактику «добавлю `async` / `await` туда и сюда, всё синхронное запущу в потоках, методам добавлю букву `a` вначале — и вот я уже асинхронный». Вот пример.

Синхронное:

```
class RunnableWithMessageHistory(RunnableBindingBase):
    def _exit_history(self, run: Run, config: RunnableConfig) -> None:
        hist: BaseChatMessageHistory = config["configurable"]["message_history"]

        inputs = load(run.inputs)
        input_messages = self._get_input_messages(inputs)
        if not self.history_messages_key:
            historic_messages = config["configurable"]["message_history"].messages
            input_messages = input_messages[len(historic_messages) :]

        output_val = load(run.outputs)
        output_messages = self._get_output_messages(output_val)
        hist.add_messages(input_messages + output_messages)
```

Меняем

- `def _exit -> async def _aexit`
- `hist.add_messages -> await hist.aadd_messages`

и дело в шляпе:

```
    async def _aexit_history(self, run: Run, config: RunnableConfig) -> None:
        hist: BaseChatMessageHistory = config["configurable"]["message_history"]

        inputs = load(run.inputs)
        input_messages = self._get_input_messages(inputs)
        if not self.history_messages_key:
            historic_messages = config["configurable"]["message_history"].messages
            input_messages = input_messages[len(historic_messages) :]

        output_val = load(run.outputs)
        output_messages = self._get_output_messages(output_val)
        await hist.aadd_messages(input_messages + output_messages)
```

Ну и конечно вы как только посмотрели на этот код, так сразу и скажете: Саня, ну это очевидно, тут в выражении `config["configurable"]["message_history"].messages` на самом деле `messages` — это не какой-то статичный атрибут со значением, а самый настоящий вычисляемый `@property`, который к тому же синхронный — мы это сразу поняли по имени!

Я не такой умный, у меня заняло некоторое время понять, в чём же проблема. Я завёл issue на github, его быстро пофиксили и побежали писать новые баги. Знаете, я даже подумываю написать статью про Langchain — он такой большой и в нём удивительно плохо всё, что я видел. А если вам хочется что-то подобное — про большущий проект, в котором куча дичи — то есть моя статья про Django.

Bittensor

Да, да, ещё один блокчейн. Но там крутятся деньги, и неплохие — вот картинка, которая должна сменить ваше лицо на менее скептическое:

Казалось бы, если крутятся деньги — то всё должно быть вылизано до блеска и покрыто тестами так, что кода не видно.

Конечно, ага!

Во-первых, они там не очень определились, как лучше: исключения или коды ошибок? Решили так: исключения и так сами будут вылетать из-за качества кода, так что для полноты картины можно ещё возвращать ошибки. Но коды и классы ошибок — это сложно, внешним разработчикам будет достаточно флага `is_success` и текста ошибки. Поэтому когда я вызываю их функцию `do_stuff()`, я как будто использую два презерватива — один проверяет и парсит ошибку, а другой ловит исключения:

```
try:
  is_success, msg = do_stuff()
  if not is_success:
    exc_class = parse_error(msg)
    raise exc_class(msg)
except Exception:
  # handle exception
```

И так на каждый вызов библиотечной функции. Удобно!

Но недавно нас на работе убило кое-что получше. «Разрабы», которые писали библиотеку, просто жить без неё не могут. Их библиотека — главная в мире, если она встречается у вас в программе, то ничего другого существовать не должно. Поэтому код инициализации такой:

```
class Subtensor:
    def __init__(
        self,
        network: Optional[str] = None,
        config: Optional[bittensor.config] = None,
        _mock: bool = False,
        log_verbose: bool = True,
    ) -> None:
        ...

        # Attempt to connect to chosen endpoint. Fallback to finney if local unavailabl
        try:
            # Set up params.
            self.substrate = SubstrateInterface(
                ss58_format=bittensor.__ss58_format__,
```

```
                use_remote_preset=True,
                url=self.chain_endpoint,
                type_registry=bittensor.__type_registry__,
            )
```

Как видно, сначала, разумеется, лезем в сеть! Какой смысл инициализировать хоть что-то без сети в 2024? Правильно, никакого!

Но что, если сеть недоступна? На это есть решение:

```
        except ConnectionRefusedError:
            _logger.error(
                f"Could not connect to {self.network} network with {self.chain_endpoint
            )
            _logger.info(
                "You can check if you have connectivity by running this command: nc -vz
                f"{self.chain_endpoint.split(':')[2]}"
            )
            exit(1)
```

Горит сарай — гори и хата! Не получилось инициализировать класс — убиваем всю программу. Ну а чо?

› Миф 4: Это программирование, поэтому всё просто

В конце концов, программа — это просто текст на (в большинстве случаев) псевдо-английском языке. Можно что-то понять, а что непонятно — можно почитать в интернете или спросить chatGPT. Ну да, ну да.

Обычный тест

Перед вами — обычный тест, я такие пачками вижу и пишу.

```
@patch("bittensor.subtensor", lambda *args, **kwargs: MockSubtensor())
@patch(
    "compute_horde_validator.validator.tasks.get_subtensor", lambda *args, **kwargs: Mo
)
@patch("compute_horde_validator.validator.tasks._run_synthetic_jobs", MagicMock())
@pytest.mark.django_db(databases=["default", "default_alias"])
def test__run_synthetic_jobs__debug_dont_stagger_validators__true(settings):
```

```
    settings.DEBUG_DONT_STAGGER_VALIDATORS = True
    settings.CELERY_TASK_ALWAYS_EAGER = True

    run_synthetic_jobs()
    from compute_horde_validator.validator.tasks import _run_synthetic_jobs
    assert _run_synthetic_jobs.apply_async.call_count == 1
```

Всего-то нужно знать: pytest, fixtures, манкипатчинг, моки и call count, лямбды, pytest-django и его фичи, celery и что такое eager mode, late imports, assert. Изи!

Bash

А вот обычный и очень простой скрипт на баше. Я когда такое вижу, начинаю немного понимать инквизиторов из средневековья.

```
set -euxo pipefail
SCRIPT_DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )"
```

Логика

Ну ладно, понятно, что есть сложный для чтения код. Но давайте что-нибудь простое! Вот, например:

```
await send_request(job)
start_time = now()

response = await ws.recv()
response_time = now()

reward = get_reward(response_time - start_time)
```

Тут вообще всё проще некуда: отправляем пользователю задачу (job), включаем таймер и замеряем время, пока он не пришлёт ответ. Ну и рассчитываем вознаграждение в зависимости от того, насколько быстро задачка была решена. Что не так?

Не так — это уверенность в том, что пользователь не считерит. А он может. Что, если он будет получать задание как обычно, но вот когда останутся последние байты (например, какие-нибудь закрывающик скобочки), он замедлит канал до 1 байта в несколько секунд? Тогда мы всё ещё будем ждать завершения передачи данных и не включать таймер, тогда

как пользователь уже получил всё то, что нужно для начала решения задачи, и успешно стартовал раньше таймера, выиграв себе дополнительное время. Каково, а? Код простой, ситуация страшная (c)

NPM

Скажите, если программирование такое простое, то почему когда я пытаюсь поставить `npx create-next-app`, мой многострадальный ноутбук несколько минут качает 360 пакетов? Что? Мало? Ну вот `npx create-remix` ставит 631 пакет. А `npx create-gatsby` — 864 пакета. А `npx create-react-app` — **1480**! Мне порой кажется, что npm — это какой-то аукцион, где важно победить по количеству зависимостей.

❯ Миф 5: Это программирование, поэтому всё логично

Бэкапы в postgres

Как мы делаем бэкап в postgres? Есть специальная утилита:

```
pg_dump postgres > dump.sql
```

Как восстановить? Есть специальная утилита...

```
pg_restore postgres < dump.sql
# pg_restore: error: input file does not appear to be a valid archive
```

Ну разумеется нет, для восстановления нужно не `pg_restore`, а просто

```
psql postgres < dump.sql
```

Почему? Потому что `pg_dump` по умолчанию выводит текст, а `pg_restore` ожидает бинарный формат. Всё логично.

Dockerfile build args

Иногда при сборке контейнеров нужно в `Dockerfile` передавать какие-то параметры. Ну, например, вы хотите «вшить» версию приложения прям в контейнер. Дело несложное: `docker build --build-arg VERSION=123`. Нюансы начинаются, когда у вас multi-stage build.

Тут понятно: аргумент передаётся в child image, так что base image про этот аргумент ничего не знает:

```
FROM alpine AS base
RUN echo "Base image: $VERSION"

FROM base
ARG VERSION
RUN echo "Child image: $VERSION"

#5 Base image:
#6 Child image: 123
```

Правила наследования вроде как работают: если передать аргумент в base image, то child image его унаследует:

```
FROM alpine AS base
ARG VERSION
RUN echo "Base image: $VERSION"

FROM base
RUN echo "Child image: $VERSION"

#5 Base image: 123
#6 Child image: 123
```

Но что случится, если аргумент объявить глобально? Ничего хорошего — ни base image, ни child image его не увидят!

```
ARG VERSION

FROM alpine:{$VERSION} AS base
RUN echo "Base image: $VERSION"

FROM base
RUN echo "Child image: $VERSION"

#5 Base image:
#6 Child image:
```

А почему? Ну вот так работает, это же даже в документации написано! Вы же читаете документацию от начала и до конца, прежде чем что-то использовать, да?

Javascript

Щас мне жсники влепят минусов за шутки в их адрес, но ведь с точки зрения питониста это правда забавно!

Во-первых, в js нельзя верить `==` :

```
> [1, 2, 3] == [1, 2, 3]
false
```

Вроде `===` тоже есть, но мне не помогло:

```
> [1, 2, 3] === [1, 2, 3]
false
```

Зато помогло приведение к строкам! Вот уж откуда помощи не ждал:

```
> JSON.stringify([1, 2, 3]) == JSON.stringify([1, 2, 3])
true
```

Погнали дальше, в сортировку:

```
> [-5, -2, 2, 1].sort()
[-2, -5, 1, 2]
```

Вот ещё удобный способ работы со строками:

```
> ('b' + 'a' + + 'a' + 'a').toLowerCase()
'banana'
```

Нулевая дата, конечно, вернёт дату *и время*, а так как `0` и `"0"` равны, то можно использовать любой вариант, и результат будет один и… что?!

```
> new Date(0)
Thu Jan 01 1970 03:00:00 GMT+0300 (Moscow Standard Time)

> 0 == '0'
true

new Date('0')
Sat Jan 01 2000 00:00:00 GMT+0300 (Moscow Standard Time)
```

Разделители тоже не имеют значе.....

```
> new Date('2024-01-01')
Mon Jan 01 2024 03:00:00 GMT+0300 (Moscow Standard Time)

> new Date('2024/01/01')
Mon Jan 01 2024 00:00:00 GMT+0300 (Moscow Standard Time)
```

Python

В питоне тоже всё просто, скобки всегда означают одно и то же...

```
{'a': 1}  # это словарь
{'a'}  # это множество (set)
{}  # это не пустое множество, это пустой словарь
set()  # а вот это пустое множество
```

```
1  # это просто число (int)
(1+2)  # это число
(1, 2)  # это кортеж (tuple)
(1)  # это тоже просто число
1,  # это тоже кортеж
()  # это пустой кортеж
(,)  # SyntaxError
```

```
[]  # это пустой список
[1]  # это список с одним элементом
[1, 2, 3]  # это список с 3 элементами
[i for i in range(3)]  # это тоже список с 3 элементами
()  # это пустой кортеж
(1)  # это просто число
(1, 2, 3)  # это кортеж
(i for i in range(3))  # это не кортеж, это generator expression
```

И поведение в питоне тоже очевидное. Например, он не даст изменять коллекцию в ходе итерации...

```
data = {1: 2}
for key, value in data.items():
  data[key+1] = value+1
# RuntimeError: dictionary changed size during iteration

data = [1, 2]
for value in data:
  data.append(value+1)
# Всё ок! Прощай, RAM :)
```

Pytest

Запускаю как-то в проекте pytest:

```
> ls
bot.py  config.py  history.py  knowledge.py  logs.py  __main__.py  models.py  __pycache
> pdm run pytest .
ERROR collecting src/tests/test_bot.py
ImportError while importing test module '~/workspace/project/src/tests/test_bot.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback: .pyenv/versions/3.11.4/lib/python3.11/importlib/__init__.py:126: in import_
    return _bootstrap._gcd_import(name[level:], package, level)
tests/test_bot.py:3: in <module>
    from ..bot import message_handler
E   ImportError: attempted relative import beyond top-level package
```

А он мне говорит знаменитое питоновское «у тебя относительные импорты, а top-level package не задан». Обычно это решается указанием «верхнего» модуля при помощи

`python -m top_level_module.script`. Но тут я запускаю `pytest` через `pdm`, всё не так просто… Я трачу время, пытаясь понять, какой ключ от меня хотят и в каком месте, и внезапно узнаю: чтобы pytest нашёл верхний модуль, нужно просто создать в нём файл `__init__.py`. Почему? Зачем?..

Dataclasses

Вот ещё пример логичности и простоты. Датаклассы `Base` и `Child`:

```
@dataclass
class Base:
  a: str

class Child(Base):
  b: str
```

Внезапно оказывается, что `Base` ведёт себя как приличный датакласс, а `Child` — как неприличный:

```
> Base()
TypeError: Base.__init__() missing 1 required positional argument: 'a'

> Base(a=’test’)
Base(a='test')

> Child(a=’test’)
Child(a='test')

> Child(a=’test’, b=’test’)
TypeError: Base.__init__() got an unexpected keyword argument 'b'
```

Почему так? Потому что «датаклассовость» не наследуется. Если хочется, чтобы `Child` тоже был датаклассом, его нужно тоже декорировать.

```
@dataclass
class Base:
  a: str

@dataclass
```

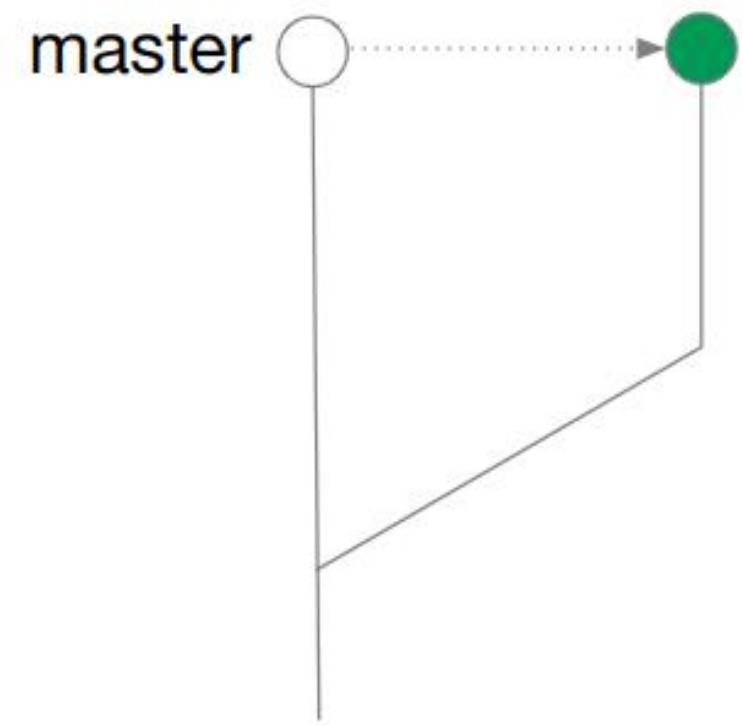
```
class Child(Base):
  b: str
```

Так что будьте осторожны при наследовании.

› Миф 6: Это программирование, поэтому никакой неведомой фигни не происходит

Github checkout action

Работаю в своей ветке. Делаю pull request, запускаются тесты и падают с ошибкой — какой-то конфликт файлов. На домашнем компе тесты, конечно, проходят без проблем. Смотрю логи — конфликтуют два файла, один в моей ветке есть, второй не существует и вообще непонятно откуда взялся при тестировании. Ищу этот файл повсюду, нахожу в ветке `master`. Как файл из `master` просочился в мою ветку?!

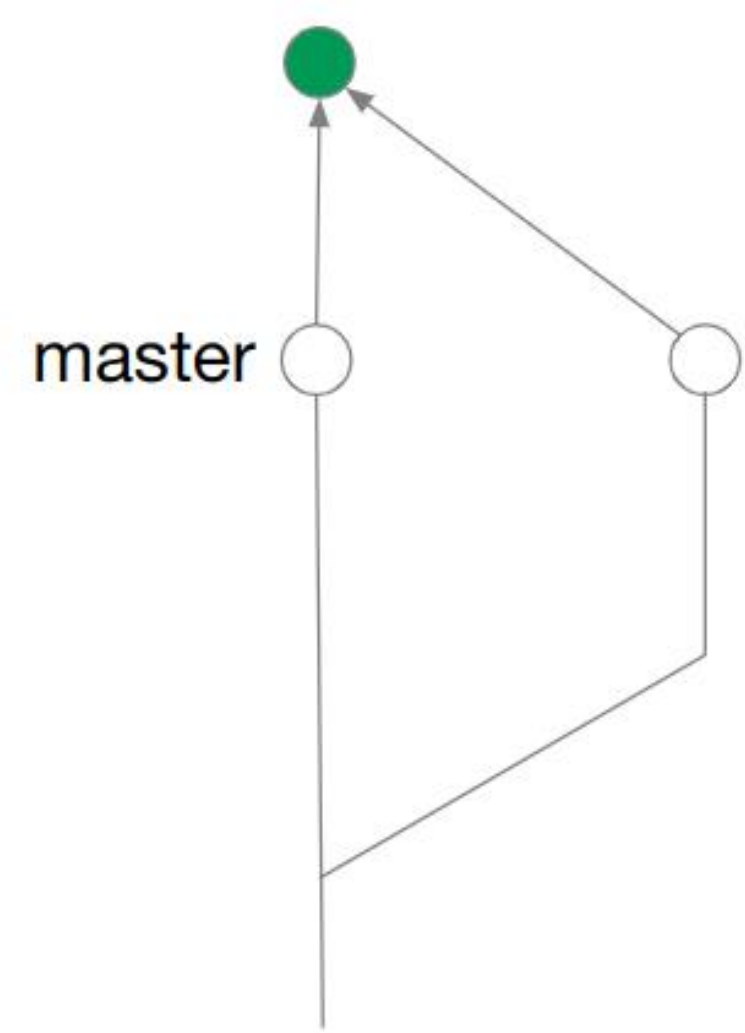


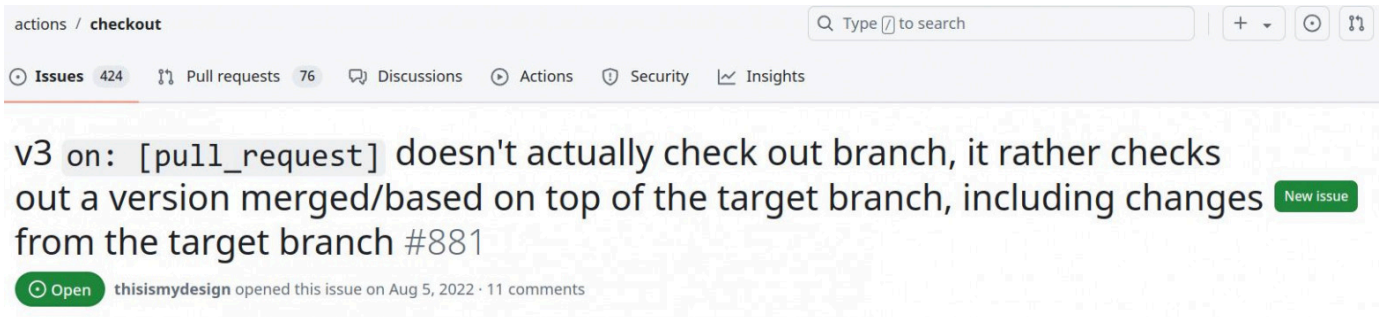
Опять чертовщина какая-то! Пошёл разбираться и узнал: оказывается, если вы пишете вот так...

```
# .github/ci.yml

test:
```

```
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0
```

… и у вас pull request, то этот экшен сначала делает merge с целевой веткой, а уже потом checkout результата.

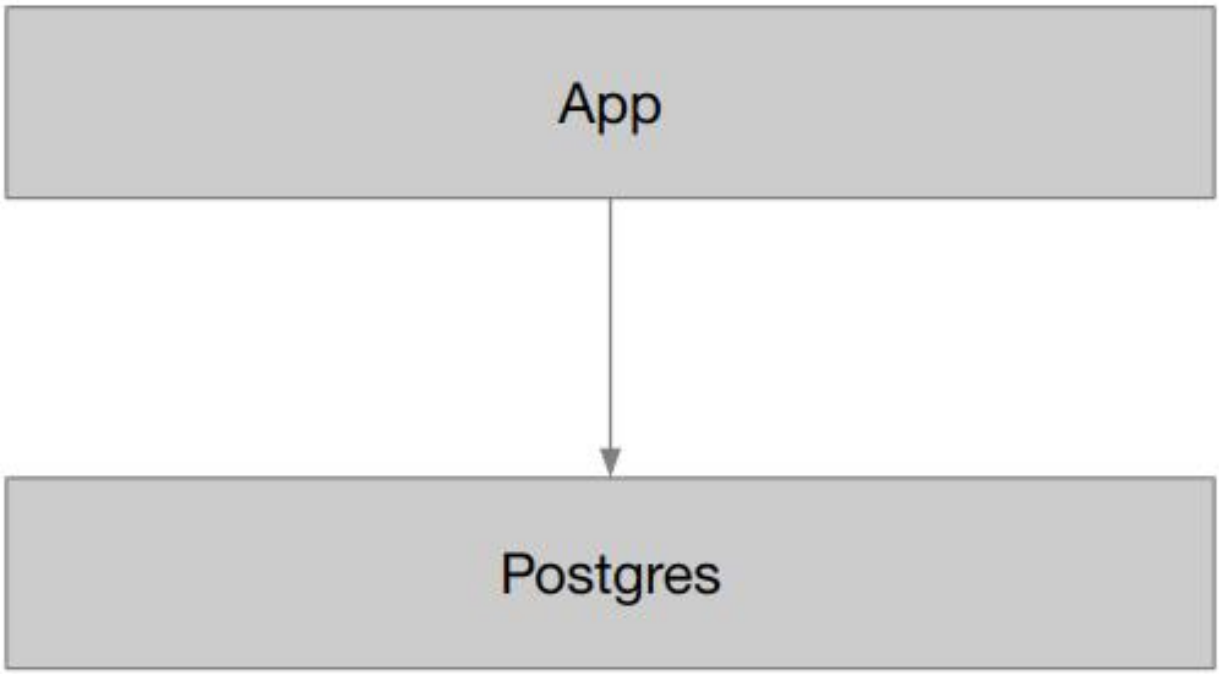




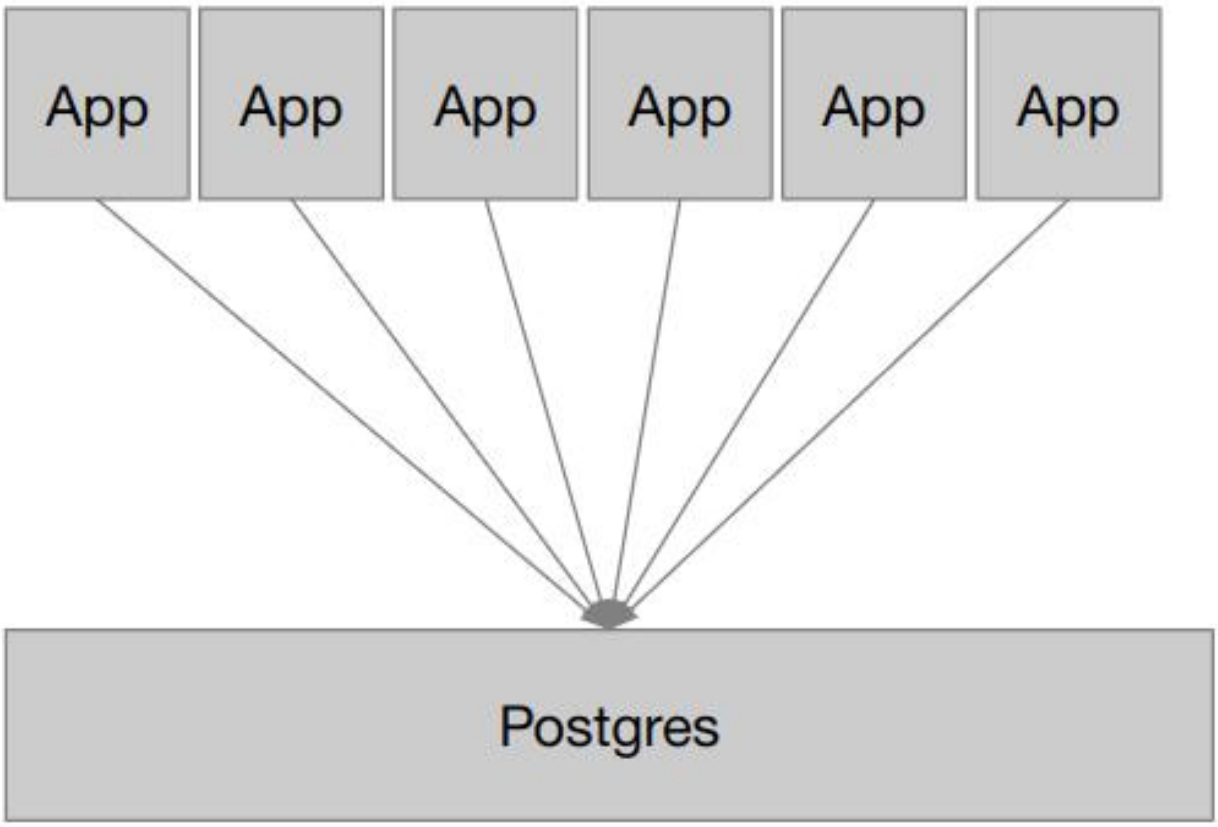
Представьте, я всю свою жизнь не знал об этом!

PgBouncer

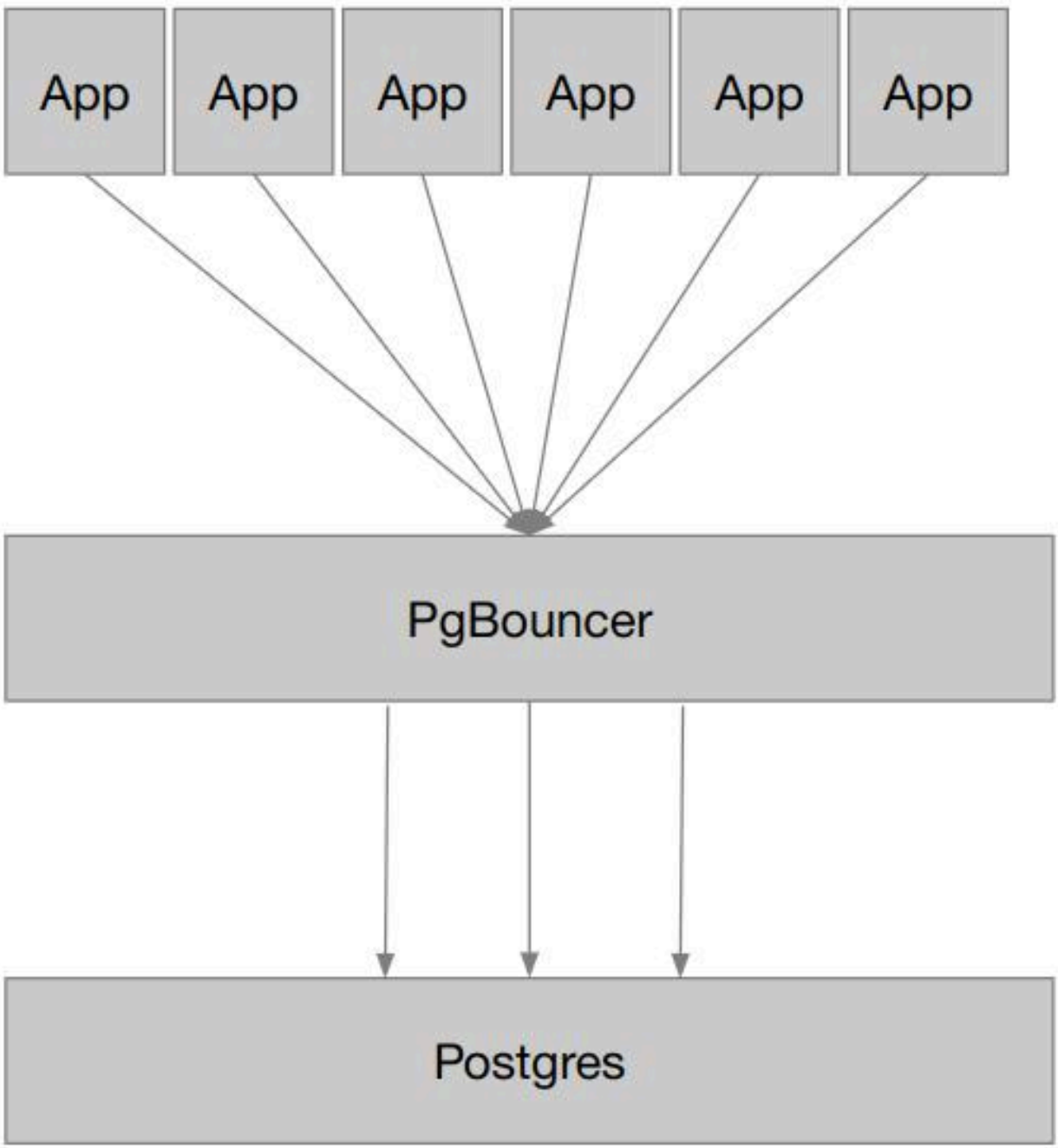
Самый простой способ подключить приложение к базе данных — это подключить приложение к базе данных! Ваш кэп.



Если приложений много, а база одна — то получится много соединений, базе данных такое может не понравиться.



К счастью, есть решение: PgBouncer. Он выступит в роли прослойки между вашими приложениями и БД, эффективно выделяя соединения для приложений:



Казалось бы, круто! Что может пойти не так?

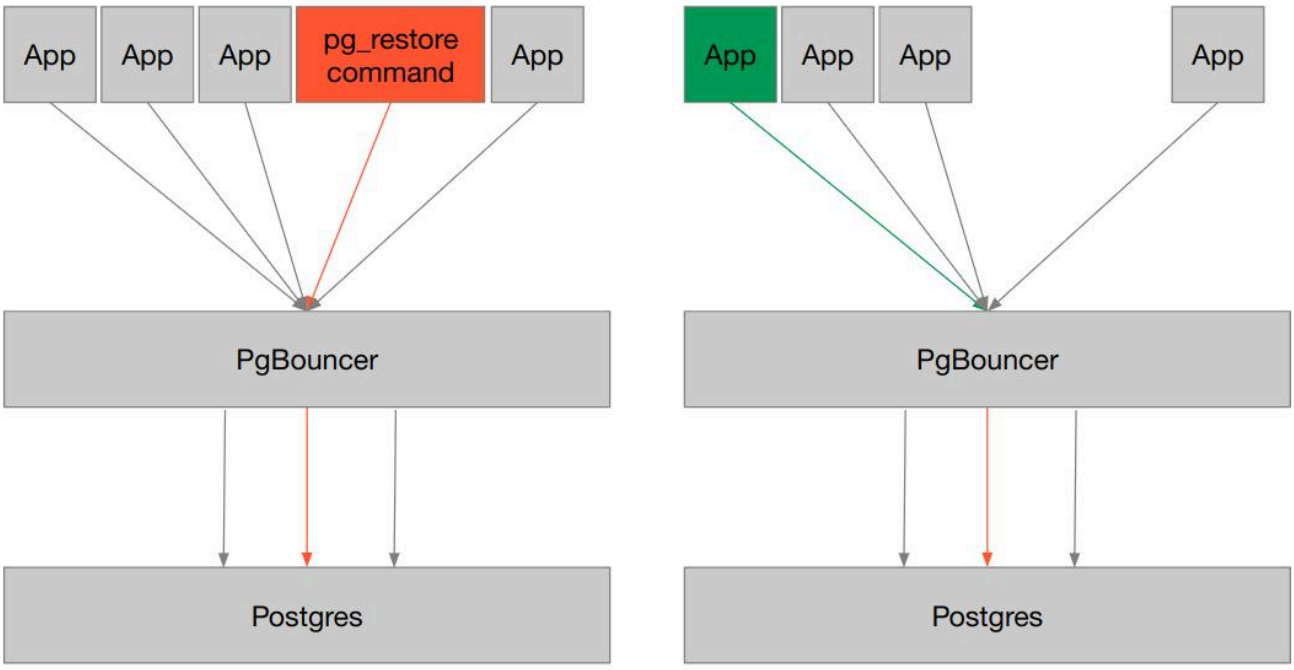
В один прекрасный день коллега (как же я счастлив, что не я!) по какой-то причине восстанавливает данные из бэкапа. Бэкап — это результат работы утилиты `pg_dump`, то есть простой текстовый файл типа

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
```

```
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
...
```

Команда отрабатывает, и приложение падает с ошибкой «таблицы не существуют». Я прям представляю, как это приятно — когда ты только что удалил данные, а восстановленные данные не находятся :D

Почему? А потому что строчка `SELECT pg_catalog.set_config('search_path', '', false);` из бэкапа как бы говорит: «postgres, забудь всё, что ты знал до этого, щас будем жёстко восстанавливаться из бэкапа». Postgres всё забывает, но т.к. мы работаем через PgBouncer, то после восстановления данных соединение с БД не исчезает в никуда, а как есть передаётся другому приложению. Да, прям в состоянии «забудь всё».



Приложение пытается что-то делать через это соединение, а оно испорчено, и приложение падает. Решение — для восстановления напрямую подключаться к БД, минуя PgBouncer. Такие дела.

› Миф 7: Это программирование, поэтому помощь повсюду: документация, IDE, LLM, ...

PDM

Вот пакетный менеджер мне говорит: «Саня, ну не получилось ничего установить. Какая ошибка — хз. Кстати, установить ничего не получилось! Хочешь квест? Иди в логи и посмотри сам, что пошло не так».

```
> pdm add django-attachments~=1.5 django-bleach~=0.6.1 django-bootstrap3~=12.1.0 django
Adding packages to default dependencies: django-attachments~=1.5, django-bleach~=0.6.1,
  0:00:09 🔒 Lock failed.  ERROR:
  0:00:09 🔒 Lock failed.
See ~/.local/state/pdm/log/pdm-lock-4aauctl8.log for detailed debug log.
[ResolutionImpossible]: Unable to find a resolution
WARNING: Add '-v' to see the detailed traceback
```

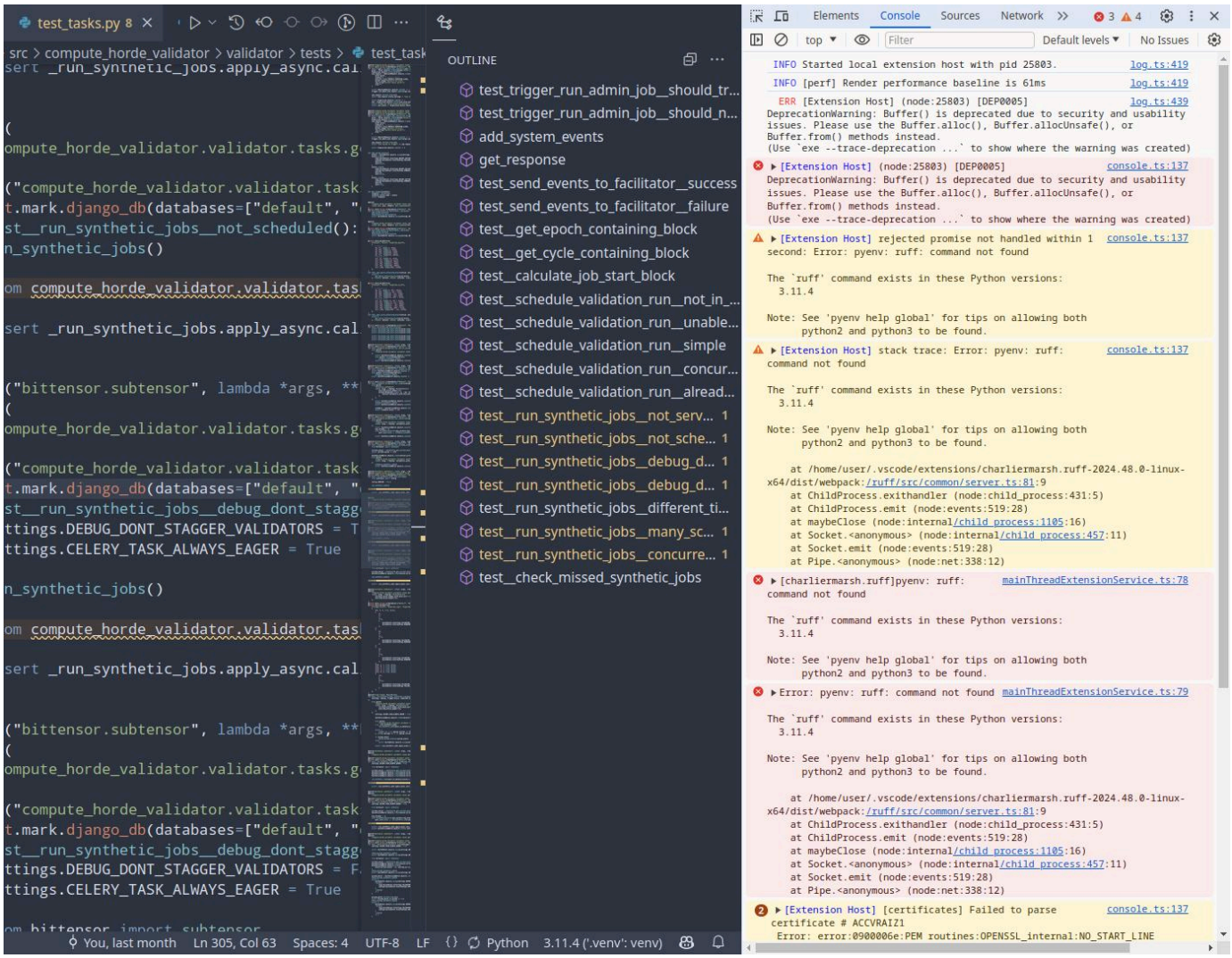
Langchain

Опять мой любимый фреймворк, на этот раз он говорит мне, что в функции `parse_json_markdown` аргумент `json_string` — это markdown строка. Ну теперь-то всё стало понятно!

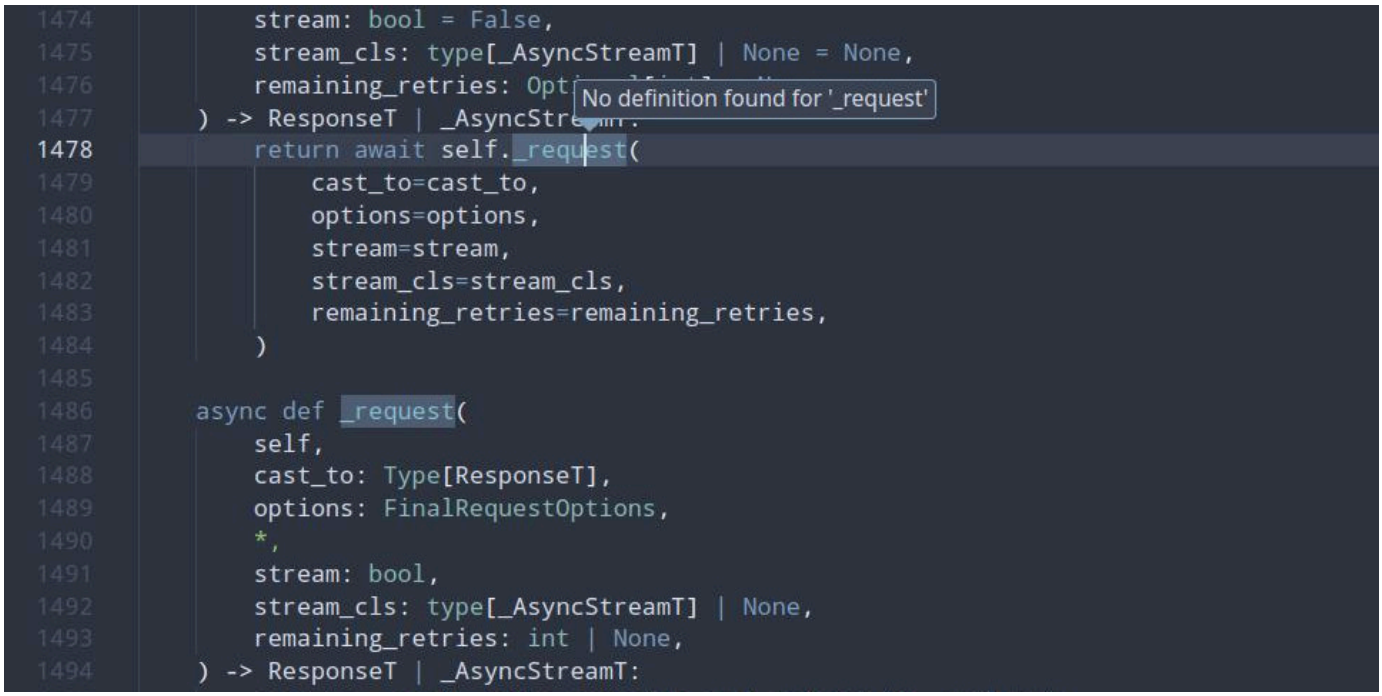
```python
def parse_json_markdown(
    json_string: str,
    *, parser: Callable[[str], Any] = parse_partial_json
) -> dict:
    """
    Parse a JSON string from a Markdown string.
    Args:
        json_string: The Markdown string.
    ...
    """
```

Vscode

Свою IDE я вообще боюсь. Даже если она правильно работает, всё равно в логах developer tools постоянно мясо: какие-то ошибки, deprecation warnings, угрозы, что всё не работает и прочее, и прочее. Мне кажется, там какая-то своя атмосфера, и я только успеваю удивляться, как при всём этом оно работает.



Хотя что это я, какое «работает»? Вот, например, `No definition found for "_request"`, хотя этот самый `_request` всего несколько строчек ниже:



Celery flower

Есть такая софтина: celery flower. Как-то я хотел её хитро настроить и спросил её, что она умеет:

```
> celery flower --help
Usage: celery flower [OPTIONS] [TORNADO_ARGV]...
Web based tool for monitoring and administrating Celery clusters.
Options:
  --help  Show this message and exit.
```

А она мне и отвечает: «я умею только показывать вот эту помощь, которую ты сейчас видишь, а чтобы её посмотреть, нужно меня вызвать с ключом `--help`, как ты и сделал. Что тебе ещё надо, пёс?».

Copilot

Иногда Github Copilot тоже думает, что я пёс, и дополняет строчку только наполовину, обрывая автокомплит где-то посередине. В этом есть резон: умный робот заботится, чтобы я не разучился программировать.

ChatGPT

Лень — моё второе «я». Зачем читать документацию, если можно спросить у ChatGPT?

И вот я в Github задал некоторые «repository variables». Это просто какие-то значения, которые можно использовать внутри github actions. Мне они нужны были, чтобы ставить дополнительные пакеты при деплое, но это не важно. Важно — это как к ним обратиться в yaml-файлах? Сейчас спрошу у ChatGPT, дело на 5 секунд, зашли и вышли...

- Ну да, ну да

Экранов 15 я пытался добиться ответа от языковой модели, а она не знала! Но самое ужасное — что модели не могут сказать «я не знаю», а вместо этого пытаются выдать хоть что-то, неважно, правда это или нет.

Пришлось искать ответ самому. Эх...

Ruff

`Ruff` классный! Он даже форматировать код умеет. Вот, например, у меня довольно большое составное условие в коде, состоящее из 3 подусловий. Я постарался его красиво написать, чтобы легко читалось человеком:

```
last_weights = Weights.objects.order_by('-created_at').first()
if (
    last_weights and
    last_weights.revealed_at is None and
    last_weights.block <= current_block_number - (reveal_weights_interval - reveal_in_a
):
```

Видите, тут условия красиво выровнены. Но у `ruff` есть идея, как сделать ещё красивее! Ну-ка...

```
last_weights = Weights.objects.order_by('-created_at').first()
if (
    last_weights
    and last_weights.revealed_at is None
    and last_weights.block
    <= current_block_number - (reveal_weights_interval - reveal_in_advance)
):
```

Ну да, строчки должны начинаться с `and`, а если какая-то из строк начинается с `<=` — так вообще красота неописуемая!

Или вот — сложный list comprehension, я его написал максимально читаемо:

```
 calls = [
    ("archive", dump.netuid, block)
    for block in list(chain.from_iterable(dumpable_blocks))[::-1]
    if block != 1673
]
```

Но `ruff` говорит, что вот так ещё красивей:

```
 calls = [
    ("archive", dump.netuid, block) for block in list(chain.from_iterable(dumpable_bloc
]
```

Спасибо, `ruff`, так гораздо лучше. Как, говоришь, тебя удалить?

React

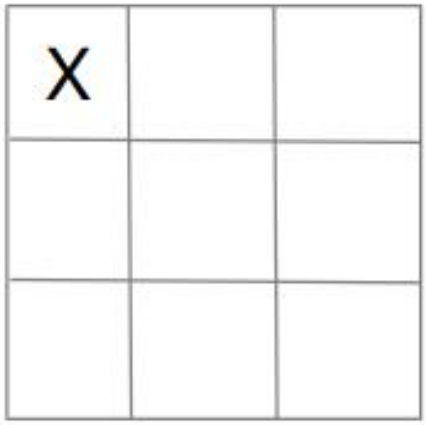
Документация тоже всегда-всегда классная.

Решил я как-то поизучать React. У них там в туториале крестики-нолики, с таким вот кодом:

```
  const [squares, setSquares] =
          useState(Array(9).fill(null));
```

```
function handleClick() {
  const nextSquares = squares.slice();
  nextSquares[0] = "X";
  setSquares(nextSquares);
}
```

Поле — это массив из 9 элементов, при нажатии на любое поле ставится крестик… но только в левое верхнее поле!



Я реально сломал себе мозг, пытаясь понять, ПОЧЕМУ. Может, в javascript всё само как-то по-хитрому сдвигается, а я не знаю? Короче, залип минут на 5, так и не понял ничего, решил читать дальше. А дальше написано:

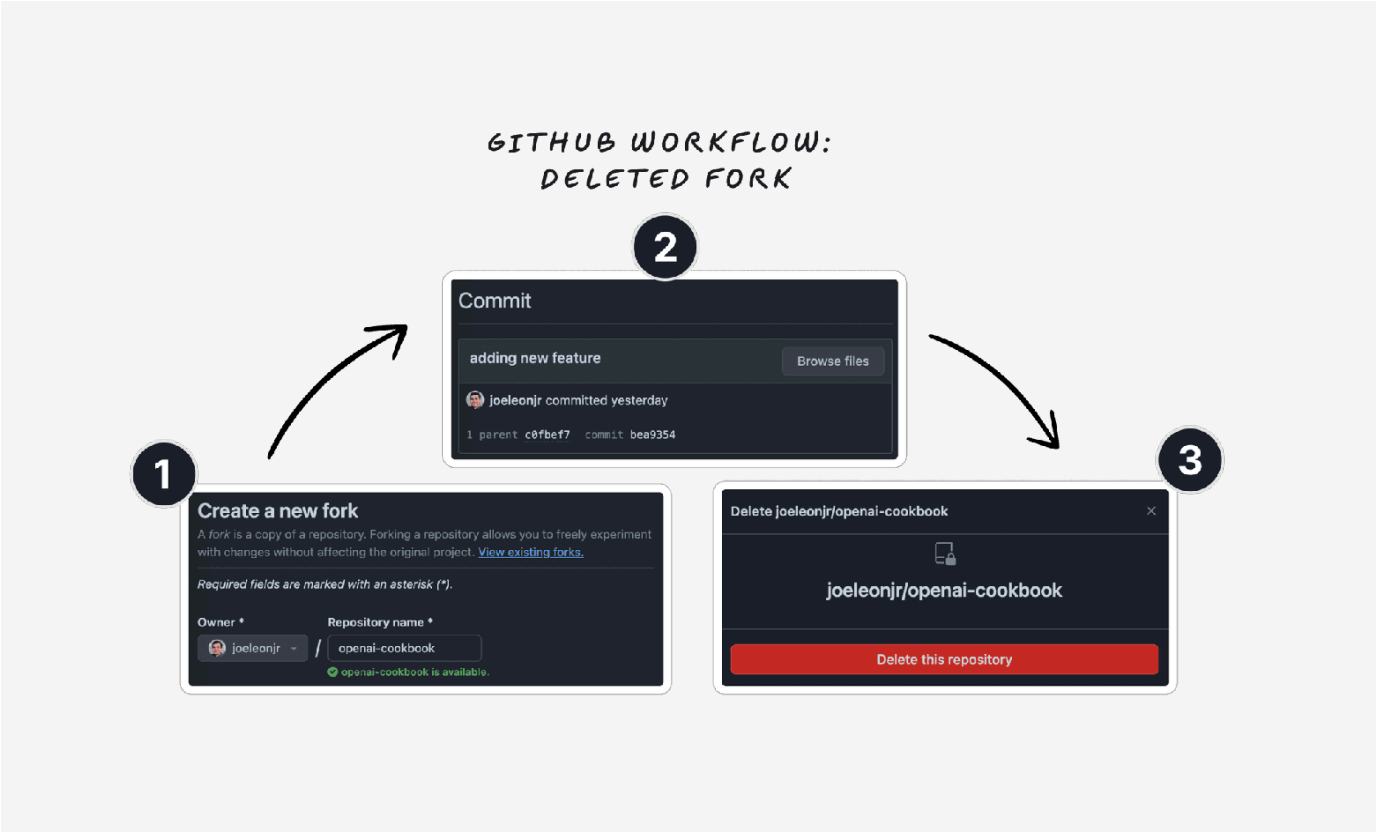
> Теперь вы можете добавлять крестики на доску… но только в верхний левый квадрат. Ваша функция «handleClick» жестко запрограммирована для обновления индекса для верхнего левого квадрата (`0`). Давайте обновим «handleClick», чтобы иметь возможность обновлять любой квадрат.

› Миф 8: Это программирование, поэтому всё надёжно

Недавно я узнал, что на github можно достать коммиты из удалённых или приватных репозиториев.

Раз!

1. Вы создаете публичный репозиторий
2. Вы добавляете код в свой форк
3. Вы удаляете свой форк
4. **Код из удалённого форка всё ещё всем доступен ;)**



Два!

1. У вас есть публичный репозиторий на GitHub.
2. Пользователь форкает ваш репо
3. Вы делаете коммит после его форка (и он никогда не синхронизируют свой форк с вашим репо)
4. Вы удаляете весь репозиторий
5. **Код, который вы закоммитили, всё ещё доступен! ;)**

Три!

1. Вы создаете приватный репо, который в конечном итоге станет публичным
2. Вы создаете ещё одну приватную версию этого репозитория (посредством форка) и пишете там какой-то супер-секретный код
3. Вы делаете свой первый репозиторий общедоступным (а форк остаётся приватным)
4. **Супер-секретный код всем доступен! ;)**

Почитайте, мир не будет прежним: https://trufflesecurity.com/blog/anyone-can-access-deleted-and-private-repo-data-github.

› В заключение

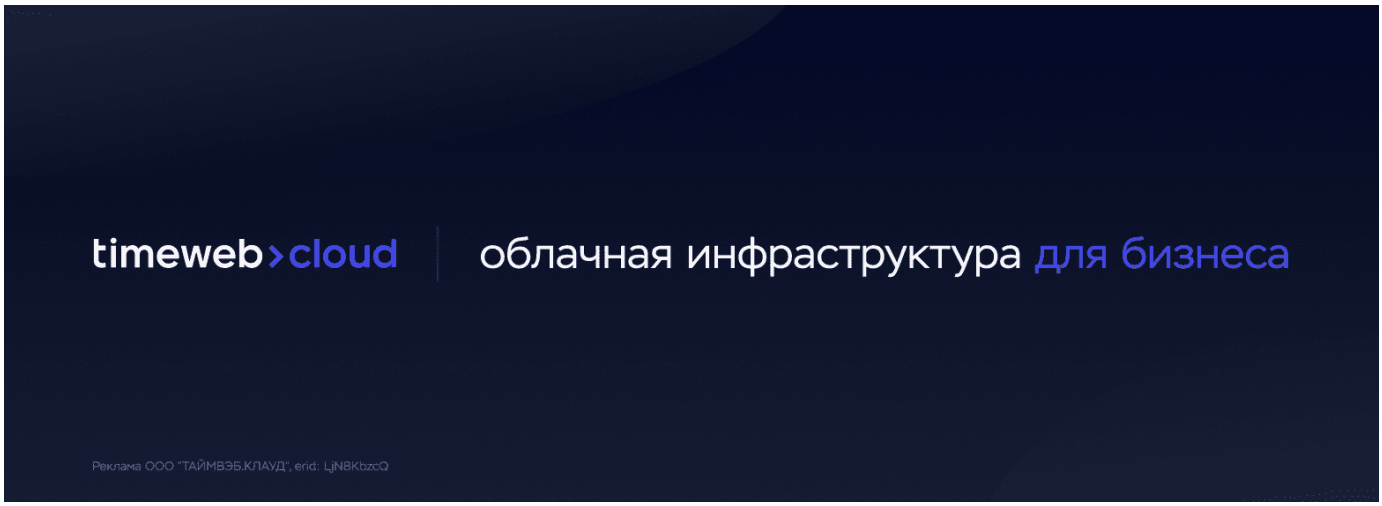
Я не считаю себя каким-то особенным. Я просто работаю и иногда записываю что-нибудь интересное с работы, и я вижу, как мои коллеги периодически тоже постят подобные истории — а значит, страдают все.

Программирование — это постоянная борьба между тем, что мы хотим, и всякой раздражающей, нелогичной и абсурдной хренью, которая нам в этом мешает. Но, может, оно и к лучшему — по крайней мере мы не скучаем! :)

Подписывайтесь, чтобы узнать, как редко я пишу в телегу: Блог погромиста

А ещё я держу все свои проекты у одного облачного провайдера — Timeweb. Поэтому нагло рекламирую то, чем сам пользуюсь — вэлкам:

> **Новости, обзоры продуктов и конкурсы от команды Timeweb.Cloud — в нашем Telegram-канале ↩**



Только зарегистрированные пользователи могут участвовать в опросе. Войдите, пожалуйста.

Есть ли ограничение на длину ответа в опросе?

50.49% Конечно есть, и вот почему: Разработчики всегда помнят о необходимости устанавливать ограничения для полей ввода в формате HTML, чтобы улучшить взаимодействие с пользователем и сохранить целостность данных. Без надлежащих ограничений пользователи могут непреднамеренно вводить данные, объем которых превышает разумные пределы, что может привести к потенциальным ошибкам или путанице. Ограничивая длину или формат вводимых данных, разработчики могут с самого начала указывать пользователям правильные данные, что предотвращает ошибки при проверке и повышает удобство использования форм на веб-сайтах или в приложениях. Например, установка максимального количества символов для имени или поля электронной почты предотвращает ненужные задержки, позволяя пользователям узнать ожидаемый размер вводимых данных. Другой важной причиной является безопасность. Неограниченные поля ввода могут быть использованы злоумышленниками с помощью таких методов, как атаки на переполнение буфера, SQL-инъекции или другие формы фальсификации данных. Устанавливая ограничения на типы, длину и значения данных, разработчики могут уменьшить вероятность атаки и защититься от этих потенциальных угроз. Например, установка числовых шаблонов проверки или регулярных выражений в полях ввода телефонных номеров или данных кредитной карты гарантирует, что обрабатываются только действительные, ожидаемые типы данных, что сводит к минимуму риски. Наконец, ограничения на ввод данных способствуют оптимизации производительности. Если оставить поля ввода без ограничений, это может привести к вводу большого количества ненужных данных, что может привести к перегрузке ресурсов сервера и снижению производительности, особенно в приложениях с интенсивным трафиком. Устанавливая соответствующие ограничения для полей ввода, разработчики могут снизить вероятность возникновения проблем с производительностью и обеспечить бесперебойную работу систем, эффективно обрабатывая запросы. В целом, применение ограничений для ввода — 308

является важным аспектом проектирования форм, который улучшает взаимодействие с пользователем, повышает безопасность и производительность системы.

49.51% Нет, никакого ограничения нет, ответ может быть любой длины 302

Проголосовали 610 пользователей. Воздержались 135 пользователей.

Теги: timeweb_статьи, IT курсы, обучение программированию, лютая дичь, боль, тлен

Хабы: Блог компании Timeweb Cloud, Python, Веб-разработка, Карьера в IT-индустрии, Программирование

+224 279 114

Timeweb Cloud

То самое облако

604 Карма 0 Рейтинг

Александр Гончаров @kesn

Какой-то чувак

Подписаться

Хабр Карьера Хабр Эксперты Сайт Telegram

Комментарии 114

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

DmitryOlkhovoi
21 час назад

Меня заставили повайбкодить

Сложный 18 мин 22K

Кейс

+89 97 93

RationalAnswer
13 часов назад

Тотальный блэкаут на юге Европы, а также чудеса нейро-лизоблюдия от ChatGPT

11 мин 11K

Дайджест

+32 12 65

lozhnikov
22 часа назад

Как доказывали теорему о четырех красках. Часть 1

Простой 11 мин 2.8K

FAQ

+30 23 1

MrSotnik
8 часов назад

Новый язык от 1С: Зачем? Кому? Стоит ли лезть?

5 мин 14K

+29 22 50

alizar
9 часов назад

Ян Лекун, создатель LeNet, формата DjVu и адвокат опенсорса

Средний 7 мин 1.3K

Обзор

+26 8 6

El_Gato_Grande
9 часов назад

Как ИИ-контент проклял интернет и почему это закономерно

8 мин 3K