[3 Разработка технологии тестирования 2](#_Toc4)

[3.1 Выбор подходов и методов тестирования 2](#_Toc5)

[3.2 Разработка плана автономного тестирования 3](#_Toc6)

[3.3 Разработка плана комплексного тестирования 4](#_Toc7)

[3.4 Выбор языка и библиотек для функционального тестирования 5](#_Toc8)

[3.5 Реализация и проведение функциональных тестов 6](#_Toc9)

[3.5 Нагрузочное тестирование 11](#_Toc10)

Обозначения

ПО — ...

API — ...

# 3 Разработка технологии тестирования

## **3.1 Выбор подходов и методов тестирования**

Процесс разработки программного обеспечения в том виде, как оно определяется в современной модели жизненного цикла программного обеспечения предполагает три стадии тестирования [1]:

* автономное тестирование компонентов ПО;
* комплексное тестирование разрабатываемого ПО;
* системное или оценочное тестирование на соответствие основным критериям качества.

Для проведения автономного и комплексного тестирования необходимо сформировать тестовые наборы, опираясь на структурный или функциональный подход.

Структурный подход базируется на том, что известна структура тестируемого ПО, в том числе его алгоритмы. Тесты строят так, чтобы обеспечить максимальное покрытие исходного кода.

Функциональный подход основывается на том, что структура ПО не известна. В этом случае тесты строят, опираясь на функциональные спецификации. Тесты строят на базе различных способов декомпозиции множества данных.

Разработанное ПО включает в себя разнородные алгоритмы, для всестороннего тестирования которых с помощью структурного подхода понадобились бы значительные затраты времени на изучение исходного кода, разработку большого числа тестов и заглушек. По этой причине было решено использовать функциональный подход, который позволил бы значительно сократить время на разработку тестов, обеспечивая при это тестирование всей необходимой функциональности [2].

В качестве оценочного тестирования согласно ТЗ было выбрано нагрузочное тестирование.

## 3.2 Разработка плана автономного тестирования

## 3.3 Разработка плана комплексного тестирования

## 3.4 Выбор языка и библиотек для функционального тестирования

Так как написание тестов на Golang требует значительного времени и такие тесты сложнее поддерживать в силу непопулярности языка среди тестировщиков, было решено тестировать разработанные микросервисы, предварительно запустив их (см. приложение Х) и обращаясь к ним по протоколу HTTP. Такой подход позволил реализовать тесты не привязываясь к языку реализации исходного ПО.

Поскольку Python обладает простым синтаксисом, большим количеством библиотек и популярен среди тестировщиков (рисунок Х), именно он был выбран для реализации тестов [3].



Рисунок Х — наиболее популярные языки в области автоматизированного тестирования

В качестве основной библиотеки для тестирования была выбрана библиотека pytest, являющаяся одной из наиболее популярных библиотек для автоматизированного тестирования [4].

Pytest обладает следующими основными преимуществами [5]:

* меньше повторяющегося кода за счет независимости от API;
* выполнение определенного набора тестов с помощью фильтрации;
* параметризация тестов — запуск одного и того же теста с разными наборами параметров;
* гибкость — архитектура библиотеки основана на плагинах, которые можно установить отдельно;
* полная обратная совместимость с unittest — возможность запуска тестов, написанных на нем;
* выполнение нескольких тестов параллельно;
* установочный код можно использовать повторно.

В дополнение к pytest была использована библиотека allure, формирующая интерактивные отчеты о прохождении тестов. Тесты в allure можно иерархически группировать и сопровождать логами и вложениями. Allure поддерживается не только для Python, но и для Java, JavaScript, Ruby, PHP , .Net и Scala.

Такой широкий набор поддерживаемых языков программирования делает allure (рисунок Х) знакомым многим разработчикам, тестировщикам и менеджерам, что упрощает поддержку тестов [6].



Рисунок Х — интерфейс allure

## 3.5 Реализация и проведение функциональных тестов

Для упрощения написания тестов и генерации отчетов был реализован вспомогательный модуль utils.py, отвечающий за отправку http-запросов к микросервисам, проверку http-ответов и их прикрепление к отчетам в allure.

Программный код utils.py приведен в листинге Х.

Листинг Х — программный код utils.py

|  |
| --- |
| import requests  import allure  import json  @allure.step("Send request") # отправка запроса - новый этап в отчете  def send\_request(port, url, payload):  path = f"http://127.0.0.1:{port}/{url}"  allure.attach(path, 'Request URL', allure.attachment\_type.TEXT)  # прикрепить данные запроса к отчету  allure.attach(json.dumps(payload, indent=4, ensure\_ascii=False).encode(), 'Request payload', allure.attachment\_type.TEXT)  resp = requests.post(path, json = payload)  # прикрепить данные ответа к отчету  allure.attach(json.dumps(resp.json(), indent=4, ensure\_ascii=False).encode(), 'Response payload', allure.attachment\_type.TEXT)  return resp  @allure.step("Check response [ok]") # проверка ответа без ошибок  def is\_ok\_response(response):  allure.attach(json.dumps(response.json(), indent=4, ensure\_ascii=False).encode(), 'Response payload', allure.attachment\_type.TEXT)  return response.status\_code == 200 \  and response.json()["status\_code"] == 200 \  and response.json()["status\_str"] == "ok"  @allure.step("Check response [error]") # проверка ответа при ошибке  def is\_error\_response(response):  allure.attach(json.dumps(response.json(), indent=4, ensure\_ascii=False).encode(), 'Response payload', allure.attachment\_type.TEXT)  return response.status\_code != 200 \  and response.json()["status\_code"] != 200 \  and response.json()["status\_str"] == "error"  def ordered\_json(obj): # упорядочивание JSON для последующего сравнения  if isinstance(obj, dict):  return sorted((k, ordered\_json(v)) for k, v in obj.items())  if isinstance(obj, list):  return sorted(ordered\_json(x) for x in obj)  else:  return obj |

Помимо utils.py, были реализованы вспомогательные модули settings.py и consts.py, которые определяют порты по которым доступны микросервисы и данные для HTTP-запросов в тестах.

Пример реализации простейшего теста (проверка работы анализатора для неправильно решенного пользователем теста с одним вариантом ответа) приведен в листинге Х.

Листинг Х — пример реализации теста

|  |
| --- |
| import utils.settings as settings  import utils.utils as utils  from utils.consts import \*  import allure  import copy  # < ...>  # определение иерархии и описания в отчете  @allure.description("Test for singlechoice wrong-answered task")  @allure.epic("Unit-testing")  @allure.story("Analyzer")  def test\_single\_correct\_negative():  # считывание данных при правильном ответе из констант  payload = copy.deepcopy(Analyzer.single\_valid\_positive)  # изменение ID выбранного ответа  payload["data"]["user\_answer\_id"] = 1  # http-запрос  resp = utils.send\_request(settings.ANALYZER\_PORT,  "check", payload)  # http-ответ успешен?  assert utils.is\_ok\_response(resp)  # http-ответ задание решено неверно?  assert resp.json()["is\_correct"] == False  # возвращена ожидаемая подсказка?  assert resp.json()[  "data"]["hint"] == payload["data"]["task"]["answers"][1]["hint"]  # < ...> |

Пример отчета, полученного после выполнения этого теста (и аналогичных ему) приведен на рисунках Х-Х.



Рисунок Х — отчет о результате теста



Рисунок Х — приложения к отчету

После реализации аналогичным образом всех тестов из таблиц M-N они были запущены.

Благодаря параллельному запуску тестов через плагин xdist (в данном случае — в 2 потока) удалось выполнить все тесты менее, чем за 600 мс (рисунок Х).



Рисунок Х — хронология запуска тестов

Статистика запуска тестов из отчета allure приведена на рисунке Х.



Рисунок Х — статистика запуска тестов

Все тесты завершились успешно, о чем свидетельствует приведенная статистика.

## 3.5 Нагрузочное тестирование

Для упрощения поддержки тестов реализовать нагрузочное тестирование было решено так же с помощью Python. Наиболее популярной библиотекой для нагрузочного тестирования с помощью Python является locust.

Источники

1. Книга Ивановой
2. <https://coderlessons.com/tutorials/kachestvo-programmnogo-obespecheniia/ruchnoe-testirovanie/chernyi-iashchik-protiv-belaia-korobka>
3. <https://habr.com/ru/post/543346/>
4. <https://www.softwaretestinghelp.com/python-testing-frameworks/>
5. <https://blog.skillfactory.ru/glossary/pytest/>
6. <https://docs.qameta.io/allure/>