RUDN university

ТЕСТИРОВАНИЕ **ПО**

ТИПЫ, ПРИНЦИПЫ, МЕТОДОЛОГИИ, PYTEST, ПОКРЫТИЕ КОДА

УСТАНОВКА НЕОБХОДИМЫХ БИБЛИОТЕК

Для дальнейшей работы необходимо установить библиотеки

```
pytest==8.3.3
pytest-cov==5.0.0
responses==0.25.3
pytest-mock==3.14.0
freezegun==1.5.1
```

УСТАНОВКА НЕОБХОДИМЫХ БИБЛИОТЕК

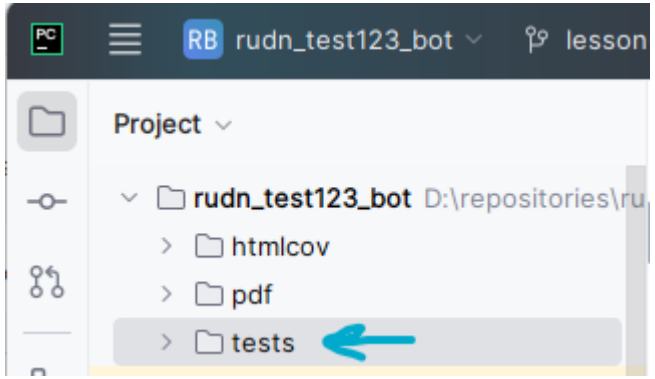
```python
import os
import shutil
from pathlib import Path

ROOT = Path(__file__).resolve().parent

# Удаляем файл .coverage
for p in ROOT.glob(".coverage"):
    try:
        p.unlink()
    except FileNotFoundError:
        pass

# Удаляем htmlcov
htmlcov = ROOT / "htmlcov"
if htmlcov.is_dir():
    shutil.rmtree(htmlcov)
```

1. Создать в корне проекта файл **cleanup_coverage.py**

с содержимым на слайде слева

В скрипте мы удаляем файл **.coverage** и каталог **htmlcov**.

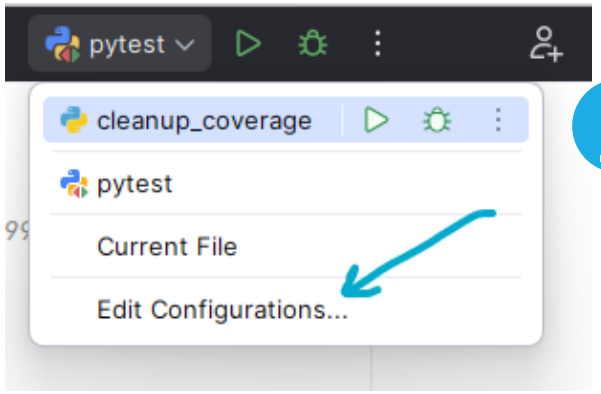
Это позволит при изменении скриптов автомачески зачищать эти сущности, чтобы создавались новые.



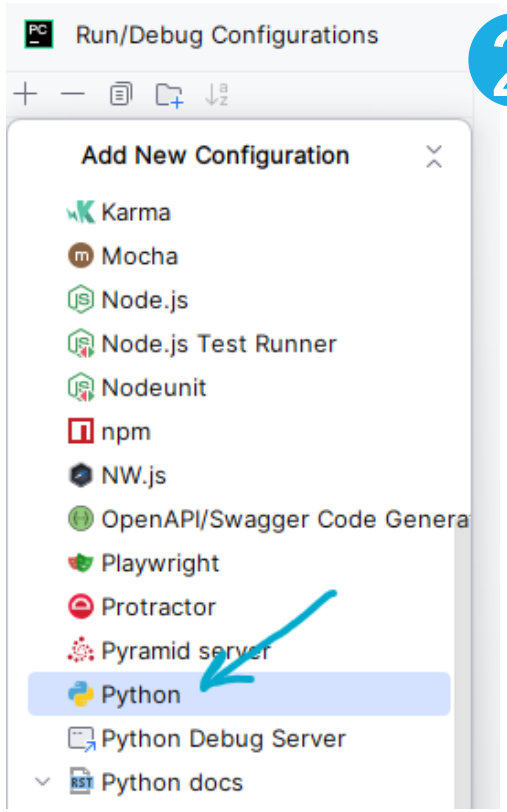
2. Создать в корне проекта директорию **tests**

В данной директории мы будем создавать наши тесты

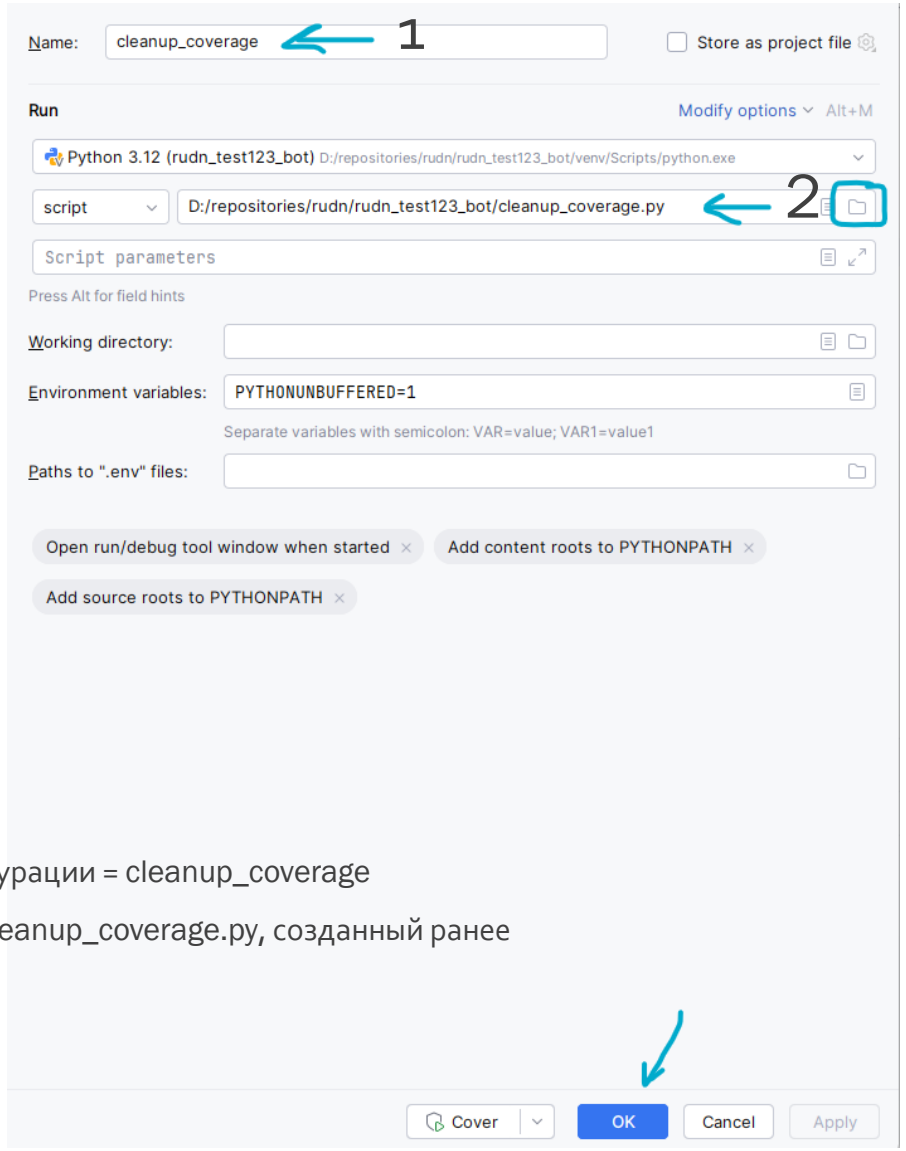
PYCHARM: СОЗДАНИЕ КОНФИГУРАЦИИ 1



1 Перейти в режим редактирования



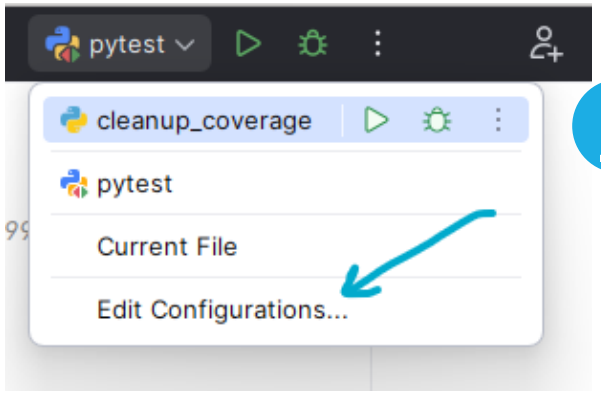
2 Нажать + и выбрать Python



3

1. Задаем имя конфигурации = cleanup_coverage
2. Выбираем скрипт cleanup_coverage.py, созданный ранее

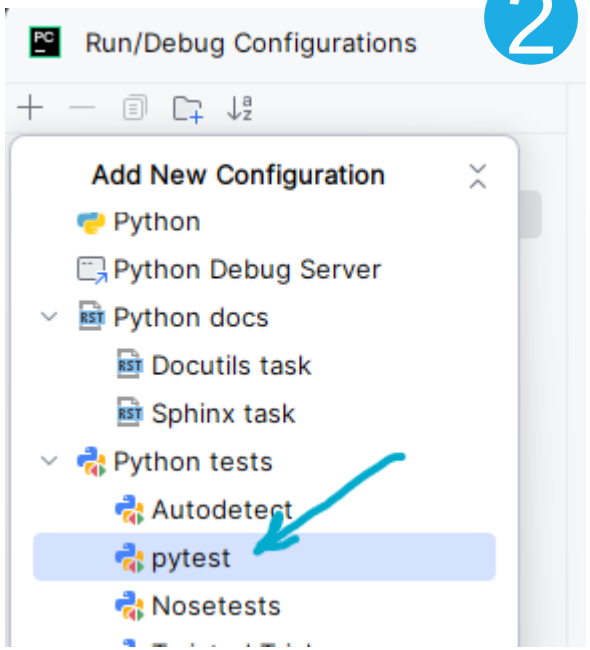
PYCHARM: СОЗДАНИЕ КОНФИГУРАЦИИ 2



1 **Перейти в режим редактирования**

Нажать +

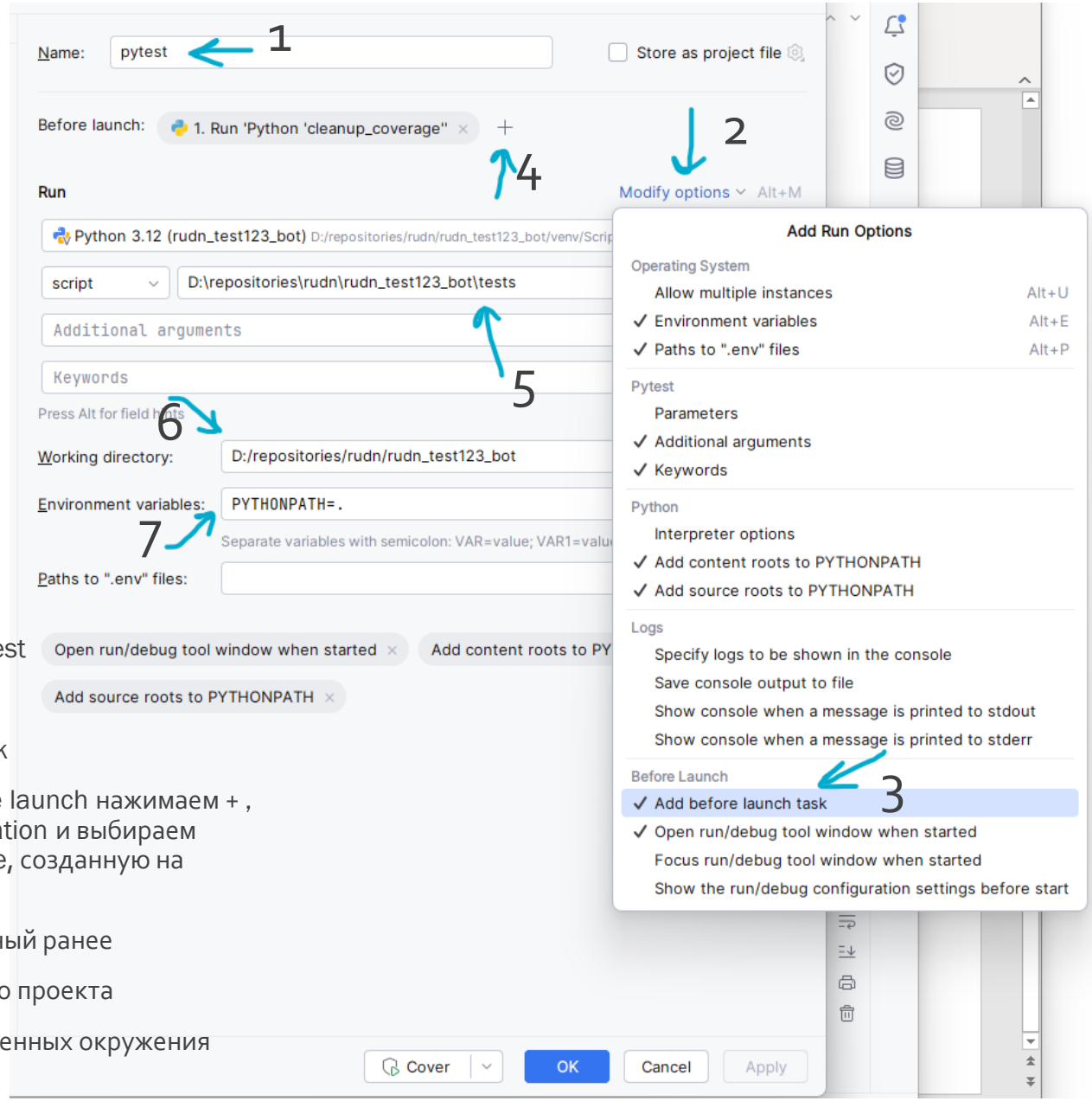
и выбрать pytest



2

3

1. Задаем имя конфигурации = pytest
2. Нажимаем Modify options
3. Выбираем Add before launch task
4. В появившейся настройке Before launch нажимаем + , нажимаем Run Another Configuration и выбираем конфигурацию cleanup_coverage, созданную на предыдущем шаге
5. Выбираем каталог tests, созданный ранее
6. Выбираем корневую директорию проекта
7. Указываем расположение переменных окружения PYTHONPATH=.



ДОБАВЛЕНИЕ КОНФИГУРАЦИЙ

1. Создать в корне проекта файл pytest.ini

```
[pytest]
pythonpath = .
testpaths = tests
addopts =
  -q
  --cov=.
  --cov-branch
  --cov-report=html
```

pytest.ini – конфигурационный файл для pytest.

Зачем нужен

- Чтобы не писать каждый раз кучу флагов в командной строке
- Чтобы одинаково запускать тесты из терминала, из PyCharm, из CI.

---

pythonpath (список путей, где python ищет модули)

testpaths (где искать тесты)

addopts (какие флаги добавлять)

-q (quiet режим)

--cov (включить pytest-cov и считать покрытие для текущей директории.

--cov-branch (считать покрытие ветвлений, а не только строк, т.е coverage смотрит, были ли выполнены обе ветки if/else, а не просто «строка была тронута»

--cov-report (после прогона тестов сгенерировать HTML-отчет (папка htmlcov/ с index.html), где можно наглядно увидеть покрытие по файлам/строкам)

ДОБАВЛЕНИЕ КОНФИГУРАЦИЙ

2. Создать в корне проекта файл .coveragerc

```
[run]
branch = True
source =
    db
    openrouter_client
    main
omit =
    tests/*
    */venv/*
    main_DailyZodiakBot.py
    conftest.py
    config.py
    cleanup_coverage.py

[report]
show_missing = True
skip_covered = False
precision = 1

[html]
directory = htmlcov
```

.coveragerc - конфигурационный файл для coverage.

Зачем нужен

- Управление подсчетом и отображением покрытия кода

---

branch (считать покрытие ветвлений, а не только строк)

source (мерить покрытие только для этих модулей)

omit (что не включать в покрытие)

show_missing (показывать номера строк, которые не покрыты)

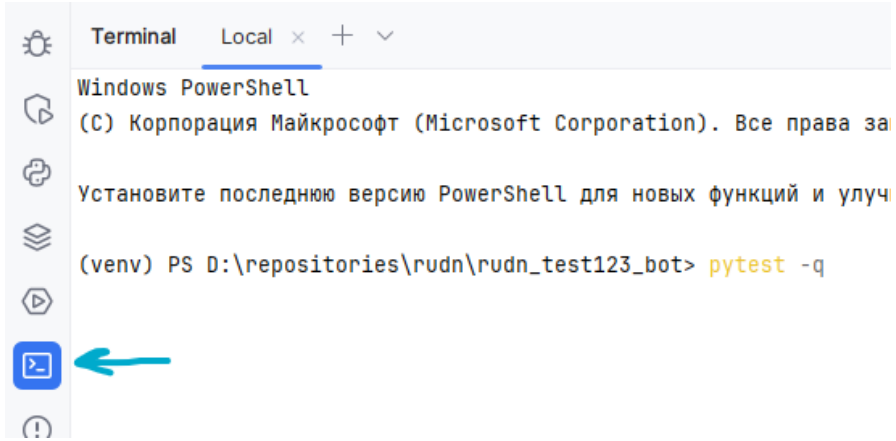
skip_covered (скрывать файлы со 100% покрытием)

precision (количество знаков после запятой в процентах)

КАК ЗАПУСТИТЬ ТЕСТЫ?

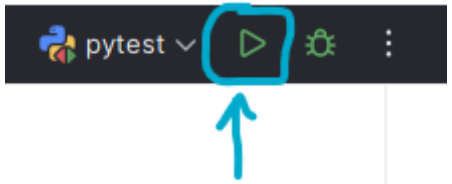
Через консоль командой

```
pytest -q
```

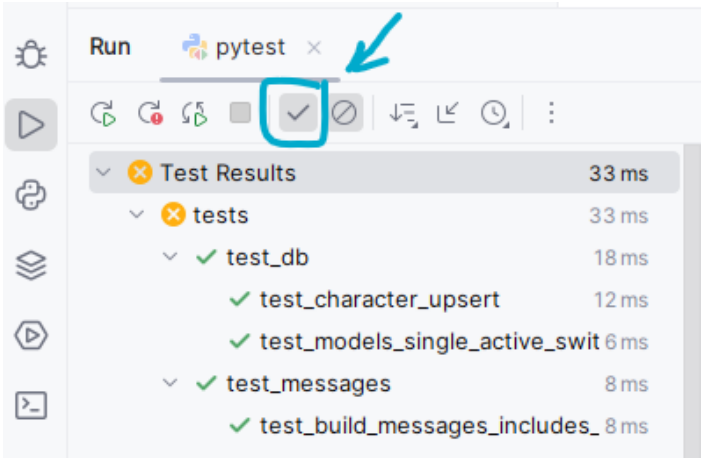


Через конфигурацию

```
pytest
```



Рекомендуется использовать данный способ, т.к под него нами были подготовлены необходимые конфигурации



После первого запуска включите флажок **Show Passed** для отображения всех запущенных тестов и статусов по ним

ФИКСТУРА **MONKEYPATCH**

monkeypatch — это фикстура pytest, которая позволяет временно подменять поведение кода во время теста:

- подменять атрибуты/функции/классы в модулях
- менять переменные окружения
- подменять текущую рабочую директорию и т.д.

После теста все автоматически откатывается назад.

Пример:

- подмена переменной окружения

```
101  # app.py
102  import os
103
104  def get_db_url():  1 usage
105      return os.getenv("DB_URL", "sqlite:///:memory:")
```

```
108  # test_app.py
109  def test_get_db_url(monkeypatch):
110      monkeypatch.setenv( name: "DB_URL",  value: "postgres://user:pass@localhost/db")
111
112      assert get_db_url() == "postgres://user:pass@localhost/db"
```

СОЗДАНИЕ ОСНОВЫ ФИКСТУР

```python
import importlib
import pytest

@pytest.fixture()  2 usages
def tmp_db_path(tmp_path):
    """
    Путь к временной БД
    """
    return str((tmp_path / "bot_test.db").absolute())

@pytest.fixture()  9 usages
def db_module(tmp_db_path, monkeypatch):
    """
    Поднимаем чистую БД в temp-файле.
    """
    db = importlib.import_module("db")
    # на случай, если DB_PATH читается из config:
    monkeypatch.setattr(db,  name: "DB_PATH", tmp_db_path, raising=False)
    db.init_db()
    return db

@pytest.fixture()  2 usages
def main_module(db_module, monkeypatch):
    """
    Импортируем main.py
    """
    main = importlib.import_module("main")
    return main

@pytest.fixture()  7 usages
def openrouter_module():
    """
    Импортируем openrouter_client.py
    """
    return importlib.import_module("openrouter_client")
```

/tests/conftest.py

В conftest.py создана вся инфраструктура:

db_module - дает нам модуль db с временной БД (через DB_PATH → temp-файл и init_db())

main_module - импортирует main, чтобы тесты могли вызвать _build_messages и другие функции

openrouter_module - импорт клиента OpenRouter

Все остальные тесты работают только через эти фикстуры, не думая о путях, .env и т.д.

СОЗДАНИЕ ТЕСТОВ 1

/tests/test_db.py

```python
def test_character_upsert(db_module):
    db = db_module
    # Предполагаем, что список персонажей уже есть. Берём любую существующую запись.
    characters = db.list_characters() if hasattr(db, "list_characters") else db.list_characters()
    assert characters, "Список персонажей пуст — проверь schema в db.py"
    any_id = characters[0]["id"]

    uid = 777001
    db.set_user_character(uid, any_id)
    p1 = db.get_user_character(uid)
    assert p1["id"] == any_id

    # Выберем другого персонажа и убедимся, что запись обновилась
    other_id = characters[-1]["id"] if characters[-1]["id"] != any_id else characters[1]["id"]
    db.set_user_character(uid, other_id)
    p2 = db.get_user_character(uid)
    assert p2["id"] == other_id

def test_models_single_active_switch(db_module):
    db = db_module
    models = db.list_models()
    assert len(models) >= 2, "Нужно >= 2 моделей в списке"
    a, b = models[0]["id"], models[1]["id"]

    # Сделать активной A
    db.set_active_model(a)
    act = db.get_active_model()
    assert act["id"] == a

    # Сделать активной B
    db.set_active_model(b)
    act2 = db.get_active_model()
    assert act2["id"] == b

    # Ровно одна активная
    with db._connect() as conn:
        cnt = conn.execute("SELECT COUNT(*) FROM models WHERE active=1").fetchone()[0]
    assert cnt == 1, "Должна быть ровно одна активная модель"
```

Первый тест про персонажей - проверка UPSERT:

- выбираем любого персонажа из characters
- set_user_character для пользователя uid
- убеждаемся, что get_user_character возвращает именно этого персонажа
- затем ставим другого персонажа и убедимся, что теперь у пользователя другой персонаж, а не две записи

Второй тест - про LLM-модели и условие «ровно одна активная модель»

- берем два разных id моделей
- делаем активной A → проверяем
- делаем активной B → проверяем
- и затем через прямой SQL (db._connect()) выполняем COUNT(*) WHERE active=1 и ожидаем ровно 1 запись

Проверяем результат. Запускаем тесты.

ДОПОЛНЕНИЕ ТЕСТОВ 1

/tests/test_db.py

```python
import pytest

def test_set_active_model_rejects_unknown_id(db_module):
    db = db_module

    # Берём гарантированно несуществующий ID (например, очень большое число)
    unknown_id = 999999

    with pytest.raises(ValueError) as excinfo:
        db.set_active_model(unknown_id)

    # Сообщение об ошибке из db.py:
    assert "Неизвестный ID модели" in str(excinfo.value)
```

В db.set_active_model() в случае неизвестного ID вызывается ValueError("Неизвестный ID модели").

Мы проверяем это поведение:

- с неизвестным ID функция не должна тихо ничего не делать
- она должна бросить исключение с понятным текстом

Такой тест защищает нас от попытки «заглушить» ошибку: если кто-то уберет raise ValueError, мы это увидим.

Проверяем результат. Запускаем тесты.

ДОПОЛНЕНИЕ ТЕСТОВ 1

/tests/test_db.py

```python
def test_get_user_character_falls_back_to_default(db_module):
    db = db_module

    # Берём пользователя, для которого мы точно ничего не записывали
    uid = 999001

    # Вызов без предварительного set_user_character
    ch = db.get_user_character(uid)

    # Ожидание: нам вернётся существующий персонаж из таблицы characters
    all_chars = db.list_characters()
    assert all_chars, "Список персонажей пуст — проверь schema в db.py"

    ids = {c["id"] for c in all_chars}
    assert ch["id"] in ids, "get_user_character должен возвращать одного из существующих персонажей"
```

В db.get_user_character() если у пользователя нет записи в user_character, он получает базового персонажа (id=1, либо первую запись).

Мы проверяем это поведение:

- функция должна вернуть какого-то существующего персонажа
- она должна бросить исключение с понятным текстом

Такой тест защищает нас от попытки «заглушить» ошибку: если кто-то уберет raise ValueError, мы это увидим.

Проверяем результат. Запускаем тесты.

СОЗДАНИЕ ТЕСТОВ 2

/tests/test_format_validator.py

```python
POINT_PREFIXES = (
    "1.", "2.", "3.", "4.", "5.", "6.",
    "1)", "2)", "3)", "4)", "5)", "6)", "- ",
)

def validate_formatted_answer(  1 usage
        text: str,
        min_points: int = 3,
        max_points: int = 5,
        max_intro_words: int = 12) -> bool:
    """
    Простая эвристика: есть короткое вступление (первая строка),
    есть 3-5 нумерованных пунктов, и есть короткий финальный вывод.
    """
    # 1) Разбить текст на непустые строки
    lines = [l.strip() for l in text.strip().splitlines() if l.strip()]
    if len(lines) < 3:
        return False

    # 2) Вступление: первая строка, не слишком длинная
    intro_words = len(lines[0].split())
    if intro_words == 0 or intro_words > max_intro_words:
        return False

    # 3) Пункты: строки между первой и последней, начинающиеся с префиксов
    points = [ln for ln in lines[1:-1] if ln.startswith(POINT_PREFIXES)]
    if not (min_points <= len(points) <= max_points):
        return False

    # 4) Вывод: последняя строка, не слишком длинная (<= 20 слов)
    return len(lines[-1].split()) <= 20
```

Создаем функцию, которая будет проверять, что ответ бота оформлен в соответствующем формате:

- короткое вступление
- 3–5 нумерованных пунктов
- короткий вывод

Сигнатура функции:

text - весь ответ бота одной строкой (с \n внутри)

min_points - минимальное количество пунктов (по умолчанию 3)

max_points - максимальное количество пунктов (по умолчанию 5)

max_intro_words - максимальная длина вступления в словах (по умолчанию 12)

СОЗДАНИЕ ТЕСТОВ 2

/tests/test_format_validator.py

```python
import pytest


@pytest.mark.parametrize(
    "text, expected_ok, reason",
    [
        (
            "Краткое вступление.\n\n"
            "1. Первый пункт\n"
            "2. Второй пункт\n"
            "3. Третий пункт\n\n"
            "Короткий вывод.",
            True,
            "минимальный валидный случай (3 пункта)",
        ),
        (
            "Вступление без пунктов.\n\n"
            "Вывод один.",
            False,
            "нет нумерованных пунктов",
        ),
        (
            "Очень длинное вступление с большим количеством слов, "
            "которое должно провалить проверку, "
            "потому что оно превышает max_intro_words.\n\n"
            "1. Пункт\n"
            "2. Пункт\n"
            "3. Пункт\n\n"
            "Вывод.",
            False,
            "слишком длинное вступление",
        ),
        (
            "Вступление.\n\n"
            "1) Пункт\n"
            "2) Пункт\n\n"
            "Вывод.",
            False,
            "слишком мало пунктов (< min_points)",
        ),
        (
            "Вступление.\n\n"
            "1. Пункт\n"
            "2. Пункт\n"
            "3. Пункт\n"
            "4. Пункт\n"
            "5. Пункт\n"
            "6. Пункт\n\n"
            "Вывод.",
            False,
            "слишком много пунктов (> max_points)",
        ),
        (
            "Вступление.\n\n"
            "1. Пункт\n"
            "2. Пункт\n"
            "3. Пункт\n\n"
            "Это очень длинный вывод, который содержит существенно больше "
            "двадцати слов, потому что мы специально набиваем его лишними "
            "фразами, чтобы эвристика validate_formatted_answer посчитала "
            "его слишком длинным и отклонила формат ответа.",
            False,
            "слишком длинный вывод",
        ),
    ],
    ids=[
        "ok_3_points",
        "no_points",
        "too_long_intro",
        "too_few_points",
        "too_many_points",
        "too_long_outro",
    ],
)
def test_validate_formatted_answer_variants(text, expected_ok, reason):
    assert validate_formatted_answer(text) is expected_ok, reason
```

Здесь мы фиксируем поведение функции:

- один валидный пример
- несколько невалидных: нет пунктов, длинное вступление, мало пунктов, слишком много пунктов, длинный вывод

Проверяем результат. Запускаем тесты.

СОЗДАНИЕ ТЕСТОВ 3

/tests/test_messages.py

Тестируем функцию _build_messages из main.py.

```python
def test_build_messages_includes_character_and_rules(db_module, main_module, monkeypatch):
    db = db_module
    main = main_module

    # создаём пользователя и устанавливаем персонажа
    uid = 42001
    characters = db.list_characters() if hasattr(db, "list_characters") else db.list_characters()
    target = characters[0]
    db.set_user_character(uid, target["id"])

    msgs = main._build_messages(uid, "Что такое API?")
    assert msgs[0]["role"] == "system"
    sys = msgs[0]["content"]

    # проверяем, что system содержит имя персонажа и часть правил
    assert target["name"] in sys
    assert "Формат" in sys or "Правила" in sys
    assert msgs[1]["role"] == "user"
    assert "Что такое API?" in msgs[1]["content"]
```

Функция:

- берет персонажа пользователя через db.get_user_character
- формирует system-сообщение с именем и prompt персонажа + блок с правилами
- добавляет user-сообщение с вопросом

Тест проверяет:

- msgs[0] - system
- system содержит имя персонажа и слова «Формат» или «Правила»
- msgs[1] - user, с текстом вопроса

Проверяем результат. Запускаем тесты.

ДОПОЛНЕНИЕ ТЕСТОВ 3

/tests/test_messages.py

```python
def test_build_messages_uses_default_character_when_not_set(db_module, main_module):
    db = db_module
    main = main_module

    uid = 50001
    question = "Расскажи что-нибудь о себе"

    # здесь мы НЕ вызываем set_user_character
    msgs = main._build_messages(uid, question)

    assert msgs[0]["role"] == "system"
    sys_text = msgs[0]["content"]

    # персонаж по-умолчанию - один из characters в БД
    characters = db.list_characters()
    assert characters, "Список персонажей пуст — проверь schema в db.py"
    names = {c["name"] for c in characters}

    # В system-тексте должно быть имя какого-то персонажа
    assert any(name in sys_text for name in names), "system должен содержать имя хоть какого-то персонажа"

    assert msgs[1]["role"] == "user"
    assert question in msgs[1]["content"]
```

Добавляем тест проверки возврата персонажа по-умолчанию, когда явного выбора не было

Проверяем результат. Запускаем тесты.

ДОПОЛНЕНИЕ ТЕСТОВ 3

/tests/test_messages.py

```python
def test_build_messages_for_character_pure_function(main_module):
    main = main_module

    character = {
        "id": 123,
        "name": "Тестовый персонаж",
        "prompt": "Отвечай коротко и по делу.",
    }
    question = "Что такое интеграционный тест?"

    msgs = main._build_messages_for_character(character, question)

    assert isinstance(msgs, list)
    assert msgs[0]["role"] == "system"
    assert "Тестовый персонаж" in msgs[0]["content"]
    assert "Отвечай коротко и по делу." in msgs[0]["content"]

    assert msgs[1]["role"] == "user"
    assert question in msgs[1]["content"]
```

Добавляем тест проверки, что функция _build_messages_for_character правильно собирает список сообщений для LLM из переданного ей персонажа и текста пользователя

БИБЛИОТЕКА **RESPONSES**

responses — это библиотека, которая перехватывает HTTP-запросы (обычно через requests) и позволяет подставлять моки

@responses.activate — это декоратор, который:

- оборачивает тест так, что внутри него все HTTP-запросы через requests (и связанные обертки) не уходят в сеть
- вместо этого проверяются на совпадение с теми, что мы добавили через responses.add()
- для совпавших запросов возвращаются заданные тобой status, json, body, headers и т.п.
- после выхода из теста мок снимается и HTTP работает как обычно

Зачем:

- Тесты не зависят от реального клиента OpenRouter, интернета и других параметров
- Один и тот же код теста всегда получает один и тот же ответ
- Мы можем моделировать ошибки (401, 429, 503 и т.д)

СОЗДАНИЕ ТЕСТОВ 4

/tests/test_openrouter_client.py

```python
import json
import responses
import os
import time
from importlib import reload

@responses.activate
def test_chat_once_ok_headers_and_body(openrouter_module, monkeypatch):
    chat_once = openrouter_module.chat_once
    url = "https://openrouter.ai/api/v1/chat/completions"
    payload = {"id": "cmpl_1", "choices": [{"message": {"content": "OK"}}]}
    responses.add(responses.POST, url, json=payload, status=200)

    monkeypatch.setenv( name: "OPENROUTER_API_KEY",  value: "test-key")
    openrouter_module = reload(openrouter_module)  # перечитает ключ
    chat_once = openrouter_module.chat_once

    text, ms = chat_once(
        messages=[{"role": "user", "content": "ping"}],
        model="mistralai/mistral-small-24b-instruct-2501:free",
        temperature=0.2,
        max_tokens=32,
        timeout_s=5,
    )
    assert text == "OK"
    # проверяем отправленное тело
    sent = json.loads(responses.calls[0].request.body.decode())
    assert sent["model"].endswith(":free")
    assert sent["messages"][0]["role"] == "user"
    assert "temperature" in sent and "max_tokens" in sent
    # и заголовки
    hdrs = responses.calls[0].request.headers
    assert hdrs["Content-Type"] == "application/json"
    assert hdrs["Authorization"] == "Bearer test-key"
```

Проверка успешного сценария использования клиента

- Берем функцию chat_once() из модуля openrouter_client
- Задаем URL OpenRouter’а
- Через responses.add регистрируем фейковый HTTP-ответ (т.е любой запрос на этот URL внутри теста получит не настоящий ответ сети, а именно этот payload)
- Подменяем API ключ, перезагружаем модуль и берем обновленную функцию chat_once()
- Вызываем chat_once() и проверяем результат (соответствие контракту)

ДОПОЛНЕНИЕ ТЕСТОВ 4

/tests/test_openrouter_client.py

Тестируем обработку ошибок HTTP (коды 401, 429)

```python
@responses.activate
def test_chat_once_errors_map_to_exception(openrouter_module, monkeypatch):
    chat_once = openrouter_module.chat_once
    OpenRouterError = openrouter_module.OpenRouterError

    url = "https://openrouter.ai/api/v1/chat/completions"
    monkeypatch.setenv( name: "OPENROUTER_API_KEY",  value: "key")
    responses.add(responses.POST, url, json={"error":"bad"}, status=401)

    try:
        chat_once(messages=[{"role":"user","content":"x"}],
                  model="meta-llama/llama-3.1-8b-instruct:free",
                  temperature=0.2,max_tokens=16,timeout_s=3)
        assert False, "Ожидалось исключение при 401"
    except OpenRouterError as e:
        assert "401" in str(e)

    responses.reset()
    responses.add(responses.POST, url, json={"error":"bad"}, status=429)
    try:
        chat_once(messages=[{"role":"user","content":"x"}],
                  model="meta-llama/llama-3.1-8b-instruct:free",
                  temperature=0.2,max_tokens=16,timeout_s=3)
        assert False
    except OpenRouterError as e:
        assert "429" in str(e)
```

- Берем функцию chat_once() из модуля openrouter_client
- Задаем URL OpenRouter’a
- Регистрируем ответ (status=401)
- Вызываем chat_once() и проверяем результат (соответствие контракту)
- Сбрасываем предыдущие mock-ответы
- Регистрируем ответ (status=429)
- Вызываем chat_once() и проверяем результат (соответствие контракту)

Проверяем результат. Запускаем тесты.

ДОПОЛНЕНИЕ ТЕСТОВ 4

/tests/test_openrouter_client.py

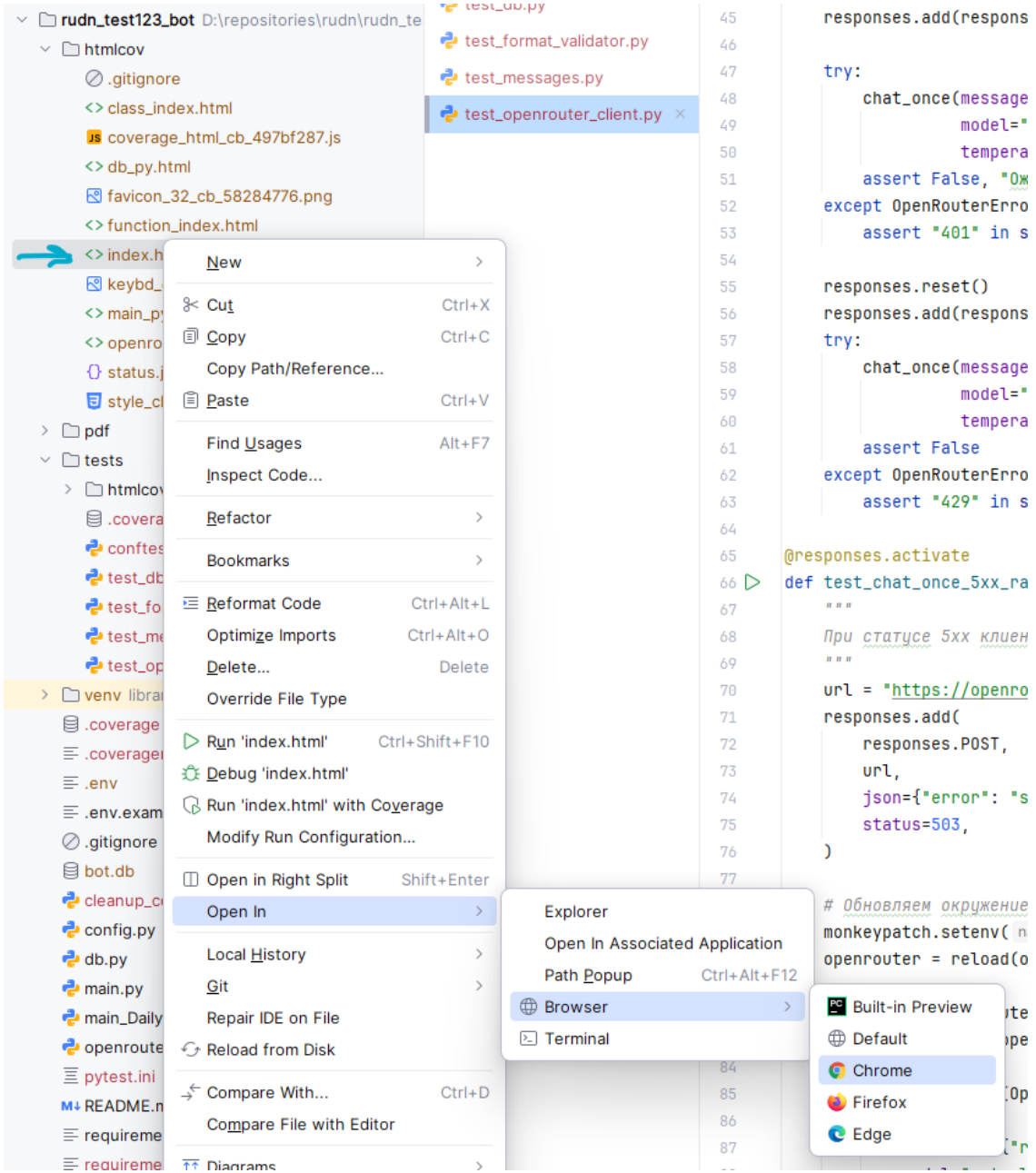
```python
@responses.activate
def test_chat_once_5xx_raises_openrouter_error(openrouter_module, monkeypatch):
    """
    При статусе 5xx клиент должен выбрасывать OpenRouterError
    """
    url = "https://openrouter.ai/api/v1/chat/completions"
    responses.add(
        responses.POST,
        url,
        json={"error": "server down"},
        status=503,
    )

    # Обновляем окружение и перечитываем модуль
    monkeypatch.setenv( name: "OPENROUTER_API_KEY",  value: "test-key")
    openrouter = reload(openrouter_module)

    chat_once = openrouter.chat_once
    OpenRouterError = openrouter.OpenRouterError

    with pytest.raises(OpenRouterError) as excinfo:
        chat_once(
            messages=[{"role": "user", "content": "ping"}],
            model="meta-llama/llama-3.1-8b-instruct:free",
            temperature=0.1,
            max_tokens=16,
            timeout_s=3,
        )

    err = excinfo.value
    assert err.status == 503
    assert "Сервис недоступен" in str(err)
```

Тестируем обработку ошибок HTTP (коды 5xx)

- Берем функцию chat_once() из модуля openrouter_client
- Задаем URL OpenRouter’а
- Вызываем chat_once() и проверяем результат (соответствие контракту)

ПРОВЕРКА ПОКРЫТИЯ

- Выполняем тесты
- Переходим в каталог htmlcov
- Открываем в любом браузере **index.html**



ПРОВЕРКА ПОКРЫТИЯ

Coverage report: 36.4%

Files | Functions | Classes

coverage.py v7.11.1, created at 2025-11-15 00:05 +0300

File ▲	statements	missing	excluded	branches	partial	coverage
db.py	145	70	0	20	3	49.7%
main.py	231	172	0	46	1	21.7%
openrouter_client.py	32	3	0	4	1	88.9%
Total	**408**	**245**	**0**	**70**	**5**	**36.4%**

- **Statements** – количество строк, которые coverage считает кодом, который можно выполнить
- **Missing** – сколько строк не выполнились («мертвые» участки до которых тесты не добираются)
- **Excluded** – сколько строк исключено из подсчета
- **Branches** – количество логических веток, покрытых тестами (например, если есть «if x > 0: … else: …» — там 2 ветви: «x > 0» и «иначе»)
- **Partial** – те случаи, когда строка с условием выполнялась, но не все варианты ветвления были пройдены тестами
- **Coverage** – процент покрытия файла тестами. Формула:

coverage ≈ (statements − missing) / (statements − excluded) * 100%

Coverage for **openrouter_client.py**: 88.9%

32 statements | 29 run | 3 missing | 0 excluded | 1 partial

« prev ^ index » next coverage.py v7.11.1, created at 2025-11-15 00:05 +0300

```
"""
Клиент взаимодействия с сервисом Openrouter
"""

from __future__ import annotations
import os, time, requests
from dataclasses import dataclass
from typing import Dict, List, Tuple
from dotenv import load_dotenv

load_dotenv()

OPENROUTER_API = "https://openrouter.ai/api/v1/chat/completions"
OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")

@dataclass
class OpenRouterError(Exception):
    status: int
    msg: str
    def __str__(self) -> str:
        return f"[{self.status}] {self.msg}"

def _friendly(status: int) -> str:
    return {
        400: "Неверный формат запроса.",
        401: "Ключ OpenRouter отклонён. Проверьте OPENROUTER_API_KEY.",
        403: "Нет прав доступа к модели.",
        404: "Эндпоинт не найден. Проверьте URL /api/v1/chat/completions.",
        429: "Превышены лимиты бесплатной модели. Попробуйте позднее.",
    }.get(status, "Сервис недоступен. Повторите попытку позже.")

def chat_once(messages: List[Dict], *,
              model: str,
              temperature: float = 0.2,
              max_tokens: int = 400,
              timeout_s: int = 30) -> Tuple[str, int]:
    if not OPENROUTER_API_KEY:
        raise OpenRouterError(401, "Отсутствует OPENROUTER_API_KEY (.env).")
    headers = {
        "Authorization": f"Bearer {OPENROUTER_API_KEY}",
        "Content-Type": "application/json",
    }
    payload = {
        "model": model,
        "messages": messages,
        "temperature": temperature,
        "max_tokens": max_tokens,
    }
    t0 = time.perf_counter()
    r = requests.post(OPENROUTER_API, json=payload, headers=headers, timeout=timeout_s)
    dt_ms = int((time.perf_counter() - t0) * 1000)
    if r.status_code // 100 != 2:
        raise OpenRouterError(r.status_code, _friendly(r.status_code))
    try:
        data = r.json()
        text = data["choices"][0]["message"]["content"]
    except Exception:
        raise OpenRouterError(500, "Неожиданная структура ответа OpenRouter.")
    return text, dt_ms
```

ДОМАШНЯЯ РАБОТА

- Добавить 3 любых теста, покрывающих любой, выбранный вами код.
- В каталог tests приложить скриншоты покрытия «до» написанных вами тестов и «после». На скриншотах должно быть видно название метода, код метода и результат покрытия (скриншоты из /htmlcov/index.html).