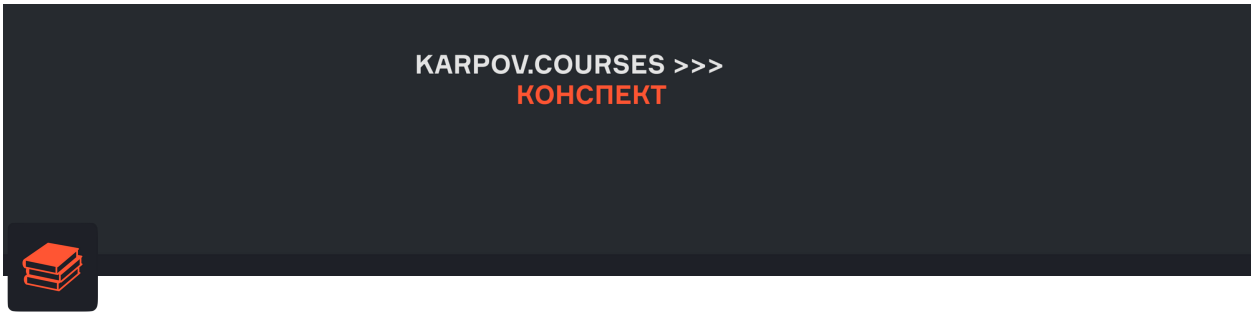


> Конспект > 12 урок > Полезные вещи в разработке

> Оглавление

> Фиксирование библиотек

В текущем и последующих степах рассмотрим полезные вещи в разработке на примере простого веб-приложения.

Совет №1

Хорошей практикой считается фиксировать все библиотеки, которые вы используете в проекте, а также фиксировать их версии

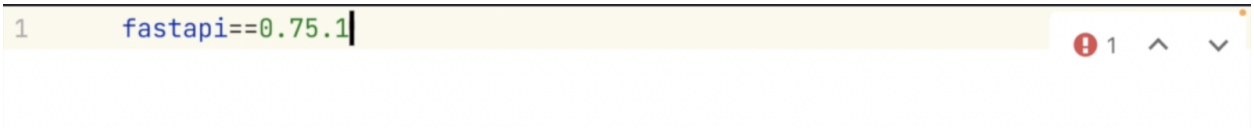
Вы уже знаете, что перечисление библиотек происходит в файле requirements.txt. Правда есть небольшая проблема: например, вы хотите прописать библиотеку *fastapi*, но не знаете, какую версию вы хотите поставить. В целом, можно посмотреть в интернете, какая последняя версия сейчас используется и прописать ее. Но есть небольша хитрость:

```
# Ключ -U попросит pip установить библиотеки и обновить до той версии, которую мы запросим
pip install -Ur requirements.txt
```

Выполнив эту команду, у вас будут установлены нужные библиотеки, но также, если обратить внимание на лог, можно заметить, что у библиотек будут прописаны версии. Например,

```
Successfully installed anyio-3.5.0 fastapi-0.75.1 idna-3.3 pydantic-1.9.0 sniffio-1.2.0 starlette-0.17.1 ty
ping-extensions-4.1.1
```

Теперь вы можете скопировать эту версию и прописать в файл requirements.txt



> Переменные окружения

Совет №2

Хорошей практикой считается скрывать логины/пароли а также всю информацию, использование которой сторонними лицами может как-то навредить вашей системе.

Решение данной проблемы подразумевает использование Переменных окружения

Переменные окружения — именованные переменные, содержащие текстовую информацию, которую могут использовать запускаемые программы. Такие переменные могут содержать общие настройки системы, параметры графической или командной оболочки, данные о предпочтениях пользователя и многое другое.

Разберем пример:

```
def get_db():
    with psycopg2.connect(
        user="robot-startml-ro"
        password="pheiphohahj1Vaif",
        host="postgres.lab.karpov.courses"
        port=6432,
        database="startml",
    ) as conn:
        return conn
```

Есть метод, отвечающий за подключение к базе данных, но вся информация для подключения написана в открытом доступе. Давайте посмотрим, как это можно исправить

Решение:

Вместо того, чтобы прописать логин/пароль и т.д в открытом коде, мы будем принимать их из переменных окружения.

Напомним, что само приложение будет запускаться через консоль, а консоль в свою очередь является неким подобием языка программирования, а значит, по аналогии, может иметь переменные. И когда вы запускаете программу, данная программа может получить доступ ко всем переменным окружения, из

которых она была вызвана. Соответственно, в эти переменные окружения мы можем положить логин и пароль.

Чтобы считать переменные окружения в вашем файле, нужно импортировать модуль os

```
import os
```

Как теперь прописать переменные окружения в коде?

```
def get_db():
    with psycopq2.connect(
        user=os.environ["POSTGRES_USER"]
        password=os.environ["POSTGRES_PASSWORD"]
        host=os.environ["POSTGRES_HOST"]
        port=os.environ["POSTGRES_PORT"]
        database=os.environ["POSTGRES_DATABASE"]
    ) as conn:
        return conn
```

Когда вы импортируете модуль os, у вас становится доступен словарь environ. У этого словаря можно брать значения по ключу. Ключом, как раз, и будут названия переменных окружения.

Обратите внимания, что переменные окружения обычно объявляются в верхнем регистре

Как же теперь запустить приложение и передать ему нужные переменные окружения? Здесь есть несколько вариантов, и мы пошагово придем к самому удобному из них.

Способ №1

Можно передавать все нужные переменные окружения непосредственно перед вызовом команды unicorn app:app --reload

```
POSTGRES_USER=helLo POSTGRES_PASSWORD=world uvicorn app:app --reload
```

Согласитесь, такой вариант совсем не удобен

Способ №2

Можно объявить через export нужные переменные перед запуском.

```
# Объявляем переменные окружения
export POSTGRES_USER=hello

# Запускаем приложение
uvicorn app:app --reload
```

Но такой способ тоже не совсем удобен, так как нужно сначала перечислить все переменные и только потом запускать программу. И если мы закроем терминал, то нужно будет снова перечислять все переменные заново.

Так какой же самый удобный способ работы с переменными окружения?

Использование так называемых .env файлов. О них будет подробнее рассказано на следующем степе

> .env файлы

Чтобы избежать все те неудобства при работе с переменными окружения, с которыми вы столкнулись на предыдущем степе, были придуманы .env файлы.

Создается файл .env, куда переносятся все необходимые переменные окружения. На самом деле, название файла не столь принципиально, но .env является общепринятым. Пример файла:

```
POSTGRES_USER="robot-startml-ro"
POSTGRES_PASSWORD="pheiphOhahj1Vaif"
POSTGRES_HOST="postgres.lab.karpov.courses"
POSTGRES_PORT=6432
POSTGRES_DATABASE="startml"
```

Чтобы все переменные окружения появились, нужно перед запуском программы выполнить команду:

```
source .env
```

Но данное решение снова имеет недостатки: во-первых, все равно приходится в консоли сначала выполнять одну команду, затем другую, а во-вторых, в Windows отсутствует команда source.

Чтобы решить данную проблему, в python существует библиотека, которая умеет правильно считывать .env файлы. В requierements.txt нужно добавить библиотеку python-dotenv

Пример файла requierements.txt

```
fastapi==0.75.1
uvicorn==0.17.6
pCycopq2-binary==2.9.3
python-dotenv==0.20.0
```

Затем в самом python файле нужно импортировать библиотеку

```
from dotenv import load_dotenv
```

Чтобы переменные окружения стали доступны в коде, нужно вызвать load_dotenv() следующим образом

```
if __name__ == '__main__':
    load dotenv()
```

При таком подходе вам не нужно заранее запускать какие-то скрипты, прописывать переменные окружения перед каждый запуском. Вы прописываете один раз в файле .env все, что вам может понадобиться, и с помощью библиотеки python-dotenv считываете все переменные.

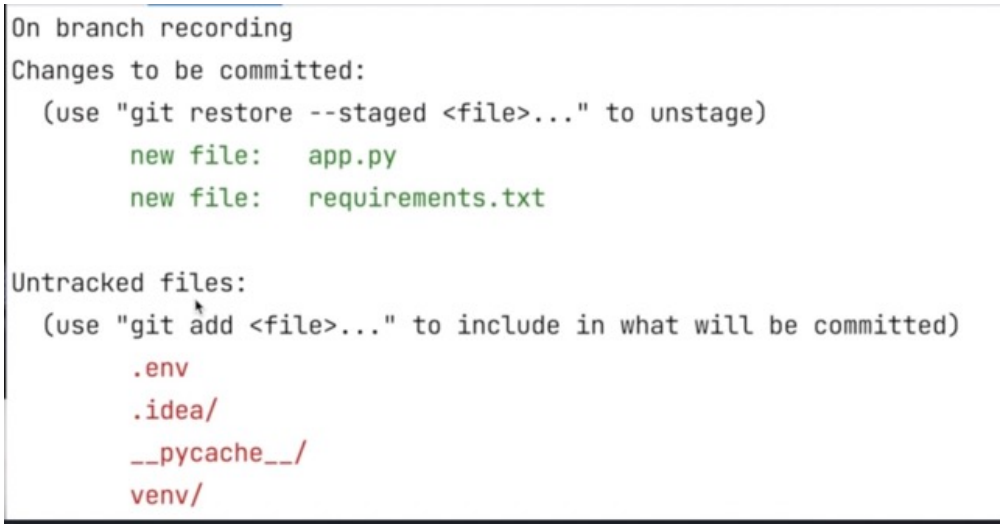
> Использование .gitignore

Совет №3

Хорошей практикой считается добавлять файлы, которые не должны быть отслеживаемыми, в файл .gitignore

---

Допустим, после всех изменений из предыдущих шагов, вы решите закоммитить свои изменения. В таком случае команда git status может вывести примерно следующий результат:



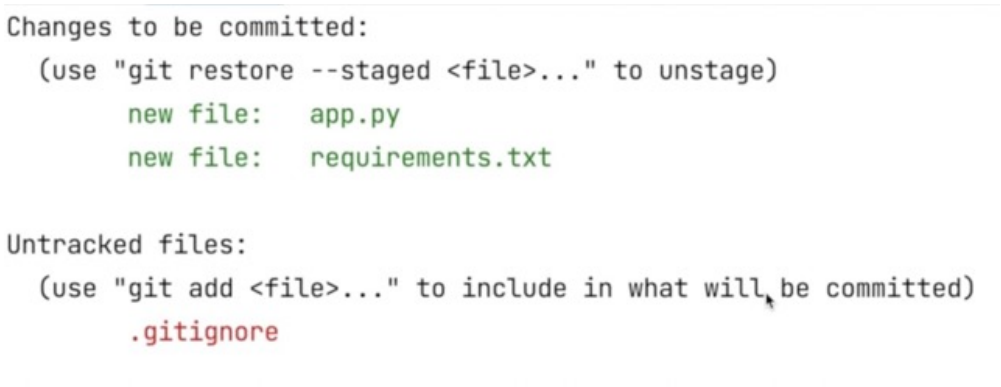
Можно заметить, что `git` предлагает отслеживать файл `.env` , но это не то, что нам нужно. Так как вся идея этого файла, как раз, состоит в том, чтобы он не был доступен публично.

В таком случае git предлагает создание специального файла, который называется `.gitignore`

Вы создаете файл `.gitignore` и в нем перечисляете те файлы, которые не хотите, чтобы были запушены. Пример файла:

```
1 .idea/
2 .env
3 __pycache__/
4 venv/
```

Если сделать `git status` после добавления файла .gitignore для примера выше, можно увидеть следующий результат:



Все файлы, которые мы не хотели версионировать, пропали, осталось только сделать `git add` для самого файла `.gitignore`.

> Проблема SQL инъекций

Давайте рассмотрим одну серьезную, проблему, связанную с безопасностью, с которой можно столкнуться при разработке системы. Рассмотрим пример кода:

```
@app.get("/user")
def get_user(limit, db: connection = Depends (get_db)):
    with db. cursor (cursor_factory=RealDictCursor) as cursor:
        cursor.execute(
            f"""
            SELECT *
            FROM "user"
            LIMIT {limit}
            """,
        )
        return cursor. fetchall()
```

Например, есть эндпоинт, который возвращает пользователей по заданному лимиту. В данном коде само значение лимита подставляется через синтаксис `f-строк`. На первый взгляд, в такой версии эндпоинта нет ничего плохого. Но, допустим, ваше приложение было выставлено в интернет: к нему может подключиться любой пользователь и забрать какие-то данные. В ваше приложение пришли злоумышленники и пытаются взломать, производя такой GET-запрос

```
http://localhost:8000/user?limit=10;SELECT * FROM post LIMIT 10
```

В таком случае результат будет не таким, как вы ожидаете: вернуться данные из таблицы `post` вместо таблицы `user`.

Но почему так происходит?

Если обратиться к коду выше, можно заметить, что мы принимаем `limit` и подставляем его f-строкой, Поэтому запрос злоумышленника выше фактически подставляется "вслепую" в место, где должен быть `limit`

Т.е после подстановки получилось примерно следующее:

```
@app.get("/user")
def get_user(limit, db: connection = Depends(get_db)):
    with db.cursor(cursor_factory=RealDictCursor) as cursor:
        cursor.execute(
            f"""
            SELECT *
            FROM "user"
            LIMIT 10; SELECT * FROM post LIMIT 10
            """,
        )
        return cursor.fetchall()
```

У нас есть два запроса, и после выполнения `cursor.fetchall()` будет возвращен результат самого последнего запроса.

Такая уязвимость называется SQL инъекция, и основана она на том, что такие символы, как, например, `;` не экранируются, а передаются, как есть, а не заменяются на безопасную последовательность.

Современные фреймворки имеют защиту от такой уязвимости:

Идея в том, что вместо передачи параметров через f-строку, нужно передавать параметры через второй аргумент метода `execute`. Эти параметры передаются при помощи словаря и подставляются затем в сам запрос. Приведем пример:

```
@app. get("/user")
def get_user(limit, db: connection = Depends(get_db)):
    with db.cursor (cursor_factory=RealDictCursor) as cursor:
        cursor.execute(
            f"""
            SELECT *
            FROM "user"
            LIMIT %(limit)s
            """,
            {"limit": Limit},
        )
        return cursor. fetchall()
```

Такое простое действие способно защитить от SQL инъекции.

> Вынесение настроек в конфиг файлы

Конфиг файлы нужны для вынесения тех параметров, которые нужно установить при запуске программы и которые не стоит принимать от пользователей.

Например, нужно написать эндпоинт, который будет возвращать ленту пользователей, начиная с определенной даты.

```
@app.get("/user/feed")
def get_user_feed (user_id: int, limit: int = 10, conn: connection = Depends(get_db)) :
    FEED_START_DATE = '2022-01-01'
    with conn.cursor() as cur:
```

```
        cur. execute(
            """
            SELECT *
            FROM feed_action
            WHERE user_id = %(user _id)s
                AND time >= FEED_START_DATE
            LIMIT %(limit)s
            """,
            {"user_id": user_id, "limit": limit}
        )
        return cur. fetchall()
```

Как один из вариантов, мы можем вынести нужную нам дату в константу, как показано на примере выше. Но в то же время, если это константа, то ее не очень хорошо хранить в файлах с исходным кодом: через какое-то время может поступить запрос поменять дату, например, и в таком случае нужно будет менять файл, делать коммит и т.д., а это может понести дополнительные накладные расходы.

Совет №5

Хорошей практикой считается вынесение отдельных частей кода в конфиг файлы

В целом, это делается по аналогии с тем, как мы выносили данные в `.env` файл. Создадим специальный файл для конфигураций, который называется `params.yaml`. yaml - язык разметки для хранения информации в формате ключ-значение. Сам файл для примера выше может выглядеть следующим образом:

feed_start_date: "2022-01-01"

Как теперь считать этот файл в коде?

Для работы с `yaml` файлами, нужно установить библиотек `pyyaml`, прописав ее в `requirements.txt`, а затем импортировать:

import yaml

Далее нужно считать сам файл `params.yaml`. Для этого напишем функцию config()

```
def config():
    with open("params.yaml", "r") as f:
        return yaml.safe_load(f)
```

Для чтения файлов в python существует функция `open()`. Первый аргументом в нее передается название файла или полный путь к нему, а вторым - в каком режиме будет взаимодействие с файлом (чтение, запись и т.д). Затем с помощью функции `safe_load` из библиотеки `yaml` загружаем yaml файл. `safe_load` вернет словарь из пар ключ-значение в том порядке, в котором они были записаны в файле `params.yaml`

Теперь `config()` можно получить через Dependency Injection в самом эндпоинте

```
@app.get("/user/feed")
def get_user_feed (user_id: int, limit: int = 10, conn: connection = Depends(get_db), config: dict = Depends(config)) :
    with conn.cursor() as cur:
        cur. execute(
            """
            SELECT *
            FROM feed_action
            WHERE user_id = %(user _id)s
```

```
                AND time >= %(start_date)s
            LIMIT %(limit)s
            """,
            {"user_id": user_id, "limit": limit, "start_date": config["feed_start_date"]}
        )
        return cur. fetchall()
```

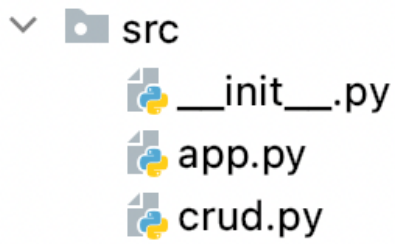
Когда же стоит выносить что-то в конфиг?

Четких инструкций здесь нет, но есть рекомендация: в конфигурационные файлы стоит выносить те настройки программы, которые не будут приниматься от пользователя. Эти настройки не должны меняться от запроса к запросу.

> Разделение кода на модули

Когда проект достигает больших размеров, становится неудобно писать весь код в одном файле. Вам захочется разбить его на множество файлов. Более того, когда файлов станет очень много, вы захотите разбивать их по папкам и импортировать файлы из разных папок друг в друга.

В первую очередь стоит обратить внимание, как в `python` используются папки. `python` не оперирует понятием *"папка",* он оперирует понятем *"модуль"*. Модуль в python - это папка, из которой вы можете импортировать файлы, переменные и т.д. Чтобы сделать папку модулем, нужно добавить в нее пустой файл `__init__.py`



В примере выше мы создали модуль src, который состоит из трех файлов: необходимый файл `__init__.py`, `crud.py`, `app.py`

В таком случае, если нам понадобиться что-то импортировать из файла `crud.py` в файл `app.py`, можно воспользоваться следующим синтакисом

from src.crud import your_function

Но если мы попробуем запустить программу, то с большей вероятностью столкнемся с ошибкой

```
ModuleNotFoundError: No module named 'src'
```

Данная ошибка напрямую связана с переменной `PYTHONATH`, о которой будет рассказано дальше

> PYTHONPATH

Когда в прошлых шагах мы разносили код по папкам и импортировали нужные файлы в другие файлах, то столкнулись с распространенной проблемой, с которой сталкиваются начинающие разработчики.

Дело в том, что `python` при запуске программы ищет все импорты в специальных системных папках. В тех системных папках, которые были указаны при установке `python`. Если вы пользуетесь виртуальным окружением, то эти папки подменяются на папки виртуального окружения, но дело в том, что модуль, добавленный нами, не попадает в этот список. Т.е python попытается сделать импорт нужного модуля, но действие завершится ошибкой, так как модуль лежит не в системных папках.

Для того, чтобы устранить эту проблему, нужно расширить переменную `PYTHONPATH`. В этой переменной прописывается явно, где `python` стоит искать модули.

```
export PYTHONPATH=$PYTHONPATH:$PWD
```

Т.е мы обновляем значение `PYTHONPATH`: считываем значение, которое было до нас при помощи `$`, и затем через `:` добавляем значение новое переменной `PWD`. `PWD` - указывает на путь, где на данный момент вы находитесь в консоли.