Программное решение Spotify_API_testing.
Техническая документация

1. Общие сведения

Программное решение "Spotify_API_testing", далее – "программное решение", "ПР", разработано в учебных целях и не предназначено для какого-либо коммерческого использования.

Настоящее ПР предназначено для частичной автоматизации тестирования программируемого интерфейса приложения Spotify Web API стримингового музыкального сервиса "Spotify" (далее API) по заранее заданным сценариям поведения и взаимодействие его со сторонним клиент-приложением, созданным и зарегистрированным в системе сервиса Spotify.

Программное решение выполнено в интегрированной среде разработки IntelliJ IDEA на языке программирования Java с использованием библиотеки для тестирования API на основе REST архитектуры "REST-Assured", библиотеки автоматизированного управления веб-браузерами "Selenium WebDriver" и фреймворка для автоматизации тестирования программного обеспечения "TestNG"; сборщиком проекта выступает Maven. В качестве рабочего веб-браузера используется "Chrome" и утилита доступа и взаимодействия с ним "ChromeDriver".

2. Структура программного решения

"Spotify_API_testing" имеет иерархическую структуру в которой модули ПР разделены на общую "main" и тестовую "test' части.

В общем "main" модуле расположена вся логика работы программного решения: классы, описывающие веб-элементы графического интерфейса, представленных в DOM принимаемой веб-страницы авторизации в сервисе с которыми взаимодействует ПР, POJO-классы для сериализации в JSON тела запроса и десериализации принимаемых в JSON ответов, а также классы и методы для установки веб-драйвера и Base64 кодирования строк.

В тестовом "test" модуле расположены классы, описывающие тестовые сценарии по которым происходит автоматизированное тестирование API сервиса Spotify и тест проверки создания POJO, классы с исходными данными для тестирования и их обработки.

Отдельно, в корневом каталоге ПР, распложены xml-файлы, описывающие тестовые последовательности для TestNg и внешние зависимости для конфигурирования и сборки pom проекта для Maven.

Структура каталогов программного решения представлена ниже:

```
Spotify_API_testing
    src
        main
            java
                pageObjects
                pojoClasses
                utils
        test
            java
                initalDataAndUtils
                testApiClasses
```

3. Описание тестовых классов и последовательности работы тестовых сценариев программного решения

В настоящее программное решение включено 2 тестовых сценария для проверки работы с плейлистами и один тест для проверки корректности создания POJO-объектов и сериализации их в JSON.

В пакете "smokeTests" расположено 2 тестовых сценария для быстрого смоук-тестирования некоторых функций взаимодействия с плейлистами, описанные в тестовых классах:

- PlaylistsTesting;
- NegativeScenarios.

Все тестовые сценарии являются логически взаимосвязанными наборами законченных тест-кейсов и могут запускаться как по отдельности из своих тестовых классов, так и в составе – с учётом последовательностей выполнения, описанных в xml-файлах тестового фреймворка TestNG.

Применяемые в данном программном решении аннотации @BeforeSuite, @BeforeTest, @Test, @AfterTest являются аннотациями фреймворка TestNG если не указано иное.

3.1 Базовый тестовый класс TestBasics

Тестовые сценарии, перечисленные выше, описаны в одноимённых .java классах и являются производными классами от базового класса TestBasics.java, находящегося в пакете "parentTestClass".

Базовый класс TestBasics.java в своём составе имеет поля и методы, общие для производных от него классов тестовых сценариев и содержит:

а)

- статическое поле *spotifyAuthorizationPage* типа String – принимает строку с адресом открываемой веб-страницы авторизации в сервисе Spotify;
- статическое поле *accessTokenUrl* типа String – принимает строку с эндпойнтом для получения токена доступа посредством взаимодействия с API;
- статическое поле *scope* типа String – принимает строку с описанием прав доступа пользователя к возможностям сервиса;
- статическое поле *appRedirectUri* типа String – принимает строку с веб-адресом перенаправления пользователя в стороннее приложение после входа в учётную запись;
- статическое поле *appClientId* типа String – принимает значение идентификатора клиента, сгенерированное при регистрации стороннего приложения, используемого для тестирования API. Используется для авторизации пользователя на сервисе Spotify через стороннее приложение;
- статическое поле *appClientSecret* типа String – принимает сгенерированное при регистрации стороннего приложения для тестирования API значение. Используется для получения токена доступа (access_token) к ресурсам сервиса Spotify через зарегистрированное стороннее приложение.

б)

- статическое поле *driver* типа WebDriver – предназначено для задания экземпляра вебдрайвера в используемых экземплярах тестовых классов;
- статическое поле *authorizationPageUri* типа String – предназначено для комбинирования полного адреса открываемой веб-страницы авторизации на сервисе Spotify с query-параметрами для входа;
- статическое поле *spotifyLoginPage* типа SpotifyLoginPage – созданный экземпляр на базе этого класса описывает веб-элементы DOM-структуры веб-страницы входа в учётную запись сервиса, принимаемой браузером во время работы тестового сценария. Класс, описывающий данное поле находится в каталоге "pageObjects" модуля "main". Подробное описание класса приведено далее в соответствующем разделе.

в)

- статические поля *accountUserName*, *accountPassword* типа String – принимают значения имени пользователя и пароля для входа в существующую учётную запись сервиса Spotify.

Значения для полей, указанных в пунктах "а", "б" (для поля driver) и "в" принимаются из файла InputData.java, содержащего исходные данные для работы тестовых сценариев. Подробнее описаны далее.

- protected TestBasics(){
задание полей из пунктов "а", "в" и экземпляра веб-драйвера;
задание экземпляра веб-драйвера;
} – конструктор при помощи которого задаются значения полей, общих для всех тестовых классов.

Метод *startActions()*, подписанный аннотацией @BeforeSuite, предназначен для выполнения начальных действий по входу в учётную запись пользователя сервиса Spotify и возврата в клиент стороннего приложения для дальнейшей проверки взаимодействия через API. Метод содержит в себе следующие шаги, описанные в последовательности инструкций:

- *spotifyLoginPage = new SpotifyLoginPage(driver)* – активация веб-элементов с которыми будет происходить взаимодействие на открываемой веб-станице входа в учётную запись;
- *authorizationPageUri = spotifyAuthorizationPage* + "?" +
"client_id="+ *appClientId* + "&response_type=code" +
"&scope=" + *scope* + "&redirect_uri=" + *appRedirectUri* – формирование строки URI с параметрами для входа в учётную запись сервиса Spotify и дальнейшего перехода в стороннее приложение;
- *driver.get(authorizationPageUri)* – открытие веб-страницы входа в аккаунт сервиса Spotify по сформированной строке URI;
- {
spotifyLoginPage.setUserName(accountUserName);
spotifyLoginPage.setPassword(accountPassword);
spotifyLoginPage.checkboxControl();
spotifyLoginPage.liginEnter();} – ввод в соответствующие поля имени пользователя и пароля от существующей учётной записи и вход в аккаунт.

Также в составе класса предусмотрены поля типа String для хранения значений, получаемых в предтестовых методах помеченных аннотацией @BeforeTest, необходимые для последующей работы с тестовыми сценариями:

- *authorizationCode* – значение кода авторизации, содержащееся в URI на перенаправляемый веб-ресурс;
- *accessToken* – приходящее в теле ответа значение токена доступа, необходимого для дальнейшей работы из учётной записи с сервисом Spotify;

Для работы сторонней программы с функциями сервиса Spotify предусмотрена авторизация пользователя при помощи протокола OAuth2.0. Результатом авторизации является получение специального уникального токена доступа (access_token), предоставляемого сервисом Spotify пользователю стороннего приложения, через которое осуществляется взаимодействие. Автоматизация процесса получения токена доступа описана в тестовых предтестовых методах get1_AuthorizationCode() и get2_AccessToken().

Предтестовый метод *get1_AuthorizationCode()*, подписанный аннотацией @BeforeTest, предназначен для получения кода авторизации, передаваемого сервером сервиса Spotify в качестве query-параметра в сформированном URI на переадресуемый веб-ресурс. Метод состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в учётную запись пользователя сервиса Spotify.

- *Assert.assertTrue(driver.getCurrentUrl().contains(appRedirectUri))* – проверка, что после успешного входа в учётную запись пользователя сервиса Spotify осуществляется переадресация и переход по требуемому веб-адресу;

- *authorizationCode = driver.getCurrentUrl().split("code=")[1]* – разделение строки перенаправляемого Url и получение из неё кода авторизации пользователя (authorization_code). Далее вывод на консоль полученного кода авторизации.

Предтестовый метод *get2_AccessToken()*, подписанный аннотацией @BeforeTest, предназначен для получения токена доступа и дальнейшей возможности осуществления действий на сервисе Spotify от имени пользователя. Метод состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлено перенаправление на веб-ресурс стороннего приложения, получен authorization_code.

- *RestAssured.baseUri = "https://accounts.spotify.com"* – задание веб-адреса с которым будет осуществляться взаимодействие через API*;*

- *String response* – создание локальной переменной куда будет передан пришедший ответ на сформированный http-запрос;

- формирование запроса*:*

response = given().log().all().
header("Content-type", "application/x-www-form-urlencoded").
header("Authorization", "Basic" + encodedString).
//*encodedString = Base64Coder.encodeToBase64(appClientId, clientSecret)* – закодированная по стандарту Base64 строка, содержащая в себе внешний и секретный идентификаторы стороннего приложения в формате "client_id:client_secret"
formParam("grant-type", "authorization-code").
formParam("code", authorizationCode).
formParam("redirect_uri", appRedirectUri)
//query-параметры, указываемые после эндпойнта для формирования POST-запроса
.*when().post("/api/token")* – метод запроса и его эндпойнт
.*then().log().all().assertThat().statusCode(200).extract().response().*
asString(); – формирование вида принимаемого ответа с заданием проверки принятого от сервера кода состояния, сохранение принятого JSON в переменной response в виде строки

accessToken = JsonConverter.stringToJson(response).
getString("access_token") – получение токена доступа access_token путём выделения его из тела ответа при помощи статического метода *stringToJson()*.

Метод *endOfSession()*, подписанный аннотацией @AfterTest, предназначен для закрытия экземпляра веб-драйвера по завершении работы тестовых сценариев.

3.2 Описание тестовых сценариев для смоук-тестирования работы API в каталоге "smokeTests"

Все http-запросы, участвующие в данном тестировании API, формируются при помощи библиотеки RestAssured по схеме given() -> when() -> then() где:

- given() – формирование запроса, описание передаваемых параметров;
- when() – задание метода и эндпойнта запроса;
- then() – задание действий и описания принимаемого ответа.

Тестовыми сценариями предусмотрено формирование body запросов как в готовом для передачи JSON в текстовом формате, так и при создании POJO с различными данными и дальнейшей сериализацией в JSON.

3.2.1 PlaylistsTesting

Тестовый сценарий PlaylistsTesting описан в классе PlaylistsTesting.java и представляет собой базовый сценарий создания плейлиста, добавления и удаления песен из него, проверки актуального состояния после каждого внесения изменений в созданный плейлист посредством взаимодействия с API и включает в себя следующие шаги:

- получение информации о текущем пользователе;
- создание нового приватного плейлиста в учётной записи пользователя;
- добавление нескольких музыкальных треков в созданный плейлист;
- изменение названия и краткого описания созданного плейлиста;
- получение информации об имеющихся у пользователя плейлистах;
- получение информации о музыкальных треках в созданном плейлисте;
- добавление ещё нескольких треков в созданный плейлист;
- получение информации об актуальном состоянии созданного плейлиста;
- удаление нескольких позиций из созданного плейлиста;
- получение информации об актуальном состоянии созданного плейлиста.

В составе класса есть поля типа String в которые передаются некоторые значения, принимаемые в ответах на запросы из тест-кейсов, необходимые для последующей работы с тестовым сценарием:

- *userId* – уникальный идентификатор пользователя сервиса Spotify;
- *userName* – имя пользователя сервиса Spotify;
- *createdPlaylistId* – уникальный идентификатор создаваемого плейлиста;
- *snapshotId* – идентификатор состояния плейлиста.

Поле *amountOfSongsInPlaylist* типа int предназначено для передачи в него информации о количестве песен в интересующем плейлисте и дальнейшей обработки этого значения.

Тестовый метод *getCurrentUserProfile(),* подписанный аннотацией @Test, предназначен для проверки получения информации о текущем пользователе в JSON-формате и выделении из него идентификатора пользователя. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token.
baseURI = "https://api.spotify.com/v1" – задание веб-адреса с которым будет осуществляться взаимодействие через API;

```
String response = given().header("Authorization", "Bearer " + accessToken)
        .when().get("/me")
        .then().log().all().assertThat().statusCode(200).extract().response().
                                                                asString();
userId = JsonConverter.stringToJson(response).getString("id");
userName = JsonConverter.stringToJson(response).getString("display_name");
```

Дальнейший вывод на консоль имени и идентификатора текущего пользователя.

Тестовый метод *createPlaylist(),* подписанный аннотацией @Test, предназначен для проверки возможности создания плейлиста для текущего пользователя. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token.
baseURI = "https://api.spotify.com/v1";
String response = given().header("Authorization", "Bearer " + accessToken).
header("content_type", "application_json").
body(
// сюда вписывается тело запроса в JSON-формате, предусмотренном для данного запроса и содержащим в себе поля:
"name", "description", "public", "collaborative"
Для формирования уникального названия плейлиста используется генератор случайных чисел, входящий в стандартную библиотеку Java
(int)(Math.random()*2024)
)
.when().post("users/" + userId + /playlists)
.then().log().all().assertThat().ststusCode(201).extract().response().asString();
createdPlaylistId = JsonConverter.stringToJson().getString("id");
snapshotId = JsonConverter.stringToJson(response).getString("snapshot_id");
Дальнейший вывод на консоль идентификатора созданного плейлиста.

Тестовый метод *addItemsToPlaylist()*, подписанный аннотацией @Test, предназначен для проверки возможности добавления нескольких песен в созданный плейлист. Добавление песен происходит путём отправления POST-запроса с передаваемым им телом в JSON-формате, включающим идентификаторы добавляемых песен. Текстовый файл требуемого формата сформирован заранее. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, создан новый плейлист и получен его идентификатор (*createdPlaylistsId*) .
baseURI = "https://api.spotify.com/v1";
given().log().all().
header("Authorization", "Bearer " + accessToken).
header("content_type", "application_json").
body(new String(Files.readAllBytes(Paths.get("path_to_file/addSongs.json"))))
//чтение тела запроса из текстового файла addSongs.json
.when().post("/playlists/" + createdPlaylistsId + "/tracks").
.then().log().all().assertThat().statusCode(200);
Проверка принятого от сервера кода состояния.

Тестовый метод changePlaylistDetails(), подписанный аннотацией @Test, предназначен для проверки возможности изменения основной информации о плейлисте: названия плейлиста, его краткого описания и его публичного статуса. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, известен идентификатор плейлиста, подлежащего изменения.
baseURI = "https://api.spotify.com/v1";
given().log().all().
header("Authorization", "Bearer " + accessToken).
header("content_type", "application_json").
pathParam("playlist_id", createdPlaylistId).
body(
//сюда вписывается тело запроса в JSON-формате, предусмотренном для данного запроса и содержащим в себе поля:
"name", "description", "public"*)*
.when().put("/playlists/{playlist_id}")
.then().log().all().statusCode(200);

Тестовый метод *getUserPlaylists()*, подписанный аннотацией @Test, предназначен для проверки возможности получения информации о плейлистах, имеющихся у пользователя. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, известен идентификатор пользователя (*userId*).

```
baseURI = "https://api.spotify.com/v1";
String response = given().log().all().
        header("Authorization", "Bearer " + accessToken)
        .when().get("/users/" + userId + "/playlists")
        .then().log().status().assertThat().statusCode(200).extract().response().
                                                                asString();
```

//Принимаемый в ответе JSON не выводится на консоль в виду его избыточной информативности.

Проверка принятого от сервера кода состояния.

int amountOfPlaylists = JsonConverter.stringToJson(response).
getInt("items.size()"); – создание локальной переменной amountOfPlaylists, которая принимает количество имеющихся у пользователя плейлистов путём чтения принятого JSON.

Вывод информации о количестве имеющихся упользователя плейлистов на консоль.

```
for(int i = 0; i < amountOfPlaylists; i++){
System.out.println(
JsonConverter.stringToJson(response).getString("items[" + i + "].name" + "\tID " +
JsonConverter.stringToJson(response).getString("items[" + i + "].id );
```

} – вывод на консоль информации о названиях плейлистов и их идентифткаторах путём чтения в принятом JSON полей items.name и items.id.

Тестовый метод *getPlaylist(),* подписанный аннотацией @Test, предназначен для проверки корректности содержимого созданного плейлиста. Сверяется фактическое количество песен в плейлисте с первоначально добавленными треками в количестве 5 позиций. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, создан новый плейлист, в него добавлено 5 песен.

```
baseURI = "https://api.spotify.com/v1";
String response = given().header("Authorization", "Bearer " + accessToken).
        queryParam("fields", "items(track(name, album(name), artists(name))")
        //задание формы ответа. Принимаемое тело JSON содержит в себе поля, указанные в значении параметра запроса
        .when().get("/playlists/" + createdPlaylistId + "/tracks")
        .then().log().all().assertThat().statusCode(200).extract().response().
                                                                asString();
amountOfSongsInPlaylist = JsonConverter.stringToJson(response).
                                                        getInt("items.size()");
Assert.assertEquals(amountSongsInPlaylist, 5);
```

Тестовый метод *addItemsToPlaylistQueryParams(),* подписанный аннотацией @Test, предназначен для проверки добавления нескольких треков в плейлист путём передачи их идентификаторов в качестве параметров запроса. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, создан новый плейлист либо обновляется уже существующий.

baseURI = "https://api.spotify.com/v1";

String response = given().header("Authorization", "Bearer " + accessToken).

queryParam("position", amountOfSongsInPlaylist). – параметр, указывающий на позицию добавляемых треков (в данном случае в конец плейлиста)

queryParam("uris", "spotify:track:some_song1_ID,

spotify:track:some_song2_ID ")

//URI треков в системе Spotify в форме:

spotify:track:идентификатор_песни

.when().post("/playlists/" + createdPlaylistId + "/tracks")

.then().log().all().assertThat().statusCode(200).extract().response().

asString();

snapstotId = JsonConverter.stringToJson(response).getString("snapshot_id");

Тестовый метод *getPlaylist2(),* подписанный аннотацией @Test, предназначен для проверки корректности содержимого плейлиста. Сверяется фактическое количество песен в плейлисте с дополнительно добавленными треками в количестве 2 позиций. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, в созданный плейлист, имеющий 5 песен, добавлено ещё 2 песни.

baseURI = "https://api.spotify.com/v1";

String response = given().header("Authorization", "Bearer " + accessToken).

queryParam("fields", "items(track(name, album(name), artists(name))")

//задание формы ответа. Принимаемое тело JSON содержит в себе поля, указанные в значении параметра запроса

.when().get("/playlists/" + createdPlaylistId + "/tracks")

.then().log().all().assertThat().statusCode(200).extract().response().

asString();

amountOfSongsInPlaylist = JsonConverter.stringToJson(response).

getInt("items.size()");

Assert.assertEquals(amountSongsInPlaylist, 7);

Тестовый метод *removePlaylistsItems()*, подписанный аннотацией @Test, предназначен для проверки функции удаления песен из плейлиста пользователя. Тело запроса на удаление треков формируется путём сериализации экземпляра POJO с идентификаторами песен в JSON. Тест-кейс состоит из следующих шагов, описаных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, в плейлисте содержатся музыкальные треки.

baseURI = "https://api.spotify.com/v1";

RemoveSongsBody removeSongsBody – создание экземпляра POJO-класса, структурой соответствующего запросу JSON в который передаются идентификаторы песен, подлежащих удалению;

removeSongsBody = CreatorRemoveSongsPojo.createPojo("some_song1_ID",

"some_song2_ID", "some_song3_ID", snapshotID);

given().log().all().

header("Authorization", "Bearer " + accessToken).

header("Content-Type", "application/json").

body(removeSongsBody)

.when().delete("/playlists/" + createdPlaylistID + "/tracks")

.then().log().all().assertThat().statusCode(200);

Тестовый метод *getPlaylist3()*, подписанный аннотацией @Test, предназначен для проверки корректности содержимого плейлиста. Принимаемый от сервера ответ из JSON десериализуется в объект POJO. Сверяется фактическое количество песен в плейлисте после удаления трёх музыкальных композиций. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, из созданного плейлиста, имевшего 7 песен, удалено 3 песни.

baseURI = "https://api.spotify.com/v1";

PlaylistItemsBody response – создание экземпляра POJO-класса, структурой соответствующего принимаемому ответу в форме JSON для десериализации;

```
response = given().header("Authorization", "Bearer " + accessToken).
    queryParam("fields", "items(track(name, album(name), artists(name))").
    expect().defaultParser(Parser.JSON)
    .when().get("/playlists/" + createdPlaylistId + "/tracks")
    .then().log().all().assertThat().statusCode(200).extract().
                                                as(PlaylistItemBody.class);
```

amountOfSongsInPlaylist = response.getItems().size() – получение значения количества песен в плейлисте путём просмотра POJO экземпляра*;*

Assert.assertEquals(amountOfSongsInPlaylist, 4) – сравнение фактического количества песен в плейлисте с ожидаемым*;*

Gson gson = new Gson() – создание экземпляра типа Gson из подключаемой в файле конфигурации Maven pom.xml библиотеки Gson для сериализации экземпляра POJO в строчный формат*;*

String json = gson.toJson(response);

JsonConverter.stringToJson(json).prettyPrint(); – вывод на консоль пришедшего ответа в формате JSON*.*

3.2.2 NegativeScenarios

Тестовый сценарий NegativeScenarios описан в классе NegativeScenarios.java и представляет собой набор тест-кейсов, направленных на проверку взаимодействия с API сервиса Spotify путём отправления запросов с некорректными данными. Включает в себя следующие проверки:

- отправка запроса по некорректному эндпоинту;
- попытка получения информации о плейлисте с неверным идентификатором;
- попытка добавления песен в закрытый плейлист другого пользователя сервиса Spotify;
- попытка изменения информации о плейлисте Spotify;
- попытка добавления в плейлист текущего пользователя нескольких песен с некорректными идентификаторами.

Тестовый метод *sendQueryToWrongEndpoint(),* подписанный аннотацией @Test, предназначен для попытки отправки запроса по адресу с некорректно введенным эндпойнтом и дальнейшей проверки приходящего ответа на это действие. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token.

baseURI = "https://api.spotify.com/v1" – задание веб-адреса с которым будет осуществляться взаимодействие через API;

```
String response = given().log().method().log().uri().
    header("Authorization", "Bearer " + accessToken)
    .when().get("/mee") – эндпойнт отправки запроса введён некорректно
    .then().log().status().log().body().assertThat().statusCode(404).
```

extract().response().asString();
String errorMessage = JsonConverter.stringToJson(response).
getString("error.message");
Assert.assertEquals(errorMessage, "Servise not found");
//Проверка принятого от сервера кода состояния и сообщения об ошибке.

Тестовый метод *getWrongPlaylist()*, подписанный аннотацией @Test, предназначен для попытки отправки запроса на получение содержимого плейлиста с некорректно указанным идентификатором и дальнейшей проверки приходящего ответа на это действие. Идентификаторы плейлистов в системе Spotify представляют собой строки из 22 символов, используемых в схеме кодировки Base62 и начинающиеся с цифрового символа от 0 до 9. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token.
String wrongPlaylistId = "123SomeWrongPlaylistId" – создание локальной переменной с присвоением значения из 22 символов некорректного идентификатора плейлиста;
baseURI = "https://api.spotify.com/v1";
String response = given().log().method().log().uri().
header("Authorization", "Bearer " + accessToken).
queryParam("fields", "items(track(name),album(name))") – параметры ответа
.when().get("/playlists/" + wrongPlaylistId + "/tracks")
.then().log().status().log().body().assertThat().statusCode(502).
extract().response().asString();
String errorMessage = JsonConverter.stringToJson(response).
getString("error.message");
Assert.assertEquals(errorMessage, "Error while loading resource");
//Проверка принятого от сервера кода состояния и сообщения об ошибке

Тестовый метод *addSongsToPrivatePlaylist()*, подписанный аннотацией @Test, предназначен для попытки добавления песен в существующий закрытый плейлист другого пользователя системы Spotify и проверки приходящего ответа на это действие. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, известен идентификатор закрытого плейлиста другого пользователя.
String privatePlaylistId – создание локальной переменной с присвоением строки из 22 символов существующего плейлиста другого пользователя. В данном программном решении совершается попытка изменения плейлиста "Spotify Web API Testing playlist";
baseURI = "https://api.spotify.com/v1";
String response = given().log().method().log().uri().log().parameters().
header("Authorization", "Bearer " + accessToken).
queryParam("uris","spotify:track:some_song1_ID,
spotify:track:some_song2_ID")
.when().post("/playlists/" + privatePlaylistId + "/tracks")
.then().log().status().log().body().assertThat().statusCode(403).
extract().response().asString();
String errorMessage = JsonConverter.stringToJson(response).
getString("error.message");
Assert.assertEquals(errorMessage, "Forbidden");
//Проверка принятого от сервера кода состояния и сообщения об ошибке

Тестовый метод *changeSpotifyPlaylistDetails()*, подписанный аннотацией @Test, предназначен для попытки изменения основной информации о плейлисте другого пользователя: названия плейлиста, его краткого описания и его публичного статуса. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, известен идентификатор плейлиста, в который осуществляется попытка изменения.
String spotifyPlaylistId – создание локальной переменной в которую присваивается значение идентификатора плейлиста другого аккаунта. В данном случае присваивается аккаунт стандартного плейлиста Spotify "Rock Classics"
baseURI = "https://api.spotify.com/v1";
String response = given().log().method().log().uri().log().headers().log().body().
header("Authorization", "Bearer " + accessToken).
header("content_type", "application_json").
body(
//сюда вписывается тело запроса в JSON-формате, предусмотренном для данного запроса и содержащим в себе поля:
"name", "description", "public"*)*
.when().put("/playlists/" + spotifyPlaylistId)
.then().log().status().log().body().assertThat().statusCode(403).extract().
response().asString();
String errorMessage = JsonConverter.stringToJson(response).
getString("error.message");
Assert.assertEquals(errorMessage, "Not allowed");
//Проверка принятого от сервера кода состояния и сообщения об ошибке

Тестовый метод *addSongsWithWrongID()*, подписанный аннотацией @Test, предназначен для попытки добавления в существующий плейлист пользователя нескольких песен, идентификатор одной из которых не корректный. Идентификаторы песен в системе Spotify закодированы по схеме Base62, состоят из 22 символов начинаются с цифрового символа от 0 до 9. Тест-кейс состоит из следующих шагов, описанных в последовательности инструкций:

Предусловие: осуществлён вход в аккаунт, получен access_token, выбраны 2 песни для добавления в плейлист, первая стоящая в запросе имеет реальный идентификатор, вторая – некорректный, 21 символ.

Данный тест-кейс состоит из трёх действий:
1 – передаётся запрос на получение информации о текущем пользователе, из присланного на него респонса выбирается его идентификатор;
2 – передаётся запрос на получение информации о плейлистах пользователя, идентификатор которого получен, из присланного респонса выбирается идентификатор первого по списку плейлиста;
3 – передаётся запрос на добавление двух песен в выбранный плейлист.

String songToAddId1 = "22charsCorrectSongID22";
String songToAddId2 = "21charIncorrectSongID" – создание локальных переменных с корректным 22 символа и некорректным 21 символ идентификатором песен;
baseURI = "https://api.spotify.com/v1";
String userIdResponse = given().log().method().log().headers().
header("Authorization", "Bearer " + accessToken)
.when().get("/me")
.then().assertThat().statusCode(200).extract().response().asString();
String userId = JsonConverter.stringToJson(userIdResponse).getString("id");
Вывод на консоль значения идентификатора пользователя;

```
String playlistsResponse = given().log().method().log().headers().
        header("Authorization", "Bearer " + accessToken)
        .when().get("/users/" + userId + "/playlists")
        .then().assertThat().statusCode(200).extract().response().asString();
String playlistId = JsonConverter.stringToJson(playlistsResponse).
                                                        getString("items[0].id");
Вывод на консоль значения идентификатора пользователя;

String response = given().log().method().log().parameters().log().headers().
        header("Authorization", "Bearer " + accessToken).
        queryParam("position", 0).
        queryParam("uris", "spotify:track:" + songToAddId1 +
                                        ", spotify:track:" + songToAddId2).
        .when().post("/playlists/" + playlistId + "/tracks")
        .then().log().status().log().body().assertThat().statusCode(400).extract().
                                                        response().asString();
String errorMessage = JsonConverter.stringToJson(response).
                                                    getString("error.message");
Assert.assertEquals(errorMessage, "Invalid base62 id");
//Проверка принятого от сервера кода состояния и сообщения об ошибке
```

3.3 Тестовый класс ValidatePojoTests

Тестовый класс ValidatePojoTests предназначен для проверки корректности создания POJO-объектов, имеющих: а) структуру тела запроса для тестового метода *removePlayItems()* с дальнейшей его сериализацией в JSON-формат; б) структуру тела ответов для тестового метода *getPlaylists3()* для дальнейшей десериализации из JSON-формата.

Метод *checkingRemoveRequestPojoVer1()* предназначен для составления POJO-объекта со строчными данными, имеющего структуру JSON-тела запроса на удаление песен из плейлиста. Метод обрабатывает фиксированное значение количества удаляемых песен, передаваемых в качестве аргументов.

Метод *checkingRemoveRequestPojoVer2()* предназначен для составления POJO-объекта со строчными данными, имеющего структуру JSON-тела запроса на удаление песен из плейлиста. Метод обрабатывает различное количество удаляемых песен, передаваемых в массиве строковых данных.

Структура JSON-тела запроса на удаление песен из плейлиста:

```
{
    "tracks": [
                { "uri": "spotify:track:идентификатор_песни_1" },
                { "uri": "spotify:track:идентификатор_песни_2" },
                …
                { "uri": "spotify:track:идентификатор_песни_n" }
        ],
    "snapshot": "идентификатор_состояния_плейлиста"
}
```

Метод *createRequestPojo()* предназначен для составления POJO-объекта со строчными данными, имеющего структуру JSON-тела принимаемого респонса с описанием всех песен, входящих в интересующий плейлист. Метод может принять и обработать различное количество поступающих в него данных, соответствующей нижеприведенной структуре JSON-тела:

```
{
        "items": [
                {
                    "track": {"album": {"name": "Название_альбома_1"}},
                     "artists": [
                                {"name": "Исполнитель_1"},
                                {"name": "Исполнитель_2"}
                    ],
                     "name": "Название_песни_1"
                },
                {
                    "track": {"album": {"name": "Название_альбома_2"}},
                     "artists": [
                                {"name": "Исполнитель_1"},
                    ],
                     "name": "Название_песни_2"
                },
                {
                    "track": {"album": {"name": "Название_альбома_3"}},
                     "artists": [
                                {"name": "Исполнитель_1"},
                                …
                                {"name": "Исполнитель_n"}
                    ],
                     "name": "Название_песни_3"
                }
               ]
}
```

Методы *checkingRemoveRequestPojoVer1(), checkingRemoveRequestPojoVer2(), createRequestPojo()* непосредственно в процессе тестирования не задействованы.

3.4 Описание файлов управления группами тестов типа testNG.xml

В настоящем программном решении разработан файл управления тестами в котором описана последовательность запуска тестовых сценариев.

Файл smokeTests.xml объединяет в себе запуск тестовых сценариев, описанных в тестовых классах PlaylistTesting и NegativeScenarios. Первыми запускаются 10 методов на проверку позитивных тест-кейсов, следующими – 5 методов негативного тестирования web API сервиса Spotify.

3.5 Каталог "initialDataAndUtils"

Каталог "initialDataAndUtils" содержит файлы (InputData.java, addSongs.json), с необходимыми исходными данными для тестирования API сервиса Spotify по приведенным выше тестовым сценариям, а также класс JsonConverter (JsonConverter.java) для преобразования строковых данных в тело запроса/ответа JSON-формата типа JsonPath библиотеки RestAssured.

В подкаталоге "payload" находится текстовый файл addSongs.json в котором содержится тело запроса в формате JSON на добавление пяти песен в плейлист. Содержимое файла используется в тест-кейсе, описанном в методе *addItemsToPlaylist()* тестового класса PlaylistsTesting.

3.5.1 Класс с исходными данными для тестирования InputData

Класс InputData хранит в себе все исходные данные, необходимые для тестирования API сервиса Spotify по данному программному решению.

Класс содержит в себе следующие приватные статические неизменяемые поля строкового типа:

- *SPOTIFY_AUTHORIZATION_URL* – веб-адрес страницы входа в учётную запись пользователя Spotify;
- *ACCESS_TOKET_URL* – веб-адрес к которому происходит запрос на получение токена доступа;
- *SCOPE* – содержит описание прав доступа пользователя для совершения действий по созданию и модификации плейлистов;
- *APP_REDIRECT_URI* – адрес по которому расположен веб-ресурс стороннего приложения которое предназначено для взаимодействия с системой сервиса Spotify посредством web API. По умолчанию задан адрес перенаправления на сервис тестирования авторизации и аутентификации приложения Postman;
- *APP_CLIENT_ID* – идентификатор стороннего приложения (client_id), зарегистрированного в системе Spotify;
- *APP_CLIENT_SECRET* – секретный идентификатор приложения (client_secret), зарегистрированного в системе Spotify;
- *USER_NAME* – имэйл-адрес пользователя, привязанный к учётной записи сервиса Spotify;
- *PASSWORD* – пароль доступа к учётной записи пользователя сервиса Spotify;
- *BROWSER_TYPE* – тип веб-браузера, автоматизация действий которого применяется в работе с данными тестовыми сценариями;
- *WEBDRIVER_PATH* – путь доступа к утилите вебдрайвера в памяти компьютера на котором происходит запуск данного программного решения.

Для обращения к данным, содержащимся в вышеперечисленных полях, используются методы-геттеры, вызываемые при необходимости в ходе выполнения тестовых сценариев.

4. Модуль "main". Описание работы методов и алгоритмов, используемых для работы тестовых сценариев

Рабочий модуль "main" включает в себя .java классы, обеспечивающие логику программного решения и взаимосвязь выполнения тестов.

В состав модуля "main" входят каталоги "pageObjects", "pojoClasses" и "utils".

В каталоге "pageObjects" находятся классы, содержащие локаторы для веб-элементов DOM, принимаемой браузером страницы веб-сайта Spotify с которой происходит взаимодействие во время автоматизированного входа в аккаунт и методы для работы с этими веб-элементами.

В каталоге "pojoClasses" находятся классы для создания POJO-объектов, предназначенных для формирования структуры тел http- запросов и ответов, реализуемые в тестовых методах.

Каталог "utils" содержит классы для вызова и установки экземпляра вебдрайвера, а так же кодировки данных для http-запросов.

4.1 Каталог "pageObjects"

Каталог "pageObjects" содержит в себе файлы классов PageObjectBasics и SpotifyLoginPage.

Для определения и инициализации веб-элементов подключен PageFactory, входящий в состав библиотеки Selenium. Все используемые в работе веб-элементы помечены

аннотацией @FindBy. В в качестве локаторов для веб-элементов выбрано определение элементов по Xpath как наиболее стабильный и неизменчивый со временем.

4.1.1 PageObjectBasics

Класс PageObjectBasics является суперклассом для класса описания элементов веб-страницы входа в аккаунт SpotifyLoginPage и предназначен для задания основных членов, общих для дочернего класса.

В своём составе суперкласс имеет поле типа WebDriver из библиотеки Selenium для задания которого составлен конструктор, принимающий на вход экземпляр вебдрайвера. Задание PageFactory для отложенной инициализации вебэлементов также предусмотрено в конструкторе.

Суперкласс содержит в себе следующие методы:

- *WebElement waitForVisibility(WebElement webElement)* – предназначен для задания явного ожидания появления веб-элементов в DOM и дальнейшей работе с ними;
- *WebDriver getDriverInstance()* – предназначен для получения экземпляра вебдрайвера, при необходимости.

4.1.2 SpotifyLoginPage

Класс LoginPage содержит в себе логику для работы на веб-странице входа в аккаунт. Экземпляры, созданные на основе этого класса входят в состав тестовых сценариев PlaylistsTesting и NegativeScenarios. В состав класса входят следующие приватные поля типа WebElement, подписанные аннотацией @FindBy:

- userNameField – веб-элемент формы ввода имени пользователя;
- passwordFiled – веб-элемент формы ввода пароля для входа в аккаунт;
- loginButton – кнопка входа в учётную запись;
- loginCheckbox – чекбокс для запоминания данных входа.

Методы класса:

- *void setUserName(String loginEmail){*
  userInputFiled.clear();
  userInputFiled.sendKeys(loginEmail);
  } – ввод имени пользователя;

- *void setPassword(String loginEmail){*
  passwordInputFiled.clear();
  passwordInputFiled (loginPassword);
  } – ввод пароля от аккаунта;

- *void setCheckboxOff(){*
  loginCheckbox.click();
  } – выключение чекбокса;

- *void loginEnter(){*
  loginButton.click();
  } – нажатие на кнопку входа в аккаунт.

4.2 Каталог "pojoClasses"

Каталог "pojoClasses" содержит в себе пакеты "playlistItemsResponse" и "removeSongRequest" с классами для создания POJO-объектов, реализующих структуру тел получаемого от сервера ответа и запроса соответственно.

4.2.1 Пакет "playlistItemsResponse"

Пакет "playlistItemsResponse" содержит в себе классы, описывающие и создающие POJO-объект для десериализации данных, поступающих от сервиса во время выполнения тестового метода *getPlaylist3()* тестового сценария PlaylistsTesting.

Структура тела принимаемого в JSON-формате респонса представлена в разделе 3.3 и имеет следующие поля:

- items[]{}
  - track{}
    - album{}
      - name
    - artists[]{}
      - name
    - name

Данные поля описаны в классах PlaylistItemsBody -> Items -> Tracks -> Album, Artists.

Создание POJO-объекта описанной структуры осуществляется в классе CreatorPlaylistItemsPojo. Данный класс содержит в себе статический метод *createPojo(String[][] songItems)* – метод принимает двумерный массив строчного типа, содержащий в себе пересылаемые значения полей в следующей последовательности:
Strring[]{"название_песни", "название_альбома", "исполнитель_1", … "исполнитель_n"}
и возвращает POJO-объект типа PlaylistsItemsBody.

```
public static PlaylistItemsBody createPojo(String[][] songItems){
//Создание экземпляра POJO и списка типа Items, для помещения в этот экземпляр
    PlaylistItemsBody playlistItemsBody = new PlaylistItemsBody();
    List<Items> items = new ArrayList<>();

//Цикл с количеством итераций равному длине принимаемого двумерного массива (соответствует количеству песен в обрабатываемом плейлисте)
    for(int i = 0; i < songItems.length; i++){
//Создание экземпляров классов, соответствующих полям содержащимся внутри items
        Items item = new Items();
        Track track = new Tracks();
        Album album = new Album();
        List<Artists> artistsList = new ArrayList<>();
//Задание значений из принимаемого методом массива
        track.setName(songItems[i][0]) – задание названия песни;
        album.setName(songItems[i][1]) – задание названия альбома;
//Внутренний цикл, читающий оставшиеся принятые значения (имена исполнителей)
        for(int j = 2; j < songItems[i].length; j++){
//Создание экземпляра, принимающего имя исполнителя песни в каждой итерации цикла
            Artists artist = new Artists();
            artist.setName(songItems[i][j]);
            artistsList.add(artist);
        }
        track.setArtists(artistsList);
        track.setAlbum(album);

        item.setTracks(track);
        items.add(i, item);
    }
    playlistItemsBody.setItems(items);
    return playlistsItemsBody;
}
```

4.2.2 Пакет "removeSongsRequest"

Пакет "removeSongsRequest" содержит в себе классы, описывающие и создающие POJO-объект для дальнейшей сериализации в JSON-формат тела запроса при выполнении тестового метода *removePlaylistItems()* тестового сценария PlaylistsTesting.

Структура тела отправляемого в JSON-формате http-запроса метода DELETE представлена в разделе 3.3 и имеет следующие поля:

- tracks[]{}
  - uri{}
- snapshot

Данные поля описаны в классах RemoveSongsBody -> Tracks.

Создание POJO-объекта описанной структуры осуществляется в классе CreatorRemoveSongsPojo. Данный класс содержит в себе статический метод *createPojo(String[] songId, String snapshotId)* – метод принимает массив строчного типа, содержащий в идентификаторы песен, подлежащих удалению, и строку состояния плейлиста и возвращает POJO-объект типа RemoveSongsBody.

```
public static RemoveSongsBody createPojo(String[] songId, String snapshotId){
//Создание экземпляра POJO и списка типа tracks, для помещения в этот экземпляр
    RemoveSongsBody removeSongsBody = new RemoveSongsBody();
    List<Tracks> tracks = new ArrayList<>();

//Цикл с количеством итераций равному длине принимаемого массива (соответствует количеству удаляемых песен)
    for(int i = 0; i < songId.length; i++){
//Создание экземпляров классов, соответствующих полям содержащимся внутри tracks
        Tracks trackUri = new Tracks();
        trackUri.setUri(songsId[i]);
        tracks.add(i, trackUri);
    }
    removeSongsBody.setTracks(tracks);
    removeSongsBody.setSnapshot(snapshotId);
    return removeSongsBody;
}
```

4.3 Каталог "utils"

Каталог "utils" содержит в себе файлы классов Base64corder и WebDriverSingleton, которые предназначены для вспомогательных действий во время работы программного решения.

4.2.1 Base64coder

Класс Base64coder предназначен для кодирования строковых данных в формате Base64 и дальнейшей передачи закодированного значения в виде строки. Кодировщик необходим в предтестовом методе get2_AccessToken() для формирования заголовка http-запроса на получение токена доступа.

В состав класса входит поле типа Base64.Encoder, входящего в стандартную библиотеку утилит Java, внутренние методы которого могут обрабатывать строковые данные.

Два строковых значения передаются в метод класса и преобразовываются в закодированную строку:

```
public static String encodeToBase64 (String string1, String string2){
    String initalString = string1 + ":" + string2 – создание строки;
    Base64.Encoder encoder = Base64.getEncoder();
    String encodedString = encoder.encodeToString(initialString.getBytes());
return encodedString; }
```

4.2.2 WebDriverSingleton

Класс WebDriverSingleton разработан для настройки единственного экземпляра вебдрайвера используемого в работе веб-браузера и передачи этого экземпляра всем тестовым классам.

В составе класса есть статический метод

static WebDriver setupDriver(String browserType, String driverPath) который в качестве аргументов принимает значение названия рабочего веб-браузера и значение адреса нахождения утилиты вебдрайвера в постоянной памяти рабочего компьютера. При условии отсутствия загруженного вебдрайвера создаётся его экземпляр по входным параметрам, настраиваются таймауты неявных ожиданий, загрузки страницы и скриптов; тут же задаётся размер рабочего окна открываемого веб-браузера.

5 Использование программного решения Spotify_API_testing

Для работы со всеми тестовыми сценариями, предусмотренными настоящим программным решением пользователю необходимо иметь идентификатор стороннего приложения (client_id) и секретный идентификатор приложения (client_secret), а также зарегистрированную учётную запись в сервисе Spotify.

Зарегистрировать учётную запись можно на веб-странице

https://spotify.com/signup

или нажать на кнопку "Зарегистрироваться" на стартовой странице веб-сайта сервиса Spotify.

Этапы регистрации стороннего приложения для получения идентификатора приложения и секретного идентификатора приложения описаны на веб-странице

https://developer.spotify.com/documentation/web-api/concepts/apps

В строку Redirect URI формы регистрации стороннего приложения вписать адрес возврата на веб-ресурс *"https://oauth.pstmn.io/v1/browser-callback"*.

В классе InputData в соответствующие поля "*USER_NAME*" и "*PASSWORD*" вписать данные для входа в учётную запись. В поля "*APP_CLIENT_ID*" и "*APP_CLIENT_SECRET*" занести значения полученных идентификатора стороннего приложения и секретного идентификатора приложения. Также в поле "*WEBDRIVER_PATH*" этого класса прописать путь доступа к утилите вебдайвера в памяти на компьютере пользователя; при использовании в работе веб-браузера, отличного от Chrome, в поле "*BROWSER_TYPE*" указать название рабочего веб-браузера (Firefox).