АПИ автотесты на Python по шагам.

На данном семинаре, пошагово покроем тестами REST API  <https://restful-api.dev> на примере методов GET, POST, PUT, DELETE. Подход, описанный в семинаре, используется на реальных проектах.

Структура семинара

1. **Используемые библиотеки**
2. **Написание тестов:**
   1. требования к системе тестов
   2. структура проекта
   3. создание АПИ клиента
   4. написание первого теста
   5. код первого теста
   6. тесты на GET, POST, PUT, DELETE
3. **Заключение**

Используемые библиотеки

1. python 3.11.4
2. pytest 7.4.0 - написание тестов. Подробнее:<https://pytest-docs-ru.readthedocs.io/ru/latest/getting-started.html>
3. pydantic 2.3.0 - библиотека для валидации структуры ответа. Подробнее:<https://docs.pydantic.dev/>
4. httpx 0.24.1 - отправка запросов. Подробнее:<https://www.python-httpx.org/quickstart/>
5. тестовое АПИ - [https://restful-api.dev](https://restful-api.dev/)
6. среда разработки - PyCharm.

Написание тестов

Перед тем, как что-то писать, сформируем набор требований, которые мы будем соблюдать в нашей системе тестов, а именно:

1. идейность (каждый тест имеет четкую идею)
2. атомарность идеи (в тесте проверяется только одна идея)
3. независимость от других тестов (действия в одном тесте не влияют на другой, тесты могут идти в любом порядке)
4. гибкость относительно изменений в системе (тесты можно легко перенести на другую конфигурацию, например другой стенд или быстро изменить в случае изменений в тестируемом приложении)

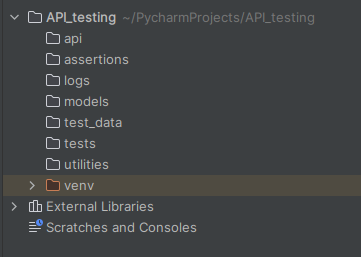
Соблюдение этих требований позволит нам писать тесты структурно и минимизировать лишний рефакторинг.

Структура проекта

Сформируем каркас проекта с папками, содержащими:

* **api** - классы для взаимодействия с api
* **logs** - логи запросов к серверу
* **assertions** - классы проверок для тестов
* **models** - модели pydantic для проверки схем ответа
* **test\_data** - эталонные файлы json для отправки запросов и проверки ответов
* **tests** - классы с тестами
* **utilities** - вспомогательные классы-утилиты (для работы с json и.т.д)

Структура будет выглядеть следующим образом:



При создании проекта также была инициализирована питоновская папкаvenv (подробнее <https://docs.python.org/3/library/venv.html>) для всех необходимых библиотек. Добавим в корень проекта файл requirements.txtсо следующей структурой:

httpx==0.24.1  
pytest==7.4.0  
pydantic==2.3.0  
python-dotenv==1.0.0

Установим перечисленные библиотеки командой pip install -r requirements.txt

Создание АПИ клиента

В проекте для отправки запросов использована библиотека httpx. Из фичей поддерживается сохранение сессии ([HTTP connection pooling](https://en.wikipedia.org/wiki/HTTP_persistent_connection)) и асинхронные запросы. Очень полезна в освоении. httpx предлагает использовать класс Client. Он позволяет открыть TCP соединение и отправить сколько угодно запросов в рамках него одного. Это экономит время при каждом запросе к серверу.

Наш АПИ клиент для удобства дополнительно будет логировать отправленные запросы по принципу: тип запроса, путь конечного эндпоинта.

В папку api добавляем файл api\_client.py

**import** os  
  
**from** httpx **import** Client, Response  
  
**from** utilities.logger\_utils **import** logger  
  
  
**class** ApiClient(Client):  
 """  
 Расширение стандартного клиента httpx.  
 """  
  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_(base\_url=f"https://{os.getenv('RESOURSE\_URL')}")  
  
 **def** request(self, method, url, \*\*kwargs) -> Response:  
 """  
 расширение логики метода httpx request с добавлением логирования типа запроса и его url,  
 логировать или нет задается в файле .env  
 :param method: метод, который мы используем (POST, GET и.т.д)  
 :param url: путь на домене, по которому отправляем запрос  
 """  
  
 **if** eval(os.getenv("USE\_LOGS")):  
 logger.info(f'{method} {url}')  
 **return** super().request(method, url, \*\*kwargs)

На каждый конечный эндпоинт добавим соответствующие методы в файл objects\_api.py

**from** api **import** routes  
  
  
**def** get\_objects(client, \*ids):  
 **return** client.get(routes.Routes.OBJECTS, params={'id': ids} **if** ids **else** **None**)  
  
  
**def** get\_object(client, obj\_id):  
 **return** client.get(routes.Routes.OBJECTS\_ITEM.format(obj\_id))  
  
  
**def** post\_object(client, \*\*kwargs):  
 **return** client.post(routes.Routes.OBJECTS, \*\*kwargs)  
  
  
**def** put\_object(client, obj\_id, \*\*kwargs):  
 **return** client.put(routes.Routes.OBJECTS\_ITEM.format(obj\_id), \*\*kwargs)  
  
  
**def** delete\_object(client, obj\_id):  
 **return** client.delete(routes.Routes.OBJECTS\_ITEM.format(obj\_id))

Также добавим класс с путями внутри сервера в файл routes.py

**from** enum **import** Enum  
  
  
**class** Routes(str, Enum):  
 OBJECTS = '/objects'  
 OBJECTS\_ITEM = '/objects/{}'  
  
 **def** \_\_str\_\_(self) -> str:  
 **return** self.value

Перед тем, как писать первый тест, добавим в корень проекта conftest.py, .env файл и pytest.ini. Они нужны для установки параметров логирования в нашем клиенте.

Файл conftest.py

**import** logging  
**import** os  
  
**from** dotenv **import** load\_dotenv  
  
**from** utilities.logger\_utils **import** logger  
  
  
**def** pytest\_configure(config):  
 # устанавливаем текущую директорию на корень проекта (это позволит прописывать относительные пути к файлам)  
 os.chdir(os.path.dirname(os.path.abspath(\_\_file\_\_)))  
  
 # загружаем переменные-параметры из файла /.env  
 load\_dotenv(dotenv\_path=".env")  
  
 # задаем параметры логгера  
 path = "logs/"  
 os.makedirs(os.path.dirname(path), exist\_ok=**True**)  
 file\_handler = logging.FileHandler(path + "/info.log", "w")  
 file\_handler.setLevel(logging.INFO)  
 file\_handler.setFormatter(logging.Formatter("%(lineno)d: %(asctime)s %(message)s"))  
  
 # создаем кастомный логгер  
 custom\_logger = logging.getLogger("custom\_loger")  
 custom\_logger.setLevel(logging.INFO)  
 custom\_logger.addHandler(file\_handler)  
  
  
**def** pytest\_runtest\_setup(item):  
 logger.info(f"{item.name}:")

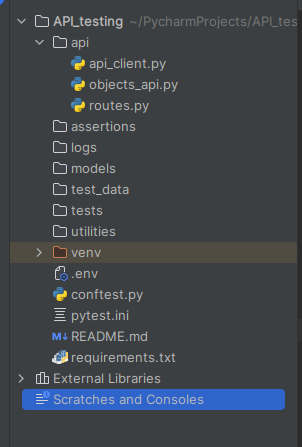
Файл.env

RESOURSE\_URL=api.restful-api.dev  
USE\_LOGS=**True**

Файлpytest.ini

[pytest]  
addopts =  
 --rootdir=.  
 -p no:logging

Все готово к написанию первого теста. По итогу структура проекта выглядит следующим образом:



Написание первого теста

Сформируем требования к структуре наших тестов. Каждый тест имеет обязательные и необязательные атрибуты (зависит от логики самого теста). Структурно это выглядит так:

1. Отправка запроса.
2. Проверка кода ответа (сравнивается ожидаемый и пришедший код).
3. Проверка схемы ответа (проверяется структура тела ответа и типы полей).
4. Проверка тела ответа и специфической логики

1 и 2 пункты мы исполняем обязательно т.к. любой запрос дает нам ожидаемый код ответа. Третий пункт мы будем проверять, если у ответа есть тело (json). 4 пункт подразумевает проверку корректности значений полей в теле ответа, а также специфической логики, которая соответствует идеи нашего теста. Этот пункт будет обязательным, если у ответа есть тело.

Для проверок 2, 3 и 4 пунктов добавим базовые методы assertion. В папку assertions добавим класс assertion\_base.py

**from** typing **import** Type  
  
**from** pydantic **import** BaseModel  
  
**from** utilities.files\_utils **import** read\_json\_test\_data, read\_json\_common\_response\_data  
**from** utilities.json\_utils **import** compare\_json\_left\_in\_right, remove\_ids  
  
  
**class** LogMsg:  
 """  
 Базовый класс для построение логов AssertionError. Конструирует сообщение в свое поле \_msg.  
 """  
  
 **def** \_\_init\_\_(self, where, response):  
 self.\_msg = ""  
 self.\_response = response  
 self.\_where = where  
  
 **def** add\_request\_url(self):  
 """  
 добавляет данные об отправленном на сервер запросе  
 """  
 self.\_msg += f"Содержимое отправляемого запроса (url, query params, тело):\n" \  
 f"\tURL: {self.\_response.request.url}\n"  
 self.\_msg += f"\tmethod: {self.\_response.request.method}\n"  
 self.\_msg += f"\theaders: {dict(self.\_response.request.headers)}\n"  
 **if** hasattr(self.\_response.request, 'params') **and** self.\_response.request.params:  
 self.\_msg += f"\tquery params: {self.\_response.request.params}\n"  
 **else**:  
 self.\_msg += f"\tquery params:\n"  
 **if** hasattr(self.\_response.request, 'content') **and** self.\_response.request.read():  
 self.\_msg += f"\tbody: {self.\_response.request.read()}\n"  
 **else**:  
 self.\_msg += f"\tbody:\n"  
 **return** self  
  
 **def** add\_response\_info(self):  
 """  
 добавляет информацию о содержимом тела ответа  
 """  
 self.\_msg += f"Тело ответа:\n\t{self.\_response.content}\n"  
 **return** self  
  
 **def** add\_error\_info(self, text):  
 **if** text:  
 self.\_msg += f"\n{text}\n"  
 **else**:  
 self.\_msg += "\n"  
 **return** self  
  
 **def** get\_message(self):  
 **return** self.\_msg  
  
  
**class** BodyLogMsg(LogMsg):  
 """  
 Добавляет в логи результаты проверок тела ответа.  
 """  
  
 **def** \_\_init\_\_(self, response):  
 super().\_\_init\_\_('В ТЕЛЕ ОТВЕТА', response)  
  
 **def** add\_compare\_result(self, diff):  
 """  
 добавляет информацию о результате сравнения полученного json с эталоном  
 :param diff: словарь с данными полей, которые после сравнения имеют разные значения  
 """  
 self.\_msg += f"{self.\_where} в json следующие поля не совпали с эталоном:\n"  
 **for** key, value **in** diff.items():  
 self.\_msg += f"ключ: {value['path']}\n\t\texpected: {value['expected']} \n\t\tactual: {value['actual']}\n"  
 **return** self  
  
  
**class** CodeLogMsg(LogMsg):  
 """  
 Добавляет в логи результаты проверки кода ответа.  
 """  
  
 **def** \_\_init\_\_(self, response):  
 super().\_\_init\_\_('В КОДЕ ОТВЕТА', response)  
  
 **def** add\_compare\_result(self, exp, act):  
 """  
 добавляет информацию об ожидаемом и полученной коде  
 :param exp: ожидаемый код  
 :param act: полученный код  
 """  
 self.\_msg += f"{self.\_where} \n\tожидался код: {exp}\n\tполученный код: {act}\n"  
 **return** self  
  
  
**class** BodyValueLogMsg(LogMsg):  
 **def** \_\_init\_\_(self, response):  
 super().\_\_init\_\_('В ТЕЛЕ ОТВЕТА', response)  
  
 **def** add\_compare\_result(self, exp, act):  
 """  
 добавляет информацию о сравнении значений в теле ответа  
 :param exp: ожидаемое значение  
 :param act: полученное значение  
 """  
 self.\_msg += f"\texptected: {exp}\n\tactual: {act}\n"  
 **return** self  
  
  
**def** assert\_status\_code(response, expected\_code):  
 """  
 сравнивает код ответа от сервера с ожидаемым  
 :param response: полученный от сервера ответ  
 :param expected\_code: ожидаемый код ответа  
 :raises AssertionError: если значения не совпали  
 """  
 **assert** expected\_code == response.status\_code, CodeLogMsg(response) \  
 .add\_compare\_result(expected\_code, response.status\_code) \  
 .add\_request\_url() \  
 .add\_response\_info() \  
 .get\_message()  
  
  
**def** assert\_schema(response, model: Type[BaseModel]):  
 """  
 проверяет тело ответа на соответствие его схеме механизмами pydantic  
 :param response: ответ от сервера  
 :param model: модель, по которой будет проверяться схема json  
 :raises ValidationError: если тело ответа не соответствует схеме  
 """  
 body = response.json()  
 **if** isinstance(body, list):  
 **for** item **in** body:  
 model.model\_validate(item, strict=**True**)  
 **else**:  
 model.model\_validate(body, strict=**True**)  
  
  
**def** assert\_left\_in\_right\_json(response, exp\_json, actual\_json):  
 """  
 проверяет, что все значения полей exp\_json равны значениям полей в actual\_json  
 :param response: полученный ответ от сервера  
 :param exp\_json: ожидаемый эталонный json  
 :param actual\_json: полученый json  
 :raises AssertionError: если в exp\_json есть поля со значениями, которые отличаются или которых нет в actual\_json  
 """  
 root = 'root:' **if** isinstance(actual\_json, list) **else** ''  
 compare\_res = compare\_json\_left\_in\_right(exp\_json, actual\_json, key=root, path=root)  
 **assert** **not** compare\_res, BodyLogMsg(response) \  
 .add\_compare\_result(compare\_res) \  
 .add\_request\_url() \  
 .add\_response\_info() \  
 .get\_message()  
  
  
**def** assert\_response\_body\_fields(request, response, exp\_obj=**None**, rmv\_ids=**True**):  
 """  
 проверяет ответ от сервера, сравнивая ожидаемый объект с полученным  
 :param request: стандартный объект request фреймворка pytest  
 :param response: ответ от сервера  
 :param exp\_obj: ожидаемый объект  
 :param rmv\_ids: флаг: значение True - удаляет id из тела ответа при проверке, False - не удаляет  
 """  
 exp\_json = read\_json\_test\_data(request) **if** exp\_obj **is** **None** **else** exp\_obj  
 act\_json = remove\_ids(response.json()) **if** rmv\_ids **else** response.json()  
 assert\_left\_in\_right\_json(response, exp\_json, act\_json)  
  
  
**def** assert\_response\_body\_value(response, exp, act, text=**None**):  
 """  
 проверяет ответ от сервера, сравнивая полученное значение с ожидаемым для тела запроса  
 :param response: ответ от сервера  
 :param exp: ожидаемое значение  
 :param act: полученное значение  
 :param text: дополнительный текст, который необходимо вывести при несовпадении exp и act  
 """  
 **assert** exp == act, BodyValueLogMsg(response) \  
 .add\_error\_info(text) \  
 .add\_compare\_result(exp, act) \  
 .add\_request\_url() \  
 .add\_response\_info() \  
 .get\_message()  
  
  
**def** assert\_empty\_list(response):  
 """  
 проверяет, что тело ответа содержит пустой список  
 :param response: ответ от сервера  
 """  
 assert\_left\_in\_right\_json(response, [], response.json())  
  
  
**def** assert\_bad\_request(request, response):  
 """  
 проверяет, что тело ответа содержит данные BAD REQUEST  
 :param request: стандартный объект request фреймворка pytest  
 :param response: ответ от сервера  
 """  
 assert\_response\_body\_fields(request, response, exp\_obj=read\_json\_common\_response\_data("bad\_request\_response"))  
  
  
**def** assert\_not\_found(request, response, obj\_id):  
 """  
 проверяет, что тело ответа содержит данные NOT FOUND  
 :param request: стандартный объект request фреймворка pytest  
 :param response: ответ от сервера  
 :param obj\_id: id объекта, который сервер не нашел  
 """  
 exp = read\_json\_common\_response\_data("not\_found\_obj\_response")  
 exp['error'] = exp['error'].format(obj\_id)  
 assert\_response\_body\_fields(request, response, exp\_obj=exp)  
  
  
**def** assert\_not\_exist(request, response, obj\_id):  
 """  
 проверяет, что тело ответа содержит данные NOT EXIST  
 :param request: стандартный объект request фреймворка pytest  
 :param response: ответ от сервера  
 :param obj\_id: id объекта, который сервер не нашел  
 """  
 exp = read\_json\_test\_data(request)  
 exp['error'] = exp['error'].format(obj\_id)  
 assert\_response\_body\_fields(request, response, exp\_obj=exp, rmv\_ids=**False**)

Базовые методы assertion начинаются сassert. Класс LogMsg и его производные используются для вывода понятных логов в stdout в случае несовпадения ожидаемого результата с фактическим.

Для проверки схемы ответа удобно использовать Pydantic. Это многофункциональная либа для сериализации и десериализации json со встроенными механизмами валидации. Для ее работы нам потребуются эталонные модели получаемых от сервера объектов. В папкуmodels добавим файл object\_models.py с моделями. Все они наследуются от BaseModel.

**from** typing **import** List  
  
**from** pydantic **import** BaseModel, Field  
  
  
**class** ObjectData(BaseModel):  
 year: int  
 price: float  
 cpu\_model: str = Field(alias="CPU model")  
 hard\_disk\_size: str = Field(alias="Hard disk size")  
  
  
**class** CustomObj(BaseModel):  
 name: str  
  
  
**class** CustomObjData(BaseModel):  
 bool: bool  
 int: int  
 float: float  
 string: str  
 array: List[str]  
 obj: CustomObj  
  
  
**class** ObjectOutSchema(BaseModel):  
 id: str  
 name: str  
 data: ObjectData  
  
  
**class** ObjectCreateOutSchema(BaseModel):  
 id: str  
 name: str | **None**  
 data: ObjectData | **None**  
 createdAt: str  
  
  
**class** CustomObjCreateOutSchema(BaseModel):  
 id: str  
 name: str  
 data: CustomObjData  
 createdAt: str  
  
  
**class** ObjectUpdateOutSchema(BaseModel):  
 id: str  
 name: str | **None**  
 data: ObjectData | **None**  
 updatedAt: str  
  
  
**class** CustomObjUpdateOutSchema(BaseModel):  
 id: str  
 name: str  
 data: CustomObjData  
 updatedAt: str

Также, чтобы сравнивать полученное тело ответа с эталонным, добавим вспомогательные классы-утилиты для работы с файлами и json.

Файл files\_utils.py

**import** json  
  
  
**def** get\_test\_data\_path():  
 **return** "test\_data"  
  
  
**def** get\_common\_response\_path():  
 **return** f"{get\_test\_data\_path()}/common/responses"  
  
  
**def** get\_common\_requests\_path():  
 **return** f"{get\_test\_data\_path()}/common/requests"  
  
  
**def** read\_json\_file\_data(path):  
 """  
 возвращает содержимое json файла в виде dict  
 :param path: путь до файла без расширения json  
 """  
 **with** open(f"{path}.json", "r") **as** f:  
 data = json.load(f)  
 **return** data  
  
  
**def** read\_json\_test\_data(request):  
 """  
 считывает данные для теста в формате json  
 :param request: стандартный объект request фреймворка pytest  
 :return: содержимое данных для теста из папки test\_data  
 """  
 **return** read\_json\_file\_data(f"{get\_test\_data\_path()}/{request.node.originalname}")  
  
  
**def** read\_json\_common\_response\_data(file\_name):  
 """  
 считывает данные для теста в формате json из общей папки  
 :param file\_name: имя файла без расширения json  
 :return: содержимое данных для теста из папки test\_data/common/responses  
 """  
 **return** read\_json\_file\_data(f"{get\_common\_response\_path()}/{file\_name}")  
  
  
**def** read\_json\_common\_request\_data(file\_name):  
 """  
 считывает данные для теста в формате json из общей папки  
 :param file\_name: имя файла без расширения json  
 :return: содержимое данных для теста из папки test\_data/common/requests  
 """  
 **return** read\_json\_file\_data(f"{get\_common\_requests\_path()}/{file\_name}")

Файл json\_utils.py

**def** remove\_ids(origin\_dict):  
 """  
 удаляет ключи из словаря, в которых есть id  
 :param origin\_dict: исходный словарь  
 :return: словарь с выпиленными id  
 """  
  
 **def** rmv\_ids(node):  
 remove\_keys = []  
 **if** isinstance(node, dict):  
 **for** key, value **in** node.items():  
 rmv\_ids(value)  
 **if** key == 'id':  
 remove\_keys.append(key)  
 **for** key **in** remove\_keys:  
 **del** node[key]  
 **elif** isinstance(node, list):  
 **for** item **in** node:  
 rmv\_ids(item)  
  
 res = origin\_dict.copy()  
 rmv\_ids(res)  
 **return** res  
  
  
**def** compare\_json\_left\_in\_right(json1, json2, key='', path=''):  
 """  
 сравнивает, что все значения ключей из json1 есть в json2, лишние ключи из левого json - игнорируются  
 :param json1: эталонный словаро  
 :param json2: словарь, с которым идет сравнение  
 :param key: корневое имя ключа  
 :param path: путь до ключа, в котором произошло несовпадение значений  
 :return: если в правом словаре есть несовпадения значений со значений из левого словаря, возвращается словарь  
 формата {"ключ\_в\_котором\_произошло\_различие": {"expected": value, "actual": value, "path": полный\_путь\_до\_ключа}}  
 """  
 diff\_dict = {}  
 **if** isinstance(json1, dict) **and** isinstance(json2, dict):  
 **for** key **in** json1:  
 **if** key **not** **in** json2:  
 diff\_dict[key] = {"expected": json1[key], "actual": "key undefined", "path": f"{path}{key}"}  
 **continue**  
 diff\_dict.update(compare\_json\_left\_in\_right(json1[key], json2[key], key, f"{path}{key}:"))  
 **elif** json1 != json2:  
 diff\_dict[key] = {"expected": json1, "actual": json2, "path": path[:-1]}  
 **return** diff\_dict

С методами проверок закончили.

Приступим к написанию тестов. Начнем с написания тестов на GET методы. Для начала сформируем шаблон с принципами и действиями, которыми будем руководствоваться при написании чек-листов.

Есть общие действия для каждого типа метода, исходя из его типа (GET, POST и.т.д). Для каждого типа метода нам надо выполнить 2 пункта: проверить, что метод выполняет свою суть в зависимости от типа и проверить его бизнес-логику. Например, суть метода GET - возвращать данные с сервера. Соответственно на каждый GET метод мы должны множеством тестов убедиться, что метод правильно возвращает значения с сервера, и проверить специфическую логику, заложенную в этот метод.

|  |  |
| --- | --- |
| Метод | Пункты, которые необходимо реализовать |
| GET | 1 проверка корректности получения объекта: - в тестах заранее заготовлен файл с данными для проверки на эталонность без айдишников (файл одинаков при каждом запуске), в базе заранее заготовлен объект, который никогда не изменяется - делаем get запрос с нужным параметром - делаем полное сравнение ответа с эталонным файлом  2 Проверка специфической логики |

Распишем чек-листы, распределив их на позитивные и негативные. В данном АПИ метод GET /objects просто возвращает список объектов с сервера, соответственно мы должны проверить, что объекты возвращаются при валидных и невалидных параметрах.

|  |  |
| --- | --- |
| **Позитивные** | **Негативные** |
| запрос с параметрами по-умолчанию  запрос с 1 сущ. айдишником  запрос с 2 сущ. айдишниками | запрос с несущ. айдишником  запрос с невалидным айдишником |

Реализуем первый тест

Файл test\_objects.py

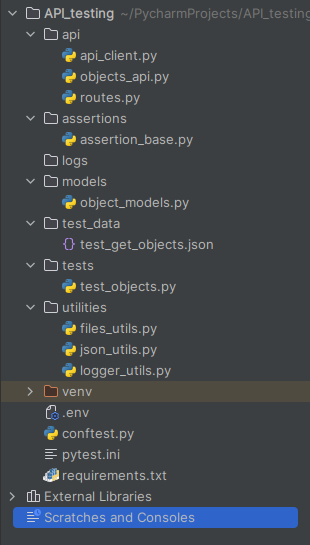
**from** http **import** HTTPStatus  
  
**import** pytest  
**from** assertions.objects\_assertion **import** should\_be\_posted\_success, should\_be\_updated\_success, should\_be\_deleted\_success, \  
 should\_be\_valid\_objects\_response  
  
**from** api.api\_client **import** ApiClient  
**from** api.objects\_api **import** get\_objects, get\_object, post\_object, put\_object, delete\_object  
**from** assertions.assertion\_base **import** assert\_status\_code, assert\_response\_body\_fields, assert\_bad\_request, \  
 assert\_not\_found, assert\_empty\_list, assert\_schema, assert\_not\_exist  
**from** models.object\_models **import** ObjectOutSchema, ObjectCreateOutSchema, CustomObjCreateOutSchema, \  
 ObjectUpdateOutSchema, CustomObjUpdateOutSchema  
**from** utilities.files\_utils **import** read\_json\_test\_data, read\_json\_common\_request\_data  
  
  
**class** TestObjects:  
 """  
 Тесты /objects  
 """  
  
 @pytest.fixture(scope='class')  
 **def** client(self):  
 **return** ApiClient()  
  
 **def** test\_get\_objects(self, client, request):  
 """  
 получение заранее заготовленных объектов из базы с параметрами по-умолчанию,  
 GET /objects  
 """  
 # получаем объекты из базы  
 response = get\_objects(client)  
  
 # убеждаемся, что в ответ пришли объекты, которые мы ожидаем  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_response\_body\_fields(request, response)

В тесте происходит следующее: мы делаем запрос, проверяем, что код ответа 200, а затем убеждаемся, что полученный объект в ответе соответствует ожидаемому (проверка тела ответа), делая прямое сравнение с проверочным файлом. При сравнении, чтобы не нарушать 4 сформированное требование к тестирующей системе (гибкость), выпиливаем все id из ответа. id — это внутреннее состояние объекта. Выпиливаем мы их, чтобы при каком-либо обновлении состояния БД наши тесты не посыпались, если логика метода и структура объекта при таком обновлении не изменились. В проверочном файле их также нет. Проверочный файл выглядит следующим образом:

Файл test\_get\_objects.json

[  
 {  
 "name": "Google Pixel 6 Pro",  
 "data": {  
 "color": "Cloudy White",  
 "capacity": "128 GB"  
 }  
 },  
 {  
 "name": "Apple iPhone 12 Mini, 256GB, Blue",  
 "data": null  
 },  
 {  
 "name": "Apple iPhone 12 Pro Max",  
 "data": {  
 "color": "Cloudy White",  
 "capacity GB": 512  
 }  
 },  
 {  
 "name": "Apple iPhone 11, 64GB",  
 "data": {  
 "price": 389.99,  
 "color": "Purple"  
 }  
 },  
 {  
 "name": "Samsung Galaxy Z Fold2",  
 "data": {  
 "price": 689.99,  
 "color": "Brown"  
 }  
 },  
 {  
 "name": "Apple AirPods",  
 "data": {  
 "generation": "3rd",  
 "price": 120  
 }  
 },  
 {  
 "name": "Apple MacBook Pro 16",  
 "data": {  
 "year": 2019,  
 "price": 1849.99,  
 "CPU model": "Intel Core i9",  
 "Hard disk size": "1 TB"  
 }  
 },  
 {  
 "name": "Apple Watch Series 8",  
 "data": {  
 "Strap Colour": "Elderberry",  
 "Case Size": "41mm"  
 }  
 },  
 {  
 "name": "Beats Studio3 Wireless",  
 "data": {  
 "Color": "Red",  
 "Description": "High-performance wireless noise cancelling headphones"  
 }  
 },  
 {  
 "name": "Apple iPad Mini 5th Gen",  
 "data": {  
 "Capacity": "64 GB",  
 "Screen size": 7.9  
 }  
 },  
 {  
 "name": "Apple iPad Mini 5th Gen",  
 "data": {  
 "Capacity": "254 GB",  
 "Screen size": 7.9  
 }  
 },  
 {  
 "name": "Apple iPad Air",  
 "data": {  
 "Generation": "4th",  
 "Price": "419.99",  
 "Capacity": "64 GB"  
 }  
 },  
 {  
 "name": "Apple iPad Air",  
 "data": {  
 "Generation": "4th",  
 "Price": "519.99",  
 "Capacity": "256 GB"  
 }  
 }  
]

По итогу структура проекта приобретает следующий вид:



Расширим тестовое покрытие метода GET /objects по нашим чек-листам. В данном случае кейсы “запрос с 2 сущ. айдишниками” и “запрос с 1 сущ. айдишниками” требуют специфических проверок. Для начала добавим класс, который будет хранить такие проверки objects\_assertion.py. Методы специфических проверок начинаются с префикс аshould\_be.

Файл objects\_assertion.py

**from** http **import** HTTPStatus  
  
**from** api.objects\_api **import** get\_object  
**from** assertions.assertion\_base **import** assert\_response\_body\_fields, assert\_status\_code, assert\_response\_body\_value  
**from** utilities.files\_utils **import** read\_json\_test\_data  
  
  
**def** should\_be\_valid\_objects\_response(request, response, param):  
 # убеждаемся, что в ответе столько объектов, сколько мы ожидаем  
 exp = read\_json\_test\_data(request)[param['index']]  
 exp\_len, act\_len = len(exp), len(response.json())  
 assert\_response\_body\_value(response, exp\_len, act\_len,  
 text="ОЖИДАЕМОЕ КОЛИЧЕСТВО ОБЪЕКТОВ НЕ СОВПАЛО С ФАКТИЧЕСКИМ")  
  
 # убеждаемся в корректности значений полей полученных объектов  
 assert\_response\_body\_fields(request, response, exp)

Добавим тесты в файлtest\_objects.py

**from** http **import** HTTPStatus  
  
**import** pytest  
**from** assertions.objects\_assertion **import** should\_be\_posted\_success, should\_be\_updated\_success, should\_be\_deleted\_success, \  
 should\_be\_valid\_objects\_response  
  
**from** api.api\_client **import** ApiClient  
**from** api.objects\_api **import** get\_objects, get\_object, post\_object, put\_object, delete\_object  
**from** assertions.assertion\_base **import** assert\_status\_code, assert\_response\_body\_fields, assert\_bad\_request, \  
 assert\_not\_found, assert\_empty\_list, assert\_schema, assert\_not\_exist  
**from** models.object\_models **import** ObjectOutSchema, ObjectCreateOutSchema, CustomObjCreateOutSchema, \  
 ObjectUpdateOutSchema, CustomObjUpdateOutSchema  
**from** utilities.files\_utils **import** read\_json\_test\_data, read\_json\_common\_request\_data  
  
  
**class** TestObjects:  
 """  
 Тесты /objects  
 """  
  
 @pytest.fixture(scope='class')  
 **def** client(self):  
 **return** ApiClient()  
  
 **def** test\_get\_objects(self, client, request):  
 """  
 получение заранее заготовленных объектов из базы с параметрами по-умолчанию,  
 GET /objects  
 """  
 # получаем объекты из базы  
 response = get\_objects(client)  
  
 # убеждаемся, что в ответ пришли объекты, которые мы ожидаем  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_response\_body\_fields(request, response)  
  
 @pytest.mark.parametrize("param", [{"index": 0, "ids": [1]}, {"index": 1, "ids": [1, 2]}])  
 **def** test\_get\_objects\_id\_param(self, client, request, param):  
 """  
 получение заранее заготовленных объектов из базы с параметром ids,  
 GET /objects  
 """  
 # получаем массив объектов с определенными айдишниками  
 response = get\_objects(client, \*param['ids'])  
  
 # убеждаемся, что в ответ пришли именно те объекты, id которых мы запросили  
 assert\_status\_code(response, HTTPStatus.OK)  
 should\_be\_valid\_objects\_response(request, response, param)  
  
 **def** test\_get\_objects\_not\_exist\_id(self, client):  
 """  
 попытка получить из базы объект с несуществующим id,  
 GET /objects  
 """  
 # пытаемся получить объект, несуществующий в системе  
 response = get\_objects(client, 8523697415)  
  
 # убеждаемся, что в ответ пришел пустой список  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_empty\_list(response)  
  
 **def** test\_get\_objects\_invalid\_id(self, client):  
 """  
 попытка получить из базы объект с невалидным по типу id,  
 GET /objects  
 """  
 # пытаемся получить объект, отправив невалидный по типу параметр ids  
 response = get\_objects(client, "kjdsf23321")  
  
 # убеждаемся, что в ответ пришел пустой список  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_empty\_list(response)

По итогу мы 5 тестами проверили, что метод GET /objects действительно возвращает корректные объекты из базы при разных параметрах и выдает ошибки при неверных id.

Распишем по такому же принципу чек-лист для метода GET /objects/{id}

|  |  |
| --- | --- |
| **Позитивные** | **Негативные** |
| запрос существующего объекта | запрос несуществующего объекта |

В файл test\_objects.py добавим

**def** test\_get\_object(self, client, request):  
 """  
 получение заранее заготовленного объекта из базы,  
 GET /objects/{id}  
 """  
 # получаем единичный объект с сервера  
 response = get\_object(client, 7)  
  
 # убеждаемся, что получен именно тот объект, который мы запросили  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, ObjectOutSchema)  
 assert\_response\_body\_fields(request, response)  
  
**def** test\_get\_object\_not\_exist(self, client, request):  
 """  
 попытка получить из базы единичный объект с несуществующим id,  
 GET /objects/{id}  
 """  
 # запрашиваем единичный объект с сервера с несуществующим id  
 response = get\_object(client, 1593576458)  
  
 # убеждаемся, что сервер вернул NOT FOUND ответ  
 assert\_status\_code(response, HTTPStatus.NOT\_FOUND)  
 assert\_not\_exist(request, response, 1593576458)

Как видно в тесте test\_get\_object добавилась проверка схемы ответа (метод assert\_schema). Pydantic проверит, что полученный объект содержит все необходимые поля и проверит, что они строго того типа, который мы ожидаем. В случае ошибки, он выведет нам поля, которые не соответствуют схеме по типу или не найдены.

Приступим к методу POST.

Cуть метода POST - сохранять данные на сервере. Как правило, это приводит к порождению нового объекта. Соответственно мы должны множеством тестов убедиться, что метод правильно сохраняет значения на сервере, и проверить специфическую логику, заложенную в этот метод.

|  |  |
| --- | --- |
| Метод | Пункты, которые необходимо реализовать |
| POST | 1 проверка корректности сохранения объекта на сервере: - создаем объект - запоминаем отправляемый объект - отправляем объект - убеждаемся, что объект в ответе корректен - убеждаемся, что в базе этот объект корректно сохранен  2 Проверка специфической логики |

В данном АПИ метод POST /objects сохраняет объект, у которого всегда есть 2 поля nameиdata**.**Сохранение данных всегда происходит в эти 2 поля, если они не заполнены, сервер установит их в null. Соответственно наша задача проверить, что данные в эти поля сохраняются в соответствии с заложенной логикой, и сервер не позволяет сохранить в базу невалидный объект. Распишем чек-лист.

|  |  |
| --- | --- |
| **Позитивные** | **Негативные** |
| запись пустого тела в базу  запись name и data с полями всех типов в базу | отправка невалидного json |

В файл test\_objects.py добавим

**def** test\_post\_object\_empty\_body(self, client, request):  
 """  
 запись объекта в базу с пустым телом,  
 POST /objects  
 """  
 # записываем объект в базу с пустым телом  
 response = post\_object(client, json={})  
  
 # убеждаемся, что объект успешно записан в базу  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, ObjectCreateOutSchema)  
 should\_be\_posted\_success(request, client, response, exp\_obj={"data": **None**, "name": **None**})  
  
**def** test\_post\_object\_with\_full\_body(self, client, request):  
 """  
 запись объекта в базу полностью заполненным телом,  
 POST /objects  
 """  
 # записываем объект в базу со всеми заполненными полями  
 exp\_obj = read\_json\_common\_request\_data("valid\_post\_object")  
 response = post\_object(client, json=exp\_obj)  
  
 # убеждаемся, что объект успешно записан в базу  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, CustomObjCreateOutSchema)  
 should\_be\_posted\_success(request, client, response, exp\_obj)  
  
**def** test\_post\_object\_send\_invalid\_json(self, client, request):  
 """  
 попытка записать в базу невалидный json,  
 POST /objects  
 """  
 # отправляем запрос на запись объекта в базу с невалидным json в теле  
 response = post\_object(client, content='{"name",}', headers={"Content-Type": "application/json"})  
  
 # убеждаемся, что сервер дал BAD REQUEST ответ  
 assert\_status\_code(response, HTTPStatus.BAD\_REQUEST)  
 assert\_bad\_request(request, response)

По итогу 3 тестами мы убедились, что POST /objects корректно записывает валидные объекты в базу, порождает объекты с пустыми полями name и data и, если мы их не отправили, не позволяет записать невалидные объекты.

Приступим к методу PUT. Cуть метода PUT - обновлять данные на сервере. Как правило это полное обновление объекта. Соответственно мы должны убедиться, что метод правильно обновляет значения на сервере, и проверить специфическую логику.

|  |  |
| --- | --- |
| Метод | Пункты, которые необходимо реализовать |
| PUT | 1 проверка корректности обновления объекта в базе: - получаем id обновляемого объекта - создаем объект с другими значениями - запоминаем состояние объекта до обновления его в базе - обновляем в базе объект - убеждаемся, что объект в ответе корректно обновлен - убеждаемся, что в базе этот объект корректно сохранен  2 Проверка специфической логики |

В данном АПИ метод PUT /objects/{id} обновляет 2 поля объекта: name и data**.**При записи обновления в базулогика такая же как у метода POST. Распишем чек-лист.

|  |  |
| --- | --- |
| **Позитивные** | **Негативные** |
| обновление name и data с полями всех типов  обновление на пустой | обновление несущ. объекта  на невалидный объект |

В файл test\_objects.py добавим

**def** test\_put\_object\_with\_empty\_body(self, client, request):  
 """  
 обновление объекта в базе на пустой объект,  
 PUT /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 post\_obj = read\_json\_common\_request\_data("valid\_post\_object")  
 response = post\_object(client, json=post\_obj)  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # обновляем этот объект на пустой объект  
 exp\_json = {"id": response.json()['id'], "name": **None**, "data": **None**}  
 response = put\_object(client, exp\_json['id'], json={})  
  
 # убеждаемся, что объект был успешно обновлен  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, ObjectUpdateOutSchema)  
 should\_be\_updated\_success(request, client, response, exp\_json)  
  
**def** test\_put\_object\_with\_full\_body(self, client, request):  
 """  
 обновление всех полей объекта в базе,  
 PUT /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 post\_obj = read\_json\_common\_request\_data("valid\_post\_object")  
 response = post\_object(client, json=post\_obj)  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # обновляем значения всех полей этого объекта на новые  
 put\_obj = read\_json\_test\_data(request)  
 put\_obj\_id = response.json()['id']  
 response = put\_object(client, put\_obj\_id, json=put\_obj)  
  
 # убеждаемся, что объект был успешно обновлен  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, CustomObjUpdateOutSchema)  
 put\_obj['id'] = put\_obj\_id  
 should\_be\_updated\_success(request, client, response, put\_obj)  
  
**def** test\_put\_object\_send\_invalid\_json(self, client, request):  
 """  
 попытка обновить объект, отправив невалидный json,  
 PUT /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 response = post\_object(client, json=read\_json\_common\_request\_data("valid\_post\_object"))  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # пытаемся обновить этот объект, отправив невалидный json  
 response = put\_object(client, response.json()['id'], content='{"name",}',  
 headers={"Content-Type": "application/json"})  
  
 # убеждаемся, что сервер дал BAD REQUEST ответ  
 assert\_status\_code(response, HTTPStatus.BAD\_REQUEST)  
 assert\_bad\_request(request, response)  
  
**def** test\_put\_object\_update\_non\_exist\_obj(self, client, request):  
 """  
 попытка обновить несуществующий объект,  
 PUT /objects/{id}  
 """  
 # пытаемся обновить несуществующие объект  
 obj\_id = "ff8081818a194cb8018a79e7545545ac"  
 response = put\_object(client, obj\_id, json={})  
  
 # убеждаемся, что сервер дал NOT FOUND ответ  
 assert\_status\_code(response, HTTPStatus.NOT\_FOUND)  
 assert\_not\_found(request, response, obj\_id)

По итогу 4 тестами мы убедились, что PUT /objects/{id} обновляет все поля именно того объекта, id которого мы отправили, и не позволяет обновить состояние объекта на невалидное.

Cуть метода DELETE - удалять данные с сервера.

|  |  |
| --- | --- |
| Метод | Пункты, которые необходимо реализовать |
| DELETE | 1 проверка удаления объекта с сервера: - получаем id удаляемого объекта - удаляем его - убеждаемся, что он удален (404 ошибка)  2 Проверка специфической логики |

Распишем чек-лист

|  |  |
| --- | --- |
| **Позитивные** | **Негативные** |
| удаление существующего объекта | удаление несуществующего объекта |

В файл test\_objects.py добавим

**def** test\_delete\_exist\_object(self, client, request):  
 """  
 удаление сущестующего объекта,  
 DELETE /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 response = post\_object(client, json=read\_json\_common\_request\_data("valid\_post\_object"))  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # удаляем этот объект  
 obj\_id = response.json()['id']  
 response = delete\_object(client, obj\_id)  
  
 # убеждаемся, что объект удален  
 assert\_status\_code(response, HTTPStatus.OK)  
 should\_be\_deleted\_success(request, response, obj\_id)  
  
**def** test\_delete\_not\_exist\_object(self, client, request):  
 """  
 удаление несущестующего объекта,  
 DELETE /objects/{id}  
 """  
 # пытаемся удалить несуществующий объект  
 obj\_id = "ff8081818a194cb8018a79e7545545ac"  
 response = delete\_object(client, obj\_id)  
  
 # убеждаемся, что сервер дал NOT FOUND ответ  
 assert\_status\_code(response, HTTPStatus.NOT\_FOUND)  
 assert\_not\_exist(request, response, obj\_id)

На каждом шаге в папку test\_data также добавляются соответствующие файлы для проверки ответов от сервера. По итогу получилось 16 тестов. Файл со всеми тестами выглядит следующим образом.

**from** http **import** HTTPStatus  
  
**import** pytest  
  
**from** api.api\_client **import** ApiClient  
**from** api.objects\_api **import** get\_objects, get\_object, post\_object, put\_object, delete\_object  
**from** assertions.assertion\_base **import** assert\_status\_code, assert\_response\_body\_fields, assert\_bad\_request, \  
 assert\_not\_found, assert\_empty\_list, assert\_schema, assert\_not\_exist  
**from** assertions.objects\_assertion **import** should\_be\_posted\_success, should\_be\_updated\_success, should\_be\_deleted\_success, \  
 should\_be\_valid\_objects\_response  
**from** models.object\_models **import** ObjectOutSchema, ObjectCreateOutSchema, CustomObjCreateOutSchema, \  
 ObjectUpdateOutSchema, CustomObjUpdateOutSchema  
**from** utilities.files\_utils **import** read\_json\_test\_data, read\_json\_common\_request\_data  
  
  
**class** TestObjects:  
 """  
 Тесты /objects  
 """  
  
 @pytest.fixture(scope='class')  
 **def** client(self):  
 **return** ApiClient()  
  
 **def** test\_get\_objects(self, client, request):  
 """  
 получение заранее заготовленных объектов из базы с параметрами по-умолчанию,  
 GET /objects  
 """  
 # получаем объекты из базы  
 response = get\_objects(client)  
  
 # убеждаемся, что в ответ пришли объекты, которые мы ожидаем  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_response\_body\_fields(request, response)  
  
 @pytest.mark.parametrize("param", [{"index": 0, "ids": [1]}, {"index": 1, "ids": [1, 2]}])  
 **def** test\_get\_objects\_id\_param(self, client, request, param):  
 """  
 получение заранее заготовленных объектов из базы с параметром ids,  
 GET /objects  
 """  
 # получаем массив объектов с определенными айдишниками  
 response = get\_objects(client, \*param['ids'])  
  
 # убеждаемся, что в ответ пришли именно те объекты, id которых мы запросили  
 assert\_status\_code(response, HTTPStatus.OK)  
 should\_be\_valid\_objects\_response(request, response, param)  
  
 **def** test\_get\_objects\_not\_exist\_id(self, client):  
 """  
 попытка получить из базы объект с несуществующим id,  
 GET /objects  
 """  
 # пытаемся получить объект, несуществующий в системе  
 response = get\_objects(client, 8523697415)  
  
 # убеждаемся, что в ответ пришел пустой список  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_empty\_list(response)  
  
 **def** test\_get\_objects\_invalid\_id(self, client):  
 """  
 попытка получить из базы объект с невалидным по типу id,  
 GET /objects  
 """  
 # пытаемся получить объект, отправив невалидный по типу параметр ids  
 response = get\_objects(client, "kjdsf23321")  
  
 # убеждаемся, что в ответ пришел пустой список  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_empty\_list(response)  
  
 **def** test\_get\_object(self, client, request):  
 """  
 получение заранее заготовленного объекта из базы,  
 GET /objects/{id}  
 """  
 # получаем единичный объект с сервера  
 response = get\_object(client, 7)  
  
 # убеждаемся, что получен именно тот объект, который мы запросили  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, ObjectOutSchema)  
 assert\_response\_body\_fields(request, response)  
  
 **def** test\_get\_object\_not\_exist(self, client, request):  
 """  
 попытка получить из базы единичный объект с несуществующим id,  
 GET /objects/{id}  
 """  
 # запрашиваем единичный объект с сервера с несуществующим id  
 response = get\_object(client, 1593576458)  
  
 # убеждаемся, что сервер вернул NOT FOUND ответ  
 assert\_status\_code(response, HTTPStatus.NOT\_FOUND)  
 assert\_not\_exist(request, response, 1593576458)  
  
 **def** test\_post\_object\_empty\_body(self, client, request):  
 """  
 запись объекта в базу с пустым телом,  
 POST /objects  
 """  
 # записываем объект в базу с пустым телом  
 response = post\_object(client, json={})  
  
 # убеждаемся, что объект успешно записан в базу  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, ObjectCreateOutSchema)  
 should\_be\_posted\_success(request, client, response, exp\_obj={"data": **None**, "name": **None**})  
  
 **def** test\_post\_object\_with\_full\_body(self, client, request):  
 """  
 запись объекта в базу полностью заполненным телом,  
 POST /objects  
 """  
 # записываем объект в базу со всеми заполненными полями  
 exp\_obj = read\_json\_common\_request\_data("valid\_post\_object")  
 response = post\_object(client, json=exp\_obj)  
  
 # убеждаемся, что объект успешно записан в базу  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, CustomObjCreateOutSchema)  
 should\_be\_posted\_success(request, client, response, exp\_obj)  
  
 **def** test\_post\_object\_send\_invalid\_json(self, client, request):  
 """  
 попытка записать в базу невалидный json,  
 POST /objects  
 """  
 # отправляем запрос на запись объекта в базу с невалидным json в теле  
 response = post\_object(client, content='{"name",}', headers={"Content-Type": "application/json"})  
  
 # убеждаемся, что сервер дал BAD REQUEST ответ  
 assert\_status\_code(response, HTTPStatus.BAD\_REQUEST)  
 assert\_bad\_request(request, response)  
  
 **def** test\_put\_object\_with\_empty\_body(self, client, request):  
 """  
 обновление объекта в базе на пустой объект,  
 PUT /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 post\_obj = read\_json\_common\_request\_data("valid\_post\_object")  
 response = post\_object(client, json=post\_obj)  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # обновляем этот объект на пустой объект  
 exp\_json = {"id": response.json()['id'], "name": **None**, "data": **None**}  
 response = put\_object(client, exp\_json['id'], json={})  
  
 # убеждаемся, что объект был успешно обновлен  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, ObjectUpdateOutSchema)  
 should\_be\_updated\_success(request, client, response, exp\_json)  
  
 **def** test\_put\_object\_with\_full\_body(self, client, request):  
 """  
 обновление всех полей объекта в базе,  
 PUT /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 post\_obj = read\_json\_common\_request\_data("valid\_post\_object")  
 response = post\_object(client, json=post\_obj)  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # обновляем значения всех полей этого объекта на новые  
 put\_obj = read\_json\_test\_data(request)  
 put\_obj\_id = response.json()['id']  
 response = put\_object(client, put\_obj\_id, json=put\_obj)  
  
 # убеждаемся, что объект был успешно обновлен  
 assert\_status\_code(response, HTTPStatus.OK)  
 assert\_schema(response, CustomObjUpdateOutSchema)  
 put\_obj['id'] = put\_obj\_id  
 should\_be\_updated\_success(request, client, response, put\_obj)  
  
 **def** test\_put\_object\_send\_invalid\_json(self, client, request):  
 """  
 попытка обновить объект, отправив невалидный json,  
 PUT /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 response = post\_object(client, json=read\_json\_common\_request\_data("valid\_post\_object"))  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # пытаемся обновить этот объект, отправив невалидный json  
 response = put\_object(client, response.json()['id'], content='{"name",}',  
 headers={"Content-Type": "application/json"})  
  
 # убеждаемся, что сервер дал BAD REQUEST ответ  
 assert\_status\_code(response, HTTPStatus.BAD\_REQUEST)  
 assert\_bad\_request(request, response)  
  
 **def** test\_put\_object\_update\_non\_exist\_obj(self, client, request):  
 """  
 попытка обновить несуществующий объект,  
 PUT /objects/{id}  
 """  
 # пытаемся обновить несуществующие объект  
 obj\_id = "ff8081818a194cb8018a79e7545545ac"  
 response = put\_object(client, obj\_id, json={})  
  
 # убеждаемся, что сервер дал NOT FOUND ответ  
 assert\_status\_code(response, HTTPStatus.NOT\_FOUND)  
 assert\_not\_found(request, response, obj\_id)  
  
 **def** test\_delete\_exist\_object(self, client, request):  
 """  
 удаление сущестующего объекта,  
 DELETE /objects/{id}  
 """  
 # записываем объект в базу со всеми заполненными полями  
 response = post\_object(client, json=read\_json\_common\_request\_data("valid\_post\_object"))  
 assert\_status\_code(response, HTTPStatus.OK)  
  
 # удаляем этот объект  
 obj\_id = response.json()['id']  
 response = delete\_object(client, obj\_id)  
  
 # убеждаемся, что объект удален  
 assert\_status\_code(response, HTTPStatus.OK)  
 should\_be\_deleted\_success(request, response, obj\_id)  
  
 **def** test\_delete\_not\_exist\_object(self, client, request):  
 """  
 удаление несущестующего объекта,  
 DELETE /objects/{id}  
 """  
 # пытаемся удалить несуществующий объект  
 obj\_id = "ff8081818a194cb8018a79e7545545ac"  
 response = delete\_object(client, obj\_id)  
  
 # убеждаемся, что сервер дал NOT FOUND ответ  
 assert\_status\_code(response, HTTPStatus.NOT\_FOUND)  
 assert\_not\_exist(request, response, obj\_id)

Подводя итог: 16 тестами мы произвели проверку работы 5 АПИ методов. Каждый тест соблюдает 4 сформированных требования. Тесты атомарны и независимы, а значит мы можем без проблем изменить 1 тест, не нарушая логику другого теста. Тесты относительно гибки. Например проверочные файлы можно группировать при одинаковых или схожих ответах, как в методе assert\_bad\_request, делая отношения множество тестов к 1 файлу. Тогда в случае изменения ответа на bad request сервером нам достаточно поправить 1 файл, не меняя код самих тестов.

Сам проект вы можете взять из архива master\_API\_testing. После загрузки проекта в терминале переходим в папку API\_testing, устанавливаем requirements.txt и запускаем тесты командой pytest из корня проекта