ЛАБОРАТОРНАЯ РАБОТА №6

Тема лабораторной работы: автоматизация тестирования

**1. Разработка автотестов**

**a. Цель работы**

Цель работы — разработка автотестов для проверки функционала телеграмм бота, который формирует рекомендации по отеля, исходя из запроса пользователя. Автотесты должны воспроизводить поведение пользователя при взаимодействии с элементами пользовательского интерфейса и обеспечивать контроль результатов выполненного сценария.

**b. Описание реализованных автотестов: инструменты, подходы**

**Инструменты:**

* **Pytest**

Фреймворк для тестирования в Python. Ее используют для написания и выполнения тестового кода.

* **Unittest**

unittest - инструмент для тестирования в Python. Это стандартный модуль для написания юнит-тестов на Python.

**Подходы:**

* Модульное тестирование: проверка функциональности отдельных компонентов и функций приложения.  
  Будет проверен следующий функционал:  
  - get\_check\_in   
  Данный функционал по параметрам пользователя о дате приезда будет искать возможные варианты заселения в отель

- get\_name\_hotel  
Данный функционал по параметрам пользователя выводит название отеля, который ему больше всего подходит  
- get\_photo  
Данный функционал по результатам поиска выводит фотографию отеля, чтобы пользователь заранее мог понимать, хочет он в него заселиться или нет

* Тестирование пользовательского интерфейса с использованием эмуляции поведения пользователя.

**c. Код автотестов**

Код представляет набор автотестов для тестирования функциональности телеграмм бота, который выводит рекомендации по заданным параметрам пользователя. В этих тестах проверяется корректность работы следующих сценариев:

1. Корректный ввод даты заселения
2. Корректный вывод названия отеля по указанным параметрам
3. Вывод фотографии по результатам поиска

Каждый тест проверяет ожидаемый результат при выполнении соответствующего сценария и сравнивает его с фактическим результатом телеграмм бота. Если ожидаемый и фактический результаты не совпадают, тест будет считаться неудачным.

Код для тестирования представлен в «Приложение 1».

**d. Отчет о тестировании**

* Тест-кейс 1: Вызов функции с информацией о дате  
  Результат: Успешно

Описание: Вызов функций с верными аргументами

* Тест-кейс 2: Корректный вывод названия отеля по указанным параметрам

Результат: Успешно

Описание: По заданным параметрам пользователя сформировался результат об отеле

* Тест-кейс 3: Выполнение запрос о выгрузке фото

Результат: Успешно

Описание: Фото выгружено по запросу

**e. Выводы по работе**

В результате тестирования приложения для телеграмм бота была подтверждена его работоспособность и стабильность. Программа корректно реагирует на ввод различного вида параметров, а также успешно формирует корректные запросы на выгрузку пользователю. Использование автоматизированных тестов позволило провести проверку функциональности более эффективно и ускорить процесс тестирования.

**Приложение 1**

**- get\_check\_in**  
pip install pytest

pip install pytest-mock  
import pytest

from unittest.mock import patch, MagicMock

from my\_telegram\_bot import get\_check\_in # Импортируем тестируемую функцию

class MockBot:

def retrieve\_data(self, user\_id, chat\_id):

return MagicMock(\_\_enter\_\_=lambda x: {}, \_\_exit\_\_=lambda x, y, z, w: None)

def send\_message(self, chat\_id, text):

return MagicMock(message\_id=1)

def register\_next\_step\_handler(self, msg, handler):

pass

@pytest.fixture

def mock\_bot(mocker):

bot = MockBot()

mocker.patch('my\_telegram\_bot.bot', bot)

return bot

# Тест функции get\_check\_in

def test\_get\_check\_in(mock\_bot, mocker):

mock\_message = MagicMock()

mock\_message.from\_user.id = 12345

mock\_message.chat.id = 67890

mock\_message.text = '2023-12-31'

with patch('my\_telegram\_bot.get\_check\_out') as mock\_get\_check\_out:

get\_check\_in(mock\_message)

# Проверяем, что retrieve\_data был вызван с правильными аргументами

mock\_bot.retrieve\_data.assert\_called\_once\_with(12345, 67890)

# Проверяем, что send\_message был вызван с правильными аргументами

mock\_bot.send\_message.assert\_called\_once\_with(67890, 'Date to departure(2020-01-01): ')

# Проверяем, что register\_next\_step\_handler был вызван с правильными аргументами

assert mock\_bot.register\_next\_step\_handler.call\_count == 1

called\_args, called\_kwargs = mock\_bot.register\_next\_step\_handler.call\_args

assert called\_args[1] == mock\_get\_check\_out

if \_\_name\_\_ == '\_\_main\_\_':

pytest.main()

**- get\_name\_hotel**import pytest

import requests

import requests\_mock

import json

from my\_telegram\_bot import get\_name\_hotel, create\_hotel\_list # Импортируем тестируемую функцию

URL = "https://api.example.com/"

HEADER = {

"Content-Type": "application/json",

"Authorization": "Bearer YOUR\_API\_KEY"

}

@pytest.fixture

def mock\_create\_hotel\_list(mocker):

return mocker.patch('my\_telegram\_bot.create\_hotel\_list', return\_value=['Hotel1', 'Hotel2'])

# Тест функции get\_name\_hotel

def test\_get\_name\_hotel(mock\_create\_hotel\_list):

query\_string = {"query": "Paris"}

mock\_message = MagicMock()

# Создаем мок HTTP-запроса

with requests\_mock.Mocker() as m:

mock\_response = {

"suggestions": [

{

"group": "CITY\_GROUP",

"entities": [

{"destinationId": "12345"}

]

}

]

}

m.get(URL + "locations/v2/search", text=json.dumps(mock\_response))

# Вызываем тестируемую функцию

result\_hotels = get\_name\_hotel(mock\_message, query\_string)

# Проверяем, что запрос был сделан с правильными параметрами

assert m.called

assert m.call\_count == 1

assert m.request\_history[0].url == f"{URL}locations/v2/search?query=Paris"

# Проверяем, что функция create\_hotel\_list была вызвана с правильными аргументами

mock\_create\_hotel\_list.assert\_called\_once\_with(mock\_message, "12345")

# Проверяем результат функции

assert result\_hotels == ['Hotel1', 'Hotel2']

if \_\_name\_\_ == '\_\_main\_\_':

pytest.main()

**- get\_photo**  
import pytest

from unittest.mock import MagicMock, patch

from telebot import types

from my\_telegram\_bot import get\_photo, bot, count\_photo # Импортируем тестируемую функцию и необходимые объекты

# Фикстура для создания мок-объекта сообщения

@pytest.fixture

def mock\_message():

message = MagicMock()

message.chat.id = 12345

return message

# Тест функции get\_photo

def test\_get\_photo(mock\_message, mocker):

# Мокаем метод bot.send\_message

mock\_send\_message = mocker.patch.object(bot, 'send\_message')

# Мокаем метод bot.register\_next\_step\_handler

mock\_register\_next\_step\_handler = mocker.patch.object(bot, 'register\_next\_step\_handler')

# Вызываем тестируемую функцию

get\_photo(mock\_message)

# Проверяем, что send\_message был вызван с правильными аргументами

assert mock\_send\_message.call\_count == 1

called\_args, called\_kwargs = mock\_send\_message.call\_args

assert called\_args[0] == 12345

assert called\_args[1] == 'Show hotels photo?'

assert isinstance(called\_kwargs['reply\_markup'], types.ReplyKeyboardMarkup)

# Проверяем, что в клавиатуре есть кнопки "Да" и "Нет"

keyboard = called\_kwargs['reply\_markup']

buttons = [button.text for button in keyboard.keyboard[0]]

assert 'Да' in buttons

assert 'Нет' in buttons

# Проверяем, что register\_next\_step\_handler был вызван с правильными аргументами

assert mock\_register\_next\_step\_handler.call\_count == 1

called\_args, called\_kwargs = mock\_register\_next\_step\_handler.call\_args

assert called\_args[1] == count\_photo

if \_\_name\_\_ == '\_\_main\_\_':

pytest.main()