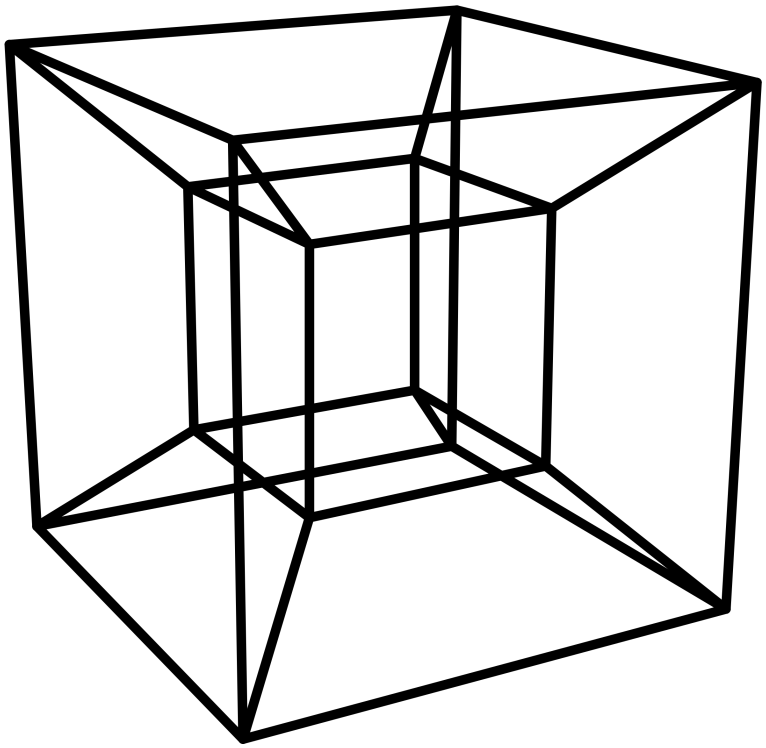
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«Санкт-Петербургский политехнический университет Петра Великого»

Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

ОТЧЕТ ПО ИНТЕГРАЦИОННОМУ ТЕСТИРОВАНИЮ

АГРЕГАТОР ЦИФРОВЫХ ФИНАНСОВЫХ АКТИВОВ «ТЕССЕРАКТ»

по дисциплине «Технологии разработки качественного программного обеспечения»



Выполнили студенты группы: 5130904/00104:

Почернин В. С.
Шиляев В. С.
Мурзаканов И. М.
Разукрантов В. Е.

Преподаватель:

Маслаков А. П.

Санкт-Петербург
2024

Содержание

1 Постановка задачи

Интеграционное тестирование предполагает наличие нескольких модулей или, если приложение построено в соответствии с микросервисной архитектурой, микросервисов разработанного приложения. В качестве модулей могут быть функциональные части (например: модуль авторизации/аутентификации, модуль взаимодействия с пользователем, модуль интеграции со сторонним сервисом).

Необходимо выполнить интеграционное тестирование нескольких модулей в соответствии с предварительно описанными сценариями тестирования. Каждый сценарий тестирования должен представлять собой некоторый законченный вариант использования системы (кейс) пользователя.

Интеграционное тестирование предполагает командную работу над программным продуктом, поэтому необходимо проводить его с использованием одного из инструментов `CI`, доступных на рынке: `Gitlab`, `Jenkins`, `Bamboo`, `TeamCity`, etc.

Минимальное количество сценариев - 10.

1.1 Обязательные требования

1) Предварительное формирование документа, описывающего тестовые сценарии.
2) Применение заглушек (mock-сервисов или mock-объектов) для изоляции от окружения и внешних сервисов или для ускорения прохождения тестов.
3) Рассмотрение негативных сценариев тестирования.
4) Поднятие сервера непрерывной интеграции и запуск задачи по интеграционному тестированию по временному триггеру или по событию изменения кода приложения/интеграционных тестов.

1.2 Содержание отчета

Отчет по интеграционному тестированию должен содержать:

1) Отчет о выполненной работе, использованных инструментах, примененных техниках тест-дизайна.
2) Тест-план со словесным описанием тестовых сценариев, которые планируется реализовать в интеграционном тестировании.
3) Отчет о прохождении тестов с результатами на сервере непрерывной интеграции.
4) Описание процедуры расширения тестового набора на примере добавления новой функциональной части (или модуля).

2 Ход работы

2.1 Отчет о выполненной работе, использованных инструментах, примененных техниках тест-дизайна

В рамках проведенной работы, код серверной части приложения был покрыт интеграционными тестами.

2.1.1 Использованные инструменты

Были использованы следующие инструменты:

- `JUnit` - фреймворк для языков программирования `Java` и `Kotlin`, предназначенный для автоматического unit-тестирования. Данный фреймворк позволяет удобно создавать, организовывать и выполнять тесты, благодаря широкому набору аннотаций и встроенной поддержки в популярных IDE. Для серверной части применялся `JUnit5`.
- `Spring Boot Test` - предоставляет инструменты для интеграционного тестирования `Spring Boot` приложений, включая поддержку тестирования в изолированном контексте приложения.
- `MockMvc` - инструмент из `Spring Test` для тестирования контроллеров в изоляции от сервлет контейнера. Позволяет отправлять `HTTP` запросы и проверять ответы без запуска сервера.
- `Testcontainers` - библиотека, которая поддерживает тесты, использующие `Docker` контейнеры. В нашем случае использовалась для поднятия `PostgreSQL` базы данных в `Docker` для тестирования взаимодействия с БД.
- `Flyway` - инструмент для версионирования базы данных, который применялся для миграции схемы и данных БД перед выполнением тестов.
- `GitHub Actions` - система непрерывной интеграции (`CI`), используемая нами для автоматического выполнения тестов при открытии `PR` или мерже в мастер ветку. Позволяет автоматизировать процесс интеграционного тестирования и не только.

2.1.2 Примененные техники тест-дизайна

Тест-дизайн - это процесс разработки техник и методов тестирования. Главная задача тест-дизайна - разработать сценарии, которые позволяют протестировать максимальное количество функций за минимальное время. Таким образом, разрабатывается тестовая документация, опирающаяся на общие принципы и логику тестирования с поправкой на особенности продукта. Суть всех техник тест-дизайна - оптимизировать процесс тестирования.

Нами были применены следующие техники тест-дизайна:

- **Классы эквивалентности**: данная техника тестирования состоит в том, что все тестовые кейсы группируются и разделяются на некие эквивалентные классы. При этом классы одной тестовой группы ведут себя одинаково. То есть, если система корректно обработает одно значение из класса, то она корректно обработает и все остальные значения из этого класса. Согласно одному из принципов тестирования - «полное тестирование программы невозможно, или займет недопустимо длительное время, поскольку в этом случае необходимо проверять слишком много комбинаций тестовых данных».

  Данная техника позволяет получить четкие результаты за ограниченное время, улучшает качество тест-кейсов, устраняя избыточность, при этом покрывая тестовые сценарии.

  Приведем пример теста:

```
@ParameterizedTest
@ValueSource(strings = {
        "",
        "ip1&",
        "incorrectPassword1&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&",
        "123&*(",
        "qweqweqwe",
        "qwe123&*(±"})
@DisplayName("Регистрация с некорректным паролем")
public void givenInvalidPassword_whenRegister_thenReturnsExpectedError(String password)
        throws Exception {
    HashMap<String, String> request = new HashMap<>();
    request.put("login", "correctLogin");
    request.put("email", "vspochernin@gmail.com");
    request.put("password", password);
```

```
    mockMvc.perform(post("/api/v1/register")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(request)))
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath("$.id").value(4))
            .andExpect(jsonPath("$.errorType").value("INVALID_PASSWORD"));
}
```

В данном примере тестируется корректность регистрации при вводе некорректного пароля. Вместо того, чтобы тестировать всевозможные комбинации паролей, мы разбиваем их на классы эквивалентности, такие как:

- Пустой пароль;
- Слишком короткий пароль;
- Слишком длинный пароль;
- Пароль без букв;
- Пароль без цифр или специальных символов;
- Пароль с некорректными символами;

- **Граничные условия**: при использовании данной техники тестирования, основное внимание уделяется значениям на границах допустимого диапазона, поскольку зачастую ошибки возникают именно в граничных точках - такая проверка помогает их быстро находить.

  Граничные значения - это такие места, в которых один класс эквивалентности переходит в другой. Это значения на границе допустимого диапазона входных данных, которые могут привести к изменению поведения программы.

  При использовании данной техники на каждой границе диапазона следует проверить по три значения:

  - Граничное значение;
  - Значение перед границей;
  - Значение после границы;

  Приведем пример теста:

```
private static Stream<Arguments> validCreateDiversificationRequestArgumentsProvider() {
    return Stream.of(
            Arguments.of(170_00L, 3),
            Arguments.of(5_000_000_00L, 3),
            Arguments.of(10_000_000_00L, 3),
            Arguments.of(5_000_000_00L, 0),
            Arguments.of(5_000_000_00L, 1),
            Arguments.of(5_000_000_00L, 2)
    );
}

@ParameterizedTest
@MethodSource("validCreateDiversificationRequestArgumentsProvider")
@DisplayName("Создание диверсификации с корректными данными")
public void givenValidCreateDiversificationRequest_whenCreateDiversification_thenSuccess(
        Long amount,
        Integer riskTypeId) throws Exception
{
    HashMap<String, Object> request = new HashMap<>();
    request.put("amount", amount);
    request.put("riskTypeId", riskTypeId);

    checkIfDiversificationWithId3IsNotExists();

    mockMvc.perform(post("/api/v1/diversifications")
```

```
                    .header("Authorization", Secrets.VRAZUKRANTOV_BEREAR_TOKEN)
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(request)))
            .andExpect(status().isCreated());

    mockMvc.perform(get("/api/v1/diversifications/3")
                    .header("Authorization", Secrets.VRAZUKRANTOV_BEREAR_TOKEN))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.currentAmount").value(lessThanOrEqualTo(amount.intValue())))
            .andExpect(jsonPath("$.riskTypeId").value(riskTypeId));
}

private static Stream<Arguments> invalidCreateDiversificationRequestArgumentsProvider() {
    return Stream.of(
            Arguments.of(169_99L, 3),
            Arguments.of(10_000_000_01L, 3),
            Arguments.of(5_000_000_00L, -1),
            Arguments.of(5_000_000_00L, 4)
    );
}

@ParameterizedTest
@MethodSource("invalidCreateDiversificationRequestArgumentsProvider")
@DisplayName("Создание диверсификации с некорректными данными")
public void givenInvalidCreateDiversificationRequest_whenCreateDiversification_thenReturnsExpectedError(
        Long amount,
        Integer riskTypeId) throws Exception
{
    HashMap<String, Object> request = new HashMap<>();
    request.put("amount", amount);
    request.put("riskTypeId", riskTypeId);

    checkIfDiversificationWithId3IsNotExists();

    mockMvc.perform(post("/api/v1/diversifications")
                    .header("Authorization", Secrets.VRAZUKRANTOV_BEREAR_TOKEN)
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(request)))
            .andExpect(status().isBadRequest());

    checkIfDiversificationWithId3IsNotExists();
}
```

Для создания диверсификаций у нас работает следующая логика: сумма диверсификации не должна быть меньше, чем стоимость самого дешевого актива и не должна быть больше, чем определенная граница (10 миллионов рублей). Уровень рискованности, в свою очередь, может принимать значения 0, 1, 2 или 3.

Таким образом, мы проверяем правильность работы алгоритма на границах диапазонов.

- **Попарное тестирование**: эта методика тестирования, при которой создаются тестовые случаи для проверки комбинаций каждой пары входных параметров. Это помогает значительно сократить количество тестов при сохранении высокого уровня покрытия, поскольку исследования показывают, что большинство ошибок вызывается взаимодействием между парой параметров.

  Приведем пример теста:

```
@Test
@DisplayName("Регистрация с корректными данными")
public void givenValidCredentials_whenRegister_thenSuccess() throws Exception {
    HashMap<String, String> request = new HashMap<>();
    request.put("login", "correctLogin");
    request.put("email", "correctEmail@gmail.com");
    request.put("password", "correctPassword123");
```

```
    mockMvc.perform(post("/api/v1/register")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(request)))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.token").exists())
            .andExpect(jsonPath("$.token").isString());
}

private static Stream<Arguments> invalidRegisterRequestArgumentsProvider() {
    return Stream.of(
            Arguments.of("correctLogin", "incorrectEmail.com", "incorrectPassword"),
            Arguments.of("il", "incorrectEmail.com", "correctPassword123"),
            Arguments.of("il", "correctEmail@gmail.com", "incorrectPassword"));
}

@ParameterizedTest
@DisplayName("Регистрация с некорректными данными")
@MethodSource("invalidRegisterRequestArgumentsProvider")
public void givenInvalidRegisterRequest_whenRegister_thenBadRequest(String login, String email,
        String password) throws Exception
{
    HashMap<String, String> request = new HashMap<>();
    request.put("login", login);
    request.put("email", email);
    request.put("password", password);

    mockMvc.perform(post("/api/v1/register")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(request)))
            .andExpect(status().isBadRequest());
}
```

Мы тестируем метод, который проверяет корректность трех полей - логина, адреса электронной почты и пароля. Если брать полную таблицу состояний - мы будем должны написать 8 тестов:

	login	**email**	**password**
1	valid	valid	valid
2	valid	valid	invalid
3	valid	invalid	valid
4	valid	invalid	invalid
5	invalid	valid	valid
6	invalid	valid	invalid
7	invalid	invalid	valid
8	invalid	invalid	invalid

Однако, используя технику попарного тестирования, мы можем сократить количество тестов до 4-х:

	login	**email**	**password**
1	valid	valid	valid
2	valid	invalid	invalid
3	invalid	invalid	valid
4	invalid	valid	invalid

2.2 Тест-план со словесным описанием тестовых сценариев которые были реализованы в интеграционном тестировании

Описание тест-плана удобно разделить на классы, созданные для тестирования.

2.2.1 AssetControllerTest.java

Данный класс тестирует контроллер активов.

1) `givenPageNumber_whenGetAssets_thenReturnsAssetsWithExpectedIds` - получение первых 10 и вторых 10 активов (имитация постраничной загрузки).
2) `givenAssetId10_whenGetAsset_thenReturnsExpectedAsset` - получение конкретного актива.
3) `givenNonExistent_whenGetAsset_thenReturnsExpectedError` - получение несуществующего актива.

2.2.2 AuthenticationControllerTest.java

Данный класс тестирует контроллер аутентификации.

1) `givenValidCredentials_whenRegister_thenSuccess` - регистрация с корректными данными.
2) `givenInvalidRegisterRequest_whenRegister_thenBadRequest` - регистрация с некорректными данными.
3) `givenCredentialsWithExistingLogin_whenRegister_thenReturnsExpectedError` - регистрация, когда пользователь уже существует.
4) `givenCredentialsWithExistingEmail_whenRegister_thenReturnsExpectedError` - регистрация, когда email уже существует.
5) `givenInvalidLogin_whenRegister_thenReturnsExpectedError` - регистрация с некорректным логином.
6) `givenInvalidPassword_whenRegister_thenReturnsExpectedError` - регистрация с некорректным паролем.
7) `givenInvalidEmail_whenRegister_thenReturnsExpectedError` - регистрация с некорректным email.
8) `givenCorrectCredentials_whenLogin_thenReturnsToken` - аутентификация с корректными данными.
9) `givenNonExistentLogin_whenLogin_thenReturnsExpectedError` - аутентификация с несуществующим логином.
10) `givenNonExistentPassword_whenLogin_thenReturnsExpectedError` - аутентификация с несуществующим паролем.
11) `givenCorrectRequest_whenChangePassword_thenSuccess` - корректное изменение пароля.
12) `givenNonExistentOldPassword_whenChangePassword_thenReturnsExpectedError` - изменение пароля, когда старый пароль не совпадает с тем, что лежит в базе.
13) `givenIncorrectNewPassword_whenChangePassword_thenReturnsExpectedError` - изменение пароля, когда пароль некорректный.

2.2.3 DiversificationControllerTest.java

Данный класс тестирует контроллер диверсификаций.

1) `givenCorrectRequest_whenGetDiversifications_thenReturnsDiversificationsWithExpectedIds` - получить первые 10 своих диверсификаций.

2) `givenDiversificationId2_whenGetDiversification_thenReturnsExpectedDiversification` - получить свою существующую диверсификацию.

3) `givenNonExistentOrNonYourDiversification_whenGetDiversification_thenReturnsExpectedErr` - получить не свою или несуществующую диверсификацию.

4) `givenValidCreateDiversificationRequest_whenCreateDiversification_thenSuccess` - создание диверсификации с корректными данными.

5) `givenInvalidCreateDiversificationRequest_whenCreateDiversification_thenReturnsExpected` - создание диверсификации с некорректными данными.

2.2.4 FavouritesControllerTest.java

Данный класс тестирует контроллер избранного.

1) `givenCorrectRequest_whenGetFavourites_thenReturnsFavouritesWithExpectedIdsAndStatus` - получить первые 10 своих избранных активов.

2) `givenExistsAssetId_whenAddFavourites_thenSuccess` - добавить существующий актив в избранное.

3) `givenExistsAssetId_whenRemoveFavourites_thenSuccess` - удалить существующий актив из избранного.

4) `givenNonExistentAssetId_whenAddFavourites_thenSuccess` - добавить несуществующий актив в избранное.

5) `givenNonExistentAssetId_whenRemoveFavourites_thenSuccess` - удалить несуществующий актив из избранного.

2.2.5 BadRequestTest.java

Данный класс тестирует некорректные тела запросов.

1) `givenInvalidQueryParams_whenHandleGetRequest_thenReturnsExpectedError` - некорректные query параметры GET запроса.

2) `givenIncorrectPathParams_whenHandleGetRequest_thenReturnsExpectedError` - некорректные параметры пути GET запроса.

3) `givenIncorrectPathParams_whenHandlePostRequest_thenReturnsExpectedError` - некорректные параметры пути POST запроса.

4) `givenIncorrectPathParams_whenHandleDeleteRequest_thenReturnsExpectedError` - некорректные параметры пути DELETE запроса.

5) `givenIncorrectBodyParams_whenHandlePostRequest_thenReturnsExpectedError` - некорректные параметры тела POST запроса.

6) `givenIncorrectBodyParams_whenHandlePutRequest_thenReturnsExpectedError` - некорректные параметры тела PUT запроса.

2.2.6 UnauthorizedTest.java

Данный класс тестирует некорректный токен аутентификации.

1) `givenBadCredentials_whenHandleGetRequest_thenReturnsExpectedError` - некорректные параметры аутентификации GET запроса.

2) `givenBadCredentials_whenHandlePostRequest_thenReturnsExpectedError` - некорректные параметры аутентификации POST запроса.

3) `givenBadCredentials_whenHandlePutRequest_thenReturnsExpectedError` - некорректные параметры аутентификации PUT запроса.

4) `givenBadCredentials_whenHandleDeleteRequest_thenReturnsExpectedError` - некорректные параметры аутентификации DELETE запроса.

2.3 Отчет о прохождении тестов с результатами на сервере непрерывной интеграции

За работу сервера непрерывной интеграции отвечает конфигурационный файл `integration_tests.yml`:

```yaml
name: Unit and Integration tests
on:
    push:
        branches: [ "master" ]
    pull_request:
        branches: [ "master" ]

jobs:
    unit-and-integration-test:
        runs-on: ubuntu-latest
        steps:
            -   name: Checkout
                uses: actions/checkout@v4

            -   name: Setup Java
                uses: actions/setup-java@v4
                with:
                    distribution: 'temurin'
                    java-version: '17'

            -   name: Setup Gradle
                uses: gradle/actions/setup-gradle@v3

            -   name: Run tests
                run: |
                    cd server
                    ./gradlew test

            -   name: Upload test reports
                if: always()
                uses: actions/upload-artifact@v3
                with:
                    name: test-reports
                    path: server/build/reports/tests/test/
```

Как можно заметить, мы выполняем следующие шаги:

- Копируем наш репозиторий в систему непрерывной интеграции.
- Устанавливаем виртуальную машину `Java`.
- Устанавливаем `Gradle`.
- Запускаем тесты с помощью `Gradle`.
- Загружаем артефакты тестов в систему интеграции.

Как можно заметить, запуск данного сценария происходит при следующих условиях:

- Произошел пуш в мастер-ветку.
- Открылся `PR` в мастер-ветку.

Страница с пройденным `CI` выглядит следующим образом:

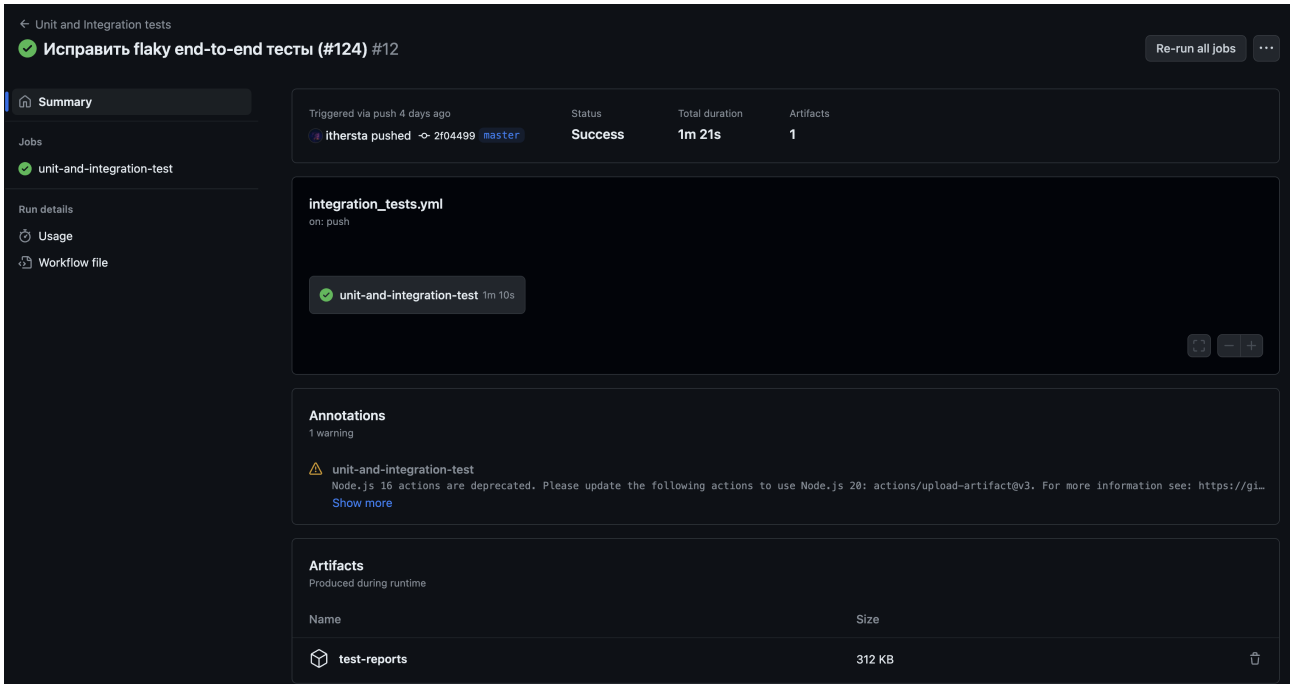
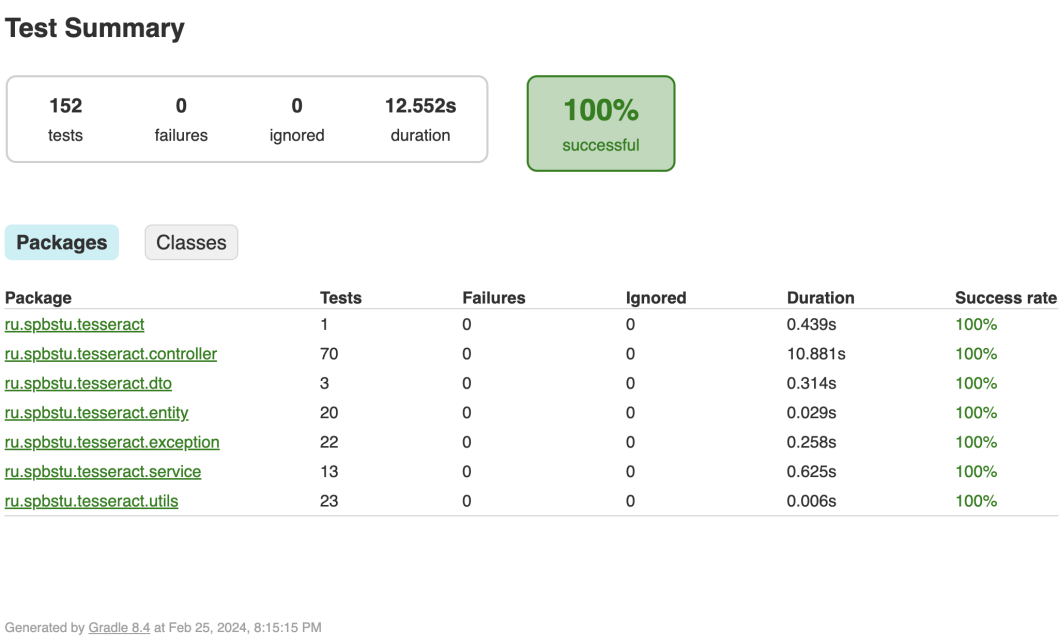


Рис. 1: Пройденный CI

Ниже можно скачать архив с артефактами тестирования. Он содержит в себе HTML-страницу с подробным отчетом о пройденных тестах.



Test Summary

152	0	0	12.552s
tests	failures	ignored	duration

100% successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
ru.spbstu.tesseract	1	0	0	0.439s	100%
ru.spbstu.tesseract.controller	70	0	0	10.881s	100%
ru.spbstu.tesseract.dto	3	0	0	0.314s	100%
ru.spbstu.tesseract.entity	20	0	0	0.029s	100%
ru.spbstu.tesseract.exception	22	0	0	0.258s	100%
ru.spbstu.tesseract.service	13	0	0	0.625s	100%
ru.spbstu.tesseract.utils	23	0	0	0.006s	100%

Generated by Gradle 8.4 at Feb 25, 2024, 8:15:15 PM

Рис. 2: Отчет о пройденных тестах

2.4 Описание процедуры расширения тестового набора на примере добавления новой функциональной части (или модуля)

При добавлении нового контроллера или же новых ручек для старого контроллера - их тестирование происходит по аналогии с уже существующими тестами.