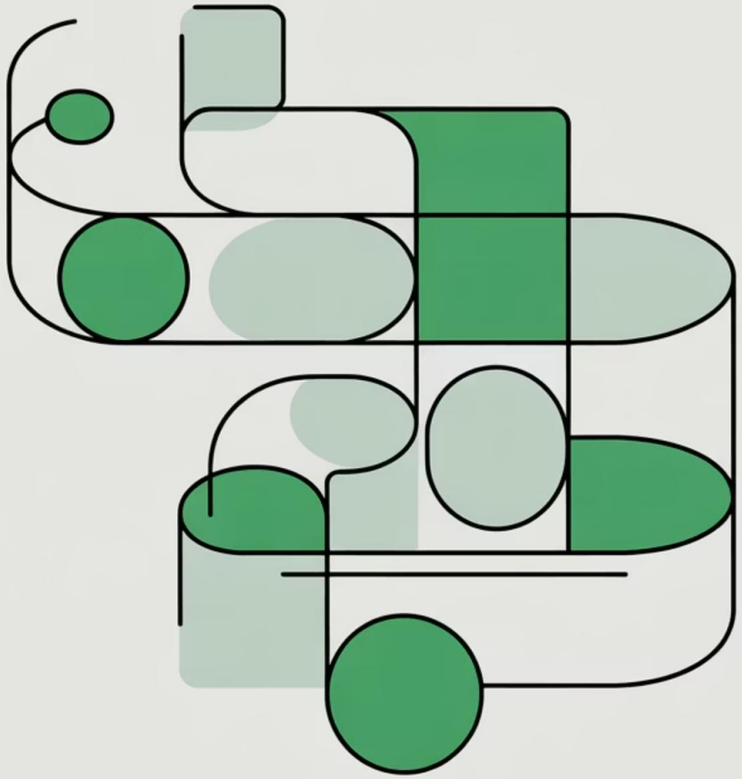
Apache AirFlow



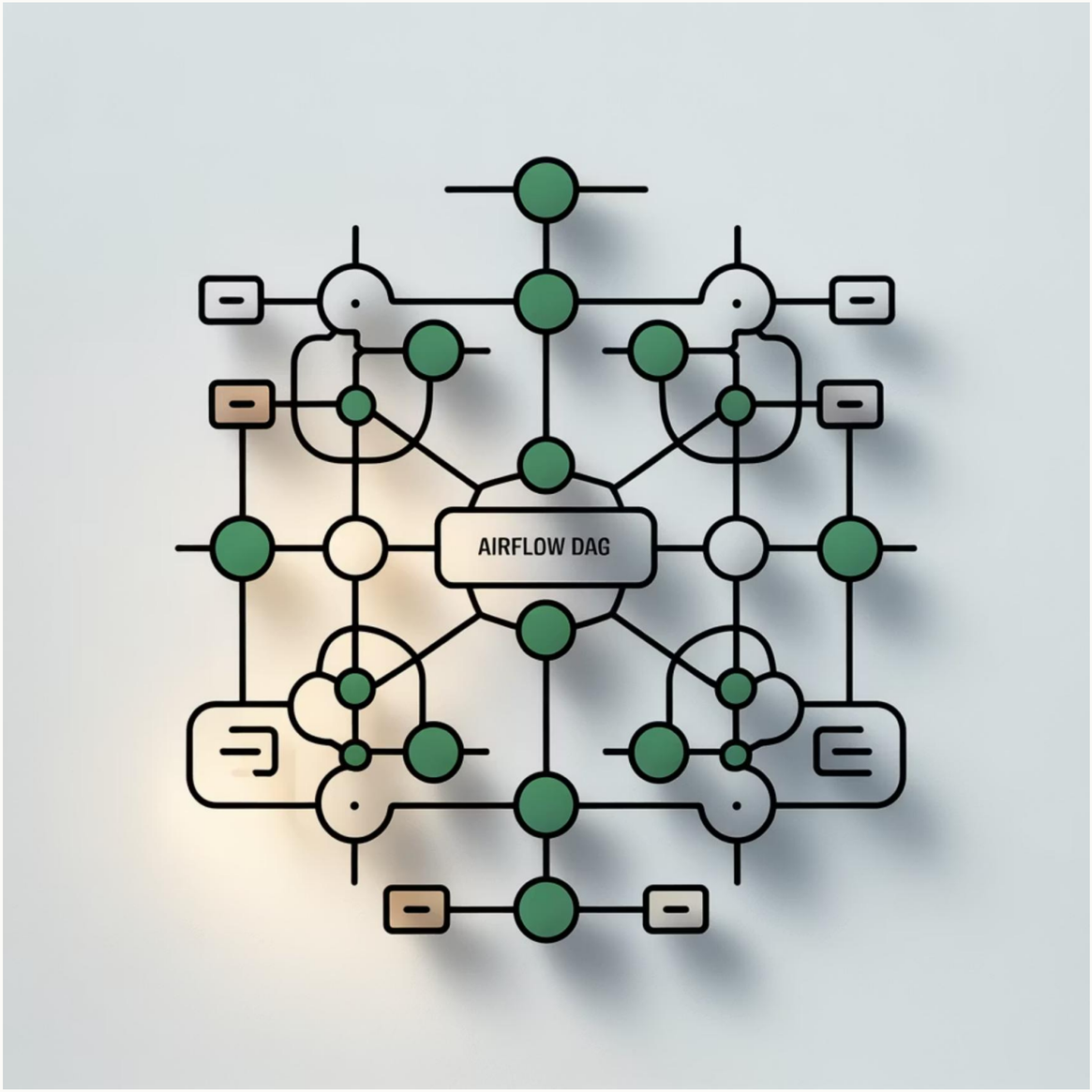
Airflow TaskFlow API: Эволюция разработки DAG в Apache Airflow

Актуальный подход к созданию пайплайнов данных

Проблема классического подхода

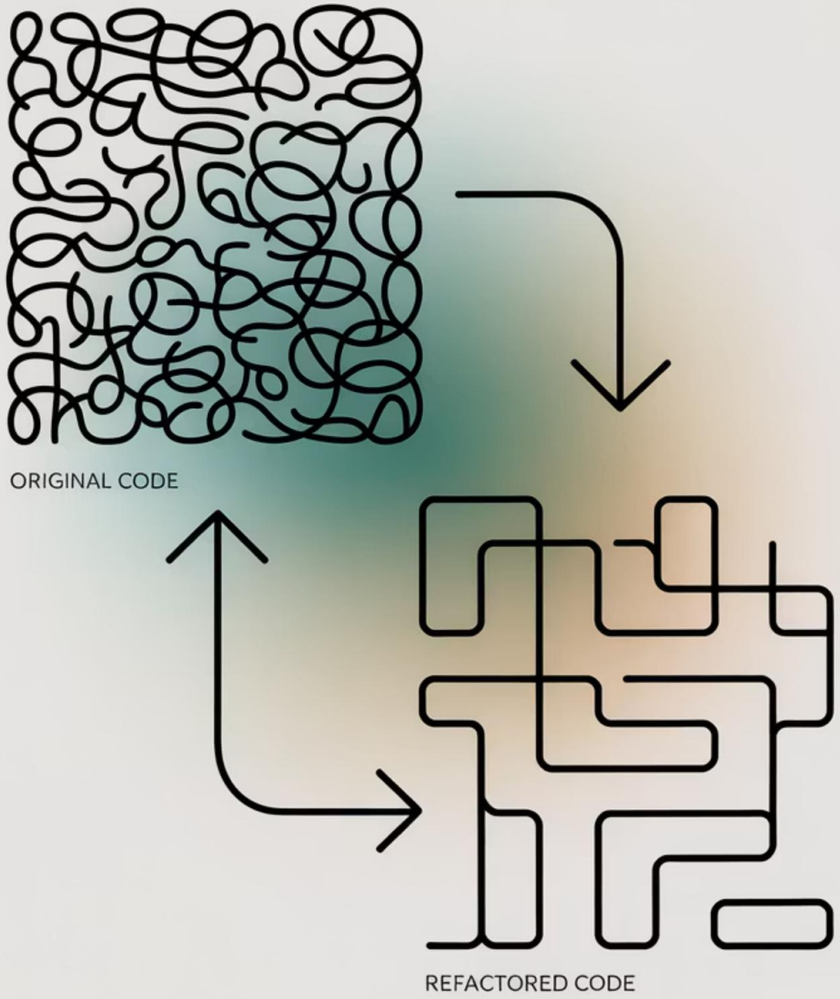
Классический код - много шаблонов

```python
# КЛАССИЧЕСКИЙ ПОДХОД - МНОГО ШАБЛОННОГО КОДА
def extract_func(**context):
    data = "source_data"
    context['ti'].xcom_push(key='extracted_data', value=data)
    return data

def transform_func(**context):
    data = context['ti'].xcom_pull(task_ids='extract', key='extracted_data')
    return data.upper()

extract_task = PythonOperator(
    task_id='extract',
    python_callable=extract_func,
    provide_context=True
)

transform_task = PythonOperator(
    task_id='transform',
    python_callable=transform_func,
    provide_context=True
)

extract_task >> transform_task
```



Каждая задача требует создания оператора, ручного управления XCom и явного указания зависимостей через операторы >>

Основные проблемы: Громоздкий код, множество шаблонов, сложность чтения, ручное управление передачей данных между задачами



Решение: TaskFlow API

Что это такое?

Новый способ описания DAG'ов через декораторы Python, который автоматизирует XCom и построение зависимостей

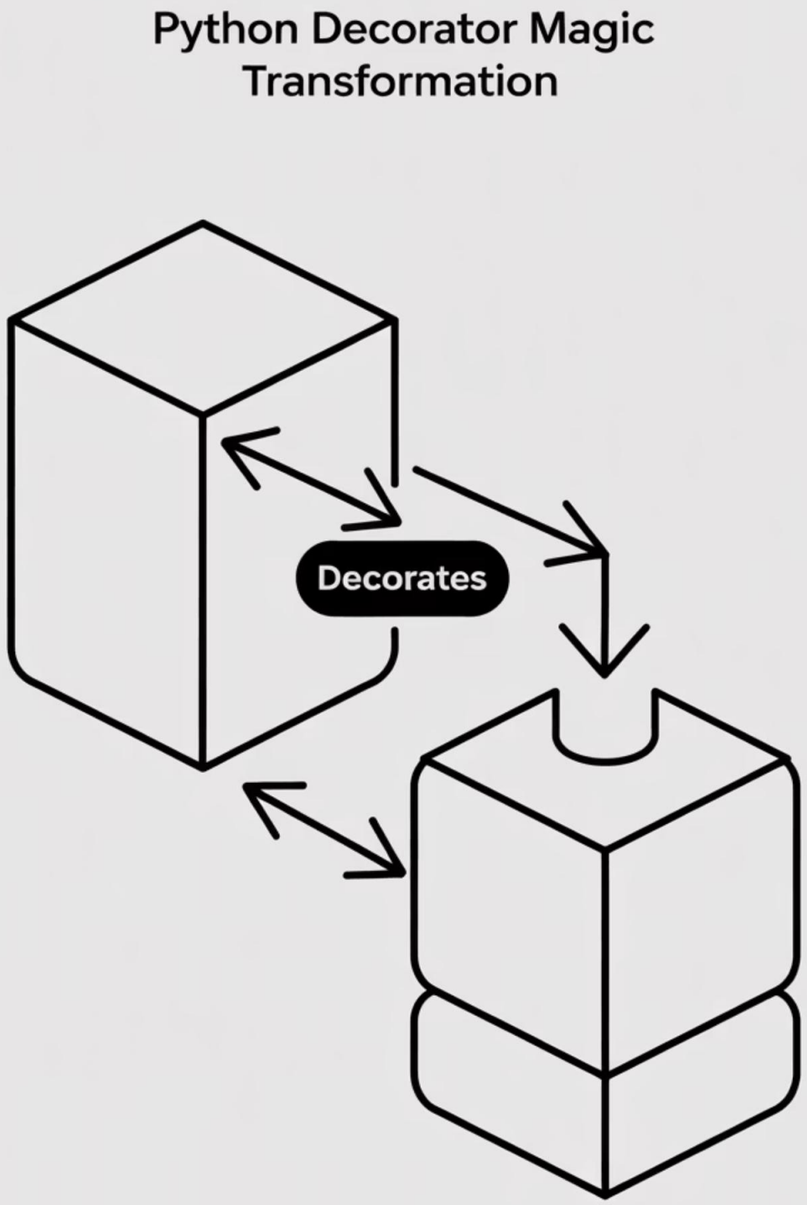
Ключевая идея

"Пишите функции, а Airflow сделает их задачами"

Python-ориентированный подход к разработке пайплайнов

Главные преимущества

- Автоматическое управление XCom
- Естественные зависимости
- Чистый читаемый код



Волшебство декоратора @task

```python
from airflow.decorators import task, dag
from datetime import datetime

@dag(start_date=datetime(2025, 1, 1), schedule="@daily")
def my_pipeline():

    @task
    def extract():
        data = "source_data"
        return data  # автоматически XCom!

    # Всего одна строка превращает функцию в задачу!
    extract_task = extract()

my_pipeline_dag = my_pipeline()
```

Создание PythonOperator

@task автоматически создает PythonOperator под капотом

Автоматический XCom

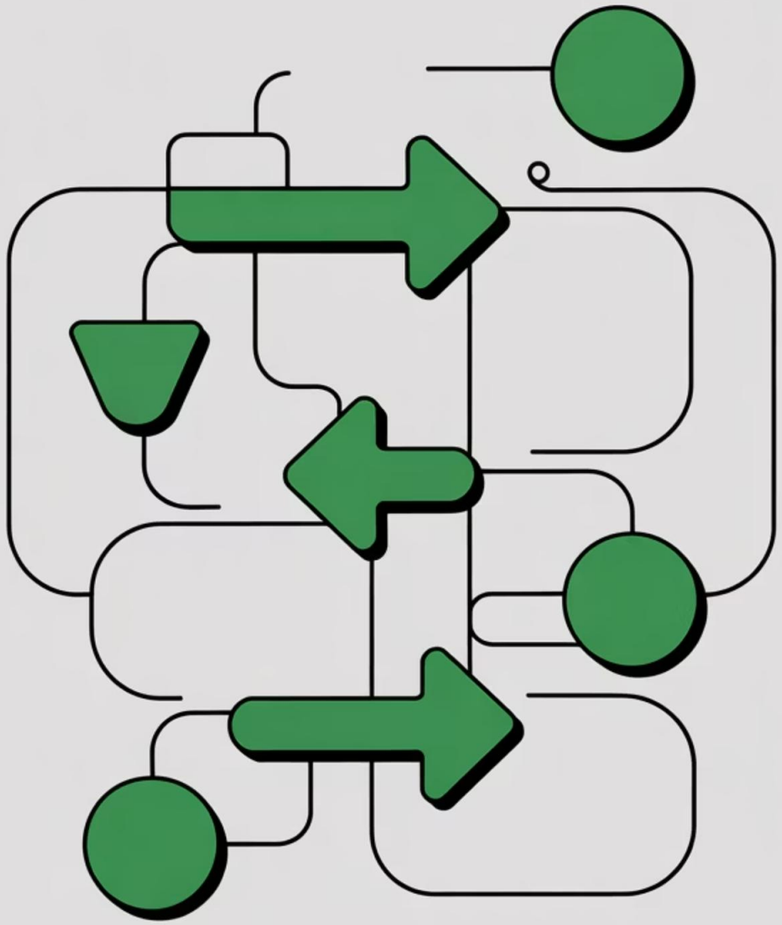
Возвращаемое значение автоматически сохраняется в XCom

Создание задачи

Вызов функции создает задачу в DAG

Автоматический XCom - забудьте про ti.xcom_pull!

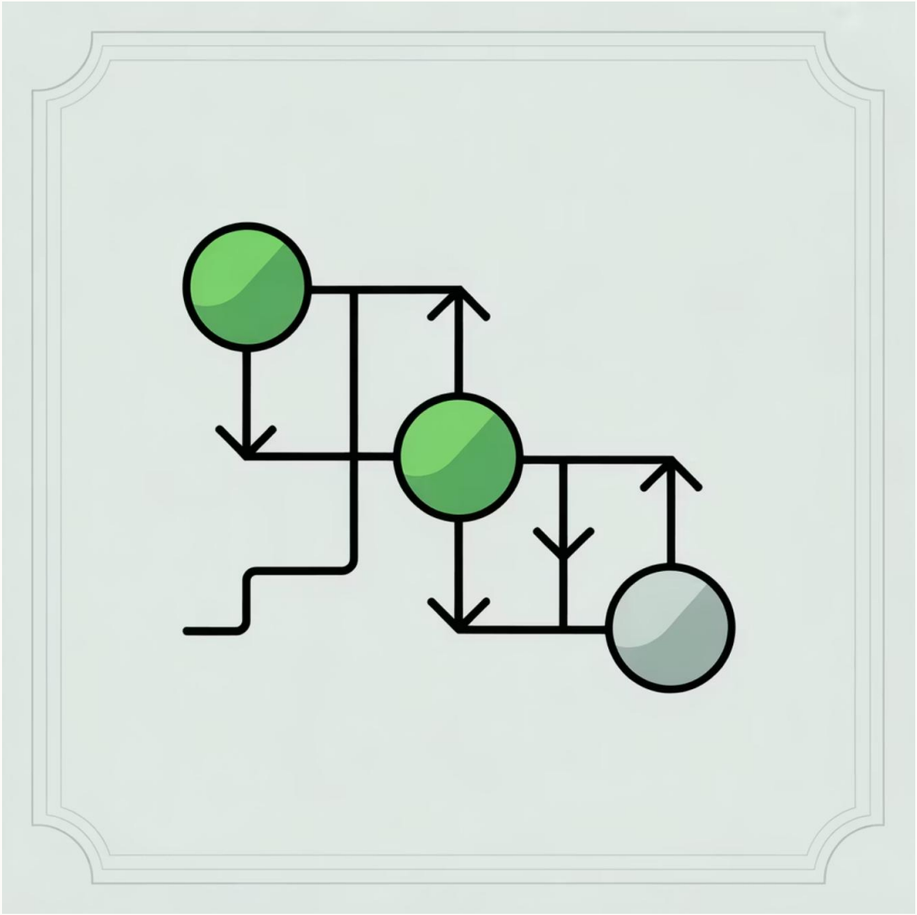
```python
@dag(start_date=datetime(2025, 1, 1), schedule="@daily")
def data_pipeline():

    @task
    def extract():
        return "my_data"  # Автоматически в XCom

    @task
    def transform(data: str):  # Автоматически из XCom!
        return data.upper()

    @task
    def load(data: str):
        print(f"Loading: {data}")

    # Зависимости через вызовы функций!
    raw_data = extract()
    transformed_data = transform(raw_data)
    load(transformed_data)
```



TaskFlow автоматически управляет передачей данных между задачами. Когда функция возвращает значение - оно сохраняется в **XCom**. Когда другая функция принимает параметр с соответствующим именем - значение автоматически подтягивается из **XCom**.

Зависимости через вызовы функций

```python
@dag(start_date=datetime(2025, 1, 1), schedule="@daily")
def complex_pipeline():

    @task
    def extract():
        return {"data": "value"}

    @task
    def clean_data(raw_data):
        return raw_data["data"].upper()

    @task
    def validate_data(cleaned_data):
        return len(cleaned_data) > 0

    @task
    def load_data(cleaned_data, is_valid):
        if is_valid:
            print(f"Loading: {cleaned_data}")

    # Граф строится на основе ВЫЗОВОВ функций!
    raw_data = extract()
    cleaned = clean_data(raw_data)
    is_valid = validate_data(cleaned)

    # Множественные зависимости - просто и понятно!
    load_data(cleaned, is_valid)
```



Естественные зависимости

- Зависимости определяются через вызовы функций
- Сложные зависимости создаются простой передачей нескольких аргументов

Особенности и ограничения TaskFlow

Сериализуемость

Возвращаемое значение должно быть сериализуемым (**JSON** или **Pickle**).

Размер данных

Избегайте передачи больших объектов через **XCom**.

Pandas и Pickle

Для **DataFrame** требуется **enable_xcom_pickling=True** или кастомный **xcom_backend**.

TaskFlow — всё ещё использует XCom, поэтому важно соблюдать ограничения на типы и размеры передаваемых данных.

Расширенные возможности @task

Виртуальное окружение

```python
@task.virtualenv(
    requirements=["pandas", "scikit-learn"],
    python_version="3.9"
)
def train_model(data):
    from sklearn.ensemble import RandomForestClassifier
    # ... тренировка модели
```

Идеально для ML задач с собственными зависимостями

Docker контейнер

```python
# Docker контейнер
@task.docker(
    image="python:3.9-slim",
    command="python my_script.py"
)
def process_in_container():
    return "container_task_complete"
```

Полная изоляция для критических операций

Параметры задач

```python
@task(
    retries=3,
    retry_delay=timedelta(minutes=2),
    execution_timeout=timedelta(minutes=30),
    pool="data_processing_pool"
)
def flaky_operation():
    # Операция, которая может иногда падать
    import random
    if random.random() < 0.2:
        raise Exception("Временная ошибка")
    return "success"
```

Все параметры операторов доступны в декораторе

Плюсы и ограничения

✓ Преимущества

- **Чистый код** - меньше шаблонного кода, больше бизнес-логики
- **Автоматический XCom** - забываем про ручное управление данными
- **Естественные зависимости** - граф строится на вызовах функций
- **Лучшая типизация** - IDE понимает код, работает автодополнение
- **Проще тестирование** - функции можно тестировать отдельно

⚠ Ограничения

- **Не для всех операторов** - BashOperator, SSHOperator остаются классическими
- **Кривая обучения** - нужно перестроить мышление
- **Отладка** - иногда сложнее понять происходящее под капотом
- **Избыточность** - для простых DAG без обмена данными может быть излишне

Best Practices и следующие шаги

1 **Осмысленные имена функций**

Имя функции становится именем задачи в UI Airflow

2 **Используйте типизацию**

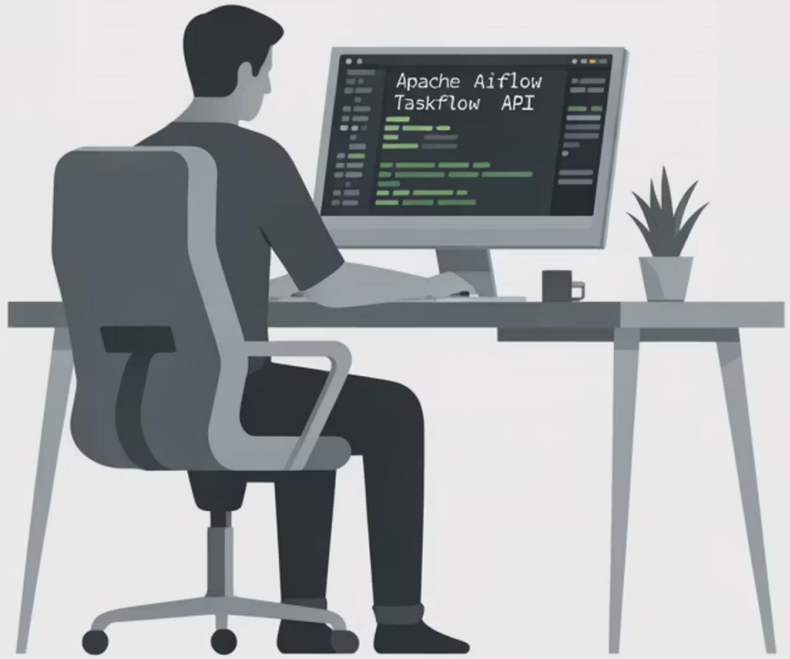
Аннотации типов помогают IDE и предотвращают ошибки

3 **Делите на мелкие функции**

Одна задача = одна ответственность

4 **Комбинируйте подходы**

TaskFlow для логики, классические операторы где нужно



Ваши следующие шаги:

01

Попробуйте на практике

Перепишите один свой DAG на TaskFlow

02

Изучите документацию

Углубитесь в официальные материалы Apache Airflow

03

Присоединяйтесь к сообществу

t.me/marat_notes

Thank You

Спасибо за внимание!

Вопросы?

Подписывайтесь на канал и группы

- Telegram-канал: t.me/marat_notes
- Обучающие видео: https://vkvideo.ru/@club231048746, https://www.youtube.com/@marat_notes
- Репозитарий: https://github.com/MaratNotes/marat_notes