Automation Testing with Kafka RESTful APIs: Real-World Projects Practices

Aila Bogasieru

Acknowledgements

This book would not have been possible without my family, especially my parents.
Special thanks to **Dumitru Bogasieru** and **Ioana Bogasieru**.
In memory of my beloved father, Dumitru Bogasieru, I will carry you in my thoughts and heart every day and forever. I would like to thank you for all your love and affection, support, and education.

Thank you for passing on your critical and analytical thinking to me.
I would like to express my deepest gratitude to my father, whose hard work, honesty, dedication, and determination always inspired me to grow and to be the person I am today, which gave me a great sense of belief in my work, being a constant and great source of strength.

I carry on your genetic heritage, I can see myself in you through your strength, independence, straightforwardness, and decisiveness.

I am extremely grateful for all the moments we have together, and this brings me a lot of joy, but at the same time, it hurts me terribly that there will not be new ones.

I would like to express my deepest gratitude to my mother for sticking with me.
I know it was not easy, I am very thankful for her advices and talks.
This book would not have been possible without their unspoken guidance and inspiration.

This book would not have been possible without all my professional experiences and opportunities, projects and technologies, my co-workers, IT fellows guiding, encouraging, helping, and criticizing me along the way to be better and improved version of myself.

I also acknowledge the help I have received from people who have read sections of the book and provided feedback.

Chapter 1

Introduction

About the Author

With more than a decade of experience as a Software Engineer, I began my journey into test automation in 2010.

My passion for technology has taken me through diverse domains: e-Commerce, transportation, finance, and streaming platforms, where I have helped build scalable, reliable systems with a strong focus on automation and architecture.

About the Book

This book is a hands-on guide for developers and testers who want to work with RESTful APIs, Kafka, and event-driven systems, without getting lost in abstract theory.

It is filled with real-world code, architecture patterns, and automation workflows that I have used across projects aimed at helping you implement practical solutions in complex environments.

I am excited to share what I have learned and continue learning from the amazing tech community.

What You'll Find Inside

- **RESTful API Design** — Learn to structure scalable, maintainable APIs with practical, real-world examples.
- **Kafka Integration** — Get started with real-time event streaming, message brokers, and microservices using Apache Kafka.
- **Live Coding Showcases** — Code from scratch, understand how it works, and grasp the "why" behind each architectural choice.
- **Production-Friendly Practices** — Built-in testing, clean code, and modular design ready for real-world deployment.

Who This Book is For

Whether you are:

- A backend developer looking to level up your architecture game
- A software engineer exploring event-driven systems
- A curious coder who prefers building over theorizing...

...this book has something for you.

Why I Wrote This Book

Automating end-to-end tests is hard when your system uses Kafka, APIs, and multiple databases.

I wanted a way to:

- Trigger real events
- Validate asynchronous flows
- Make it all reproducible

So I built a series of examples and wrapped them into this book.
Available now on Amazon Kindle & Paperback.

Although there is an abundance of documentation and courses available on platforms such as YouTube, Coursera, Udemy, and Pluralsight, few truly demonstrate how to build a reliable and maintainable testing framework.
Even with the latest and powerful AI, capable of gathering, structuring and summarizing the data from multiple sources also capable to learn and adapt, I still think that in order to navigate through the automation seas you need to have a string foundation and structured thinking to manage successfully any software engineering product especially for technologies such as Restful APIs and Kafka so popular now days.

Kafka is such a powerful technology, used by big companies for real-time data streaming, event-driven architectures, and data integration.
It is worth mentioning few tech companies using Kafka technology:

- LinkedIn – Originally developed by Kafka and continues to use it extensively.
- Netflix – Uses Kafka for real-time monitoring and operational data pipelines.
- Uber Handles high-throughput messaging between microservices with Kafka.
- Airbnb – Employs Kafka for logging, metrics, and analytics pipelines.
- Goldman Sachs – Builds event-driven architectures and data pipelines using Kafka.
- JPMorgan Chase – Uses Kafka for real-time fraud detection and trade monitoring.
- PayPal – Processes and analyzes financial transactions in real time with Kafka.
- Walmart – Leverages Kafka for real-time inventory and analytics.
- Alibaba – Processes billions of messages daily through Kafka.
- BMW – Utilizes Kafka in their IoT platform for connected cars.
- Spotify – Integrates Kafka for real-time data processing and integration.

Kafka is also widely used across industries for log aggregation, stream processing, fraud detection, and real-time analytics.

Keeping it simple is essential.
Adding unnecessary complexity can hinder the effectiveness of your automation framework.
My technology stack primarily consists of Java, Selenium (for web applications), Rest Assured (for RESTful APIs), JUnit/TestNg (as testing frameworks), Apache Maven (as the build tool), and CI/CD tools like Jenkins. Over the past four years, there has been a notable shift toward Python, leading to a decline in Java's popularity. The pandemic further accelerated the demand for AI-driven projects, redefining automation
practices. Concurrently, "low-code" platforms such as Katalon and ReadyAPI
have emerged, yet they often fall short in delivering the necessary speed, robustness, and reliability. These platforms exhibit significant limitations, particularly the absence of object-oriented programming (OOP)
capabilities, which are crucial for creating scalable and efficient test automation solutions. In the first years of automation testing, automation frameworks were built from scratch (Selenium was created later on), so every click, list, button, and text field needed to be created, so more time and effort were put into the automation testing frameworks. As technology continues to evolve, we see enhancements in performance, data handling, and user-friendly interfaces for web applications. This evolution extends beyond web applications to encompass desktop and mobile applications, as well as RESTful APIs, broadening the scope of automation from basic frameworks to more scalable and robust solutions.

In the first years of automation testing, automation frameworks were built from scratch (Selenium was created later on), so every click, list, button, and text field needed to be created, so more time and effort were put into the automation testing frameworks. Once Selenium gained traction and popularity because having amazing capabilities to save a lot of hard work on coding, bringing solutions to handle multi browsers, languages, among others, it was clearly a step ahead for automation testing.

In my humble opinion, it is more important to focus on the logic behind writing tests than on the specific framework or programming language used, specific framework or programming language, but in the underlying logic of writing robust tests. While the choice of technology is indeed vital, it is equally important to architect automation frameworks that are streamlined and efficient, leveraging the capabilities of various libraries.

Even though we are living in a society where technology and speed are playing an important role, just a few clicks and we have our products delivered, streaming platforms available (no need to actually go to a cinema theater), basically we have applications to solve almost any requirement, we should not forget and be aware about the quality of the software products that could be significantly impacted.

Another important factor to be considered in the product's quality might be the shift from office work to remote work and this resulted in cost-efficiency alternatives by hiring people all over the world, not that skilled and if you include also the AI increasing popularity resulting the projects' trajectory.

Part 1: Introduction to Automation Testing

- **Chapter 1: Understanding the Basics of Automation Testing**
  - Benefits of automation testing
  - Types of automation testing (unit, regression, system, end-to-end and integration)
  - Challenges and misconceptions

- **Chapter 2: Choosing the Right Automation Framework**
  - Popular frameworks (Selenium and Cypress)
  - Factors to consider when selecting a framework
  - Case study of successful framework implementations for web app

- **Chapter 3: API Testing**
  - RESTful APIs automated tests

Chapter 3

Part 2: Practical Automation Techniques

- **Chapter 4: Setting Up Your Automation Environment**
  - Installing necessary tools and libraries
  - Configuring your testing environment

- **Chapter 5: Creating Effective Test Scripts**
  - Writing maintainable and reusable test scripts
  - Best practices for test script design

- **Chapter 6: Continuous Integration and Continuous Delivery (CI/CD)**
  1. Integrating automation testing into CI/CD
  2. Benefits of CI/CD
  3. Best practices for CI/CD implementation

- **Chapter 7: Low-Code Platforms in Practice**
  1. Overview of low-code platforms (Katalon, ReadyAPI)
  2. Key limitations (e.g., lack of OOP, redundancy issues)

Chapter 4

Part 3: Case Studies and Best Practices

- **Chapter 8: Real-World Automation Success Stories**
  1. Case studies of successful automation implementations for web apps, RESTful APIs, Kafka streams and Databases
  2. Lessons learned and best practices

Chapter 5

Part 4: Overcoming Common Challenges

- **Chapter 9: Addressing Common Issues in Automation Testing**
    1. Tips for troubleshooting and debugging

- **Chapter 3: API Testing**
  - **Testing APIs using automation tools**
  - **RESTful APIs testing frameworks.**

REST API stands for Representational State Transfer API. It is a type of API (Application Programming Interface) that allows communication between different systems over the Internet. REST APIs work by sending requests and receiving responses, typically in Json format, between the client and server.

REST APIs use HTTP methods (such as GET, POST, PUT, DELETE) to define actions that can be performed on resources. These methods align with CRUD (Create, Read, Update, Delete) operations, which are used to manipulate resources over the web.

How the data could be accessed by a call request depends on the authentication involved, could be basic using username and password, bearer token, okta token and oauth 2.0

- Basic authentication To demonstrate how an automation framework can be built for RESTful APIs, basic authentication,
  we could consider a real world project:

  Case 1: Identity Access Management.
  Technologies stack: Java, TestNG, rest assured, Apache Maven.
  Firstly we will define the API's payload as a Hash Map containing all the pairs: key-value representing all the pairs field-value from the message body.
  Secondly, we will define the rest assured connector to create our request in the upcoming test. RestAssuredConnector class defines the capability for the API: POST, GET, PUT, PATCH, DELETE and the properties set on the Header-level method. PostDeviceCreateData class builds up the Json payload for the RESTful API Call. Data modeling for the Json payload is illustrated below:

```
package com.IAM.apis;
import com.google.gson.Gson;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class PostDeviceCreateData {
public static String generateCreatePostBody (
String name, String hostName,
String serialNumber,String partnerAddress,
String partnerContName,String partnerContEmail,
String partnerType,String isStrategicPartner,
String partnerStartDate,String partnerEndDate,
String iamDevContactName,String iamDevContactEmail,
```

```
String iamDevReqdBy,String iamDevContactTeam,
String partnerAddInfo)

List<HashMap<String, String>> results = new ArrayList<>();

HashMap<String, String> devices = new HashMap<String,String>(15);

devices.put("name", name);
devices.put("hostName", hostName);
devices.put("serialNumber", serialNumber);
devices.put("partnerAddress", partnerAddress);
devices.put("partnerContName", partnerContName);
devices.put("partnerContEmail", partnerContEmail);
devices.put("partnerType", partnerType);
devices.put("isStrategicPartner", isStrategicPartner);
devices.put("partnerStartDate", partnerStartDate);
devices.put("partnerEndDate", partnerEndDate);
devices.put("iamDevContactName", iamDevContactName);
devices.put("iamDevContactEmail", iamDevContactEmail);
devices.put("iamDevReqBy", iamDevReqBy);
devices.put("iamDevContactTeam", iamDevContactTeam);
devices.put("partnerAddInfo",partnerAddInfo);

results.add(devices);
String devicePayload = new Gson().toJson(results);
return devicePayload;
}
```

RestAssuredConnector class defines the methods allowed for the Restful API: GET, POST, DELETE, PUT, PATCH.

As we go along with other examples we will add the needed methods, our class defined with POST method:

```
import io.restassured.builder.RequestSpecBuilder;
import io.restassured.http.Headers;
import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;
import java.util.HashMap;
import java.util.Map;
import static io.restassured.RestAssured.given;
public class RestAssuredConnector {
public RequestSpecification request = new
RequestSpecBuilder().build();
public static Response response;
public static void main(String[] args) {
```

```
RestAssuredConnector rac = new
RestAssuredConnector();
System.out.println(response.asString());
}
public Response postRequest(String uri, Map<String,
String> headers, String body) {
request = given().baseUri(uri).body(body).headers(headers);

return request.when().post();
}
public static Map<String, String> setHeaders(String
token) {
Map<String, String> headers = new HashMap<>();

headers.put("Content-Type", "application/json");
headers.put("Authorization", token);
return setHeaders;
}
}
```

APICreateOrderPositiveTest class will contain the necessary constants representing the stored values for the fields, setupData() method sets the url and request body for our RESTful API, all the properties are read from local-config.properties file.

```
import io.restassured.response.Response;
import org.junit.Assert;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Map;
import static org.junit.Assert.assertEquals;
public class API_CreateOrderPositiveTest {

protected static final Logger logger =
LoggerFactory.
getLogger(API_CreateOrderPositiveTest.class);

public static final String ID =
    GenerateRandomDataUtils.generateRandomNumber(1);

public static final String PET_ID =
    GenerateRandomDataUtils.generateRandomNumber(1);

public static final String NAME = "AutoRandom"
+
```

```
GenerateRandomDataUtils.generateRandomNumber(5);
public static final String HOST_NAME = "AutoRandom"+
GenerateRandomDataUtils.generateRandomNumber(5);
public static final String SERIAL_No =
GenerateRandomDataUtils.generateRandomNumber(10)
public static final String PARTNER_ADDRESS =
GenerateRandomDataUtils.generateRandomLetters(10)

public static final String PARTNER_NAME = "AutoRandom" +
GenerateRandomDataUtils.generateRandomNumber(5);
public static final StringPARTNER_EMAIL = "autoa@gmail.com"
public static final PARTNER_TYPE = "type"
public static final IS_STRATEGIC_PARTNER = "Yes"
public static final PARTNER_START_DATE = "10-10-2025"
public static final PARTNER_END_DATE = "10-10-2026"
public static final CONTACT_NAME = "Auto"
public static final CONTACT_EMAIL= "autoa@gmail.com"
public static final REQUESTED_BY = "AutoTest"
public static final CONTACT_TEAM = "Auto team"
public static finalPARTNER_ADD_INFO = "Random info"
private RestAssuredConnector connector = new
RestAssuredConnector();
private String requestBody;
private String requestUri;

@BeforeTest
public void setupData() {
PostDeviceCreateData requestDevice = new PostDeviceCreateData();

requestDevice.requestDevice(NAME, HOST_NAME,
SERIAL_NP, PARTNER_ADDRESS, PARTNER_NAME,
PARTNER_EMAIL, PARTNER_TYPE,
IS_STRATEGIC_PARTNER, PARTNER_START_DATE,
PARTNER_END_DATE, CONTACT_NAME, CONTACT_EMAIL,
REQUESTED_BY, CONTACT_TEAM, PARTNER_ADD_INFO)
requestDevice = requestMessageBody.toString();
requestUri = ConfigUtils.getProperty(ConfigKeys.BASE_URL_DEVICE.
toString())+ConfigUtils.
getProperty(ConfigKeys.END_URL_DEVICE.toString());
}

@Test
public void createPostOrderTest() {
logger.info("Making a POST report Call...)
Map<String, String> headersDevice =
RestAssuredConnector.setHeaders(ConfigUtils.
```

```
getProperty(ConfigKeys.TOKEN.toString()));
Response createDeviceResponse =
connector.postRequest(requestUri,requestHeaders, requestBody);

logger.info("requestUri" + requestUri + "" +
"requestBody: " + requestBody);
createDeviceResponse.getBody().asString());
assertEquals(createDeviceResponse.getStatusCode(),
200);
  }
}
```

Case 2 real-world project: dxp-data: gathers data from multiple systems: mParticle, Amplitude, Analytics, Youbora, Stripe.
RESTful-APIS between Stripe-Sigma, IMSmParticle, BI data Report - CDS, basic authentication.

RESTful APIs Technology Stack: Java, Junit, rest assured.

The architecture of the project is shown below.

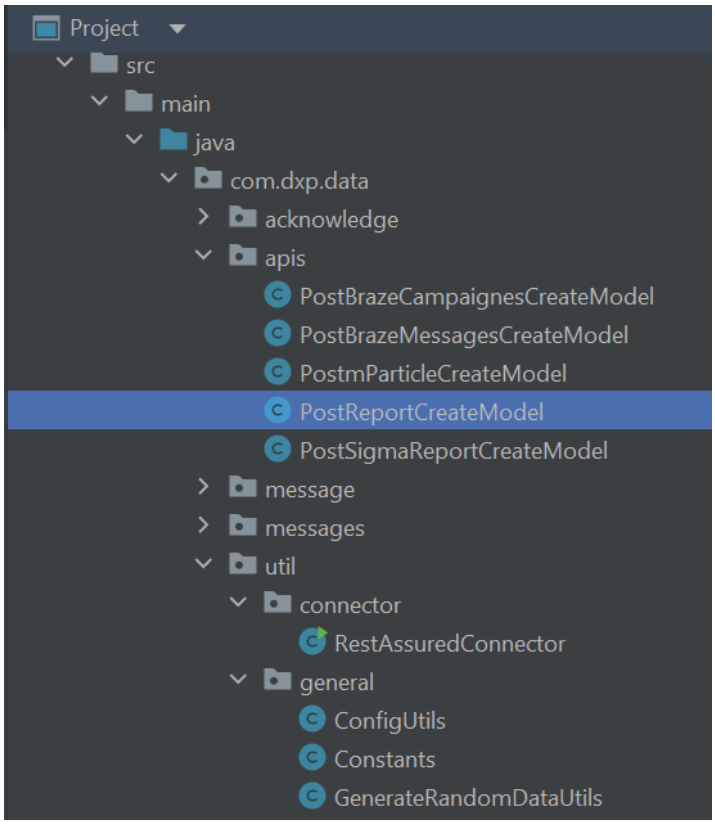

Figure 5.1: Post Report Create Model

The logic implemented is the same as in the previous example, the difference is that for each RESTful API the data is modeled using Setters and Getters.
Payload:

```
{
"date": "24-09-2025",
"network" :"app",
"type": "type",
"reason":"reason",
"event":"event",
"count":1
}
```

For each field from the payload we are creating a Setter and a Getter as we can see in the code below.

```
package com.dxp.data.apis;
public class PostReportCreateModel {
private String date;
```

```
private String network;
private String app;
private String type;
private String reason;
private String event;
private Integer count;
public void setDate(String date) {
this.date = date;
}
public void setNetwork(String network) {
this.network = network;
}
public void setApp(String app) {
this.app = app;
}
public void setType(String type) {
this.type = type;
}
public void setReason(String reason) {
this.reason = reason;
}
public void setEvent(String event) {
this.event = event;
}
public void setCount(Integer count) {
this.count = count;
}

@Override
public String toString() {
return "{\n" +
" \"date\": \"" + date + "\",\n" +
" \"network\": \"" + network + "\",\n" +
" \"app\": \"" + app + "\",\n" +
" \"type\": \"" + type + "\",\n" +
" \"reason\": \"" + reason + "\",\n" +
" \"event\": \"" + event + "\",\n" +
" \"count\": " + count + "\n" +
"}";
 }
}
```

How the tests are designed in the project is illustrated in the following figure:

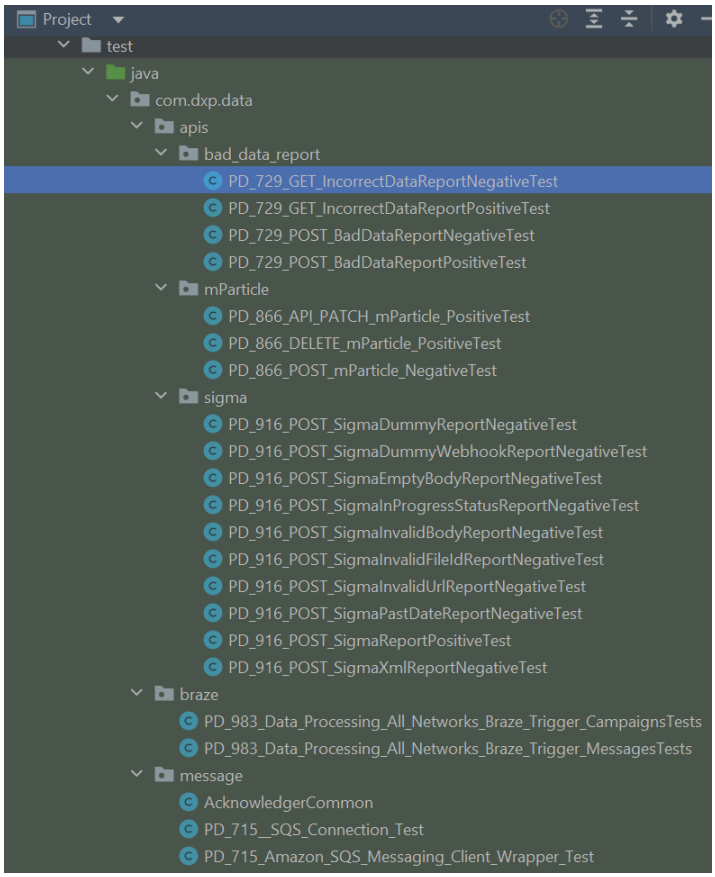

Figure 5.2: RESTful APIs tests

JUnit test contains values set for each field, @Before with setupData() method in which we build our RESTful API payload based on the model we've seen previously ,@Test - setting the properties from the Header, saving the response message and doing validation on https response code and response message; postReportHappyTest() and postReportUnHappyTest() are the two methods to capture the happy and unhappy paths to make a call to: url: "http://qa-bi-data-reportsda.cds.dxpdata.com.domain /20221002/valid-user-attribute/api/v1/bad-data-report"

The logic for the POST Report Call is illustrated in the code below:

```
package com.dxp.data.apis.bad_data_report;
import com.dxp.data.apis.PostReportCreateModel;
import com.dxp.data.util.connector.RestAssuredConnector;
import com.dxp.data.util.general.ConfigUtils;
import io.restassured.response.Response;
```

```
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Map;
import static org.junit.Assert.assertEquals;
public class PostBadDataReportHappyTest {
protected static final Logger logger =
LoggerFactory.getLogger(
POST_BadDataReportHappyTest.class);
public static final String DATE_NOW = "14-09-2023";
public static final String NETWORK = "AMC";
public static final String APP = "android";
public static final String TYPE = "invalid event";

public static final String REASON = "unplpfxed";
public static final String EVENT = "xxxxxxx";
public static final Integer COUNT = 1500;
public static final String DATE = "09-09-2021";
public static final String NETWORKDUMMY = "DUMMY";
public static final String APPDUMMY = "DUMMY";
public static final String TYPEDUMMY = "DUMMY";
public static final String REASONDUMMY = "DUMMY";
public static final String EVENTDUMMY = "DUMMY";
public static final Integer COUNTDUMMY = 1;
public static final RestAssuredConnector connector =
new RestAssuredConnector();
private String requestBody;
private String requestUri;
private String proxy;

@Before
public void setupData() {
PostReportCreateModel requestMessageBody = new
PostReportCreateModel();
requestMessageBody.setDate(DATE_NOW);
requestMessageBody.setNetwork(NETWORK);
requestMessageBody.setApp(APP);
requestMessageBody.setType(TYPE);
requestMessageBody.setReason(REASON);
requestMessageBody.setEvent(EVENT);
requestMessageBody.setCount(COUNT);
requestBody = requestMessageBody.toString();
requestBody = requestMessageBody.toString();
requestUri =
```

```
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
BASE_URL_QA.toString()) +
ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.END_URL_REPORT.toString());

proxy =
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
PROXY_QA.toString());
}
```

```
@Test
public void postReportHappyTest() {
Map<String, String> headers =
RestAssuredConnector.setHeaders();
Response reportResponse =
connector.postRequest(requestUri, proxy,
headers, requestBody);
logger.info("Making a POST Report Call: " + " " +
requestUri);
logger.info("requestBody: " + " " + requestBody);
logger.info("Response: " +
reportResponse.getBody().asString());
assertEquals(200, reportResponse.getStatusCode());
Assert.assertTrue(reportResponse.getBody().
asString(), true);
}
```

```
@Test
public void postReportUnHappyTest() {
requestMessageBody.setDate(DATE);
requestMessageBody.setNetwork(NETWORKDUMMY);
requestMessageBody.setApp(APPDUMMY);
requestMessageBody.setType(TYPEDUMMY);
requestMessageBody.setReason(REASONDUMMY);
requestMessageBody.setEvent(EVENTDUMMY);
requestMessageBody.setCount(COUNTDUMMY);
requestBody = requestMessageBody.toString();
requestUri = ConfigUtils.getProperty(
ConfigUtils. ConfigKeys.BASE_URL_QA.toString())
+
ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.END_URL_REPORT.toString());
proxy = ConfigUtils.getProperty(ConfigUtils.
```

```
ConfigKeys.PROXY_QA.toString());
Map<String, String> headers =
RestAssuredConnector.setHeaders();
Response reportResponse =
connector.postRequest(requestUri, proxy,
headers, requestBody);
logger.info("requestUri: " + " " + requestUri);
logger.info("requestBody: " + " " + requestBody);
logger.info("Response: " + reportResponse.getBody().asString());

assertEquals(reportResponse.getStatusCode(), 400);
Assert.assertTrue(reportResponse.getBody().
asString(), true);
  }
}
```

GET Call:
we need to add GET method in our Rest Assured Connector class:

```
public Response getRequest(String uri, String host,
Map<String, String> headers) {

request =
given().baseUri(uri).proxy(host).headers(headers);
return request.when().get();
}
```

Happy path:
GET Data Report Call:
"http://dev-bi-data-reports-da.cds.dexpdata.com.domain
/20221129/20221129/20221001/20221002/valid-user-attribute"

```
@Test
public void getIncorrectDataReportHappyTest()
{
proxy=ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.PROXY_DEV.toString());
requestUri=ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.BASE_URL_DEV.toString())
+
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
END_URL_REPORT.toString())
+
ConfigUtils.getProperty(ConfigUtils.
```

```
ConfigKeys.START_DATE_END_DATE.toString())
+
ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.EVENT_TYPE.toString());

Map<String, String>
headers=RestAssuredConnector.setHeaders();
Response reportResponse=connector.
getRequest(requestUri,proxy,headers);

logger.info("Making a GET Report: "+"" +requestUri);

logger.info("requestUri: "+" "+requestUri);
logger.info("Response:"+reportResponse.
getBody().asString());

assertEquals(200,reportResponse.getStatusCode());
Assert.assertTrue(reportResponse.
getBody().asString(),true);
}
```

Unhappy paths:

GET Data Report Call:
"http://dev-bi-data-reportsda.cds.dexpdata.com.domain
/17001202/17001203/20221129/20221001/20221002
/invalid-user-attribute" and
http://dev-bi-data-reports-da.cds.dexpdata.com.domain
/20221129/20221129/20221001/20221002/dummy"
getIncorrectDataReportInvalidDateInvalidDateTypeTest () and
getIncorrectDataReportInvalidEventTypeTest() methods designed below:

```
@Test
public void
getIncorrectDataReportInvalidDateInvalidDateTypeTest()
{
proxy = ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.PROXY_DEV.toString());
requestUri = ConfigUtils.
getProperty(ConfigUtils.ConfigKeys.
BASE_URL_DEV.toString())
+
ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.END_URL_REPORT.toString())
```

```
+
ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.INVALID_START_DATE_END_DATE.toString())
+
ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.INVALID_EVENT_TYPE.toString());

Map<String, String>
headers=RestAssuredConnector.setHeaders();
Response reportResponse=connector.
getRequest(requestUri,proxy,headers);

logger.info("requestUri: "+" "+ requestUri);
logger.info("Response: "+ reportResponse.getBody().asString());

assertEquals(404,reportResponse.getStatusCode());
}
```

```
@Test
public void getIncorrectDataReportInvalidEventTypeTest(){
proxy=ConfigUtils.getProperty(
ConfigUtils.ConfigKeys.PROXY_DEV.toString());
requestUri=ConfigUtils.getProperty(
ConfigUtils.ConfigKeys.BASE_URL_DEV.toString())
+
ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.END_URL_REPORT.toString())
+
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
START_DATE_END_DATE_VALID.toString())
+
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
INVALID_EVENT_TYPE.toString());

Map<String, String>
headers=RestAssuredConnector.setHeaders();
Response reportResponse=connector.
getRequest(requestUri,proxy,headers);

logger.info("Making a GET Report Call: "+requestUri);

logger.info("proxy: "+" "+proxy);
logger.info("Response:
"+reportResponse.getBody().asString());
assertEquals(404,reportResponse.getStatusCode());
```

```
}

@Test
public void getIncorrectDataReportNonExistentDatesTest(){
proxy=ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.PROXY_DEV.toString());
requestUri=ConfigUtils.getProperty(ConfigUtils.
ConfigKeys.BASE_URL_DEV.toString())
+
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
END_URL_REPORT.toString())
+
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
START_DATE_END_DATE_NON_EXISTENT.toString())
+
ConfigUtils.getProperty(ConfigUtils.ConfigKeys.
EVENT_TYPE_VALID.toString());
Map<String, String>
headers=RestAssuredConnector.setHeaders();
Response reportResponse=connector.getRequest(
requestUri,proxy,headers);
logger.info("Making a GET Report Call: "+"
"+requestUri);
logger.info("Response:
"+reportResponse.getBody().asString());
assertEquals(404,reportResponse.getStatusCode());
}
```

DELETE Call: we need to add DELETE method in our Rest-assured Connector class:

```
public Response deleteRequest(String uri, String
host, Map<String, String> headers) {
request =
given().baseUri(uri).proxy(host).headers(headers);
return request.when().delete();
}
```

Happy path:
Delete a plan from mParticle with url:
"http://qa-mparticle-ims-da.cds.amcn.com.domain//api/v1/6420/plans/2"

Test implementation:

```
@Test
public void deletemParticleUnHappyTest(){
logger.info("DELETE mParticle API Call: ");
Map<String, String> headermParticle=new HashMap<>();
headermParticle.put("Content-Type","application/json");
Response mParticleResponse=connector.deleteRequest(
requestUrimParticle,proxy,headermParticle);
logger.info("requestUrimParticle: "+"
requestUrimParticle);
logger.info("Response: " +mParticleResponse.getBody().asString());

assertEquals(200,mParticleResponse.getStatusCode());
}
```

PATCH Call: we need to add we need to add PATCH method in our RestAssuredConnector class:

```
host,Map<String, String> headers,String body)
{
request=given().baseUri(uri).proxy(host).
headers(headers).body(body);
return request.when().patch();
}
```

- Okta Token Token Authentication is more secure when compared to basic authentication, since it involves the use of a unique token that is generated for each user. This token is sent with each request, and is used to authenticate the user.

  Token authentication is also a good choice for applications that require frequent authentication, such as single-page applications or mobile applications. Since the authentication process does not require user passwords in each request, once a user enters the credentials, receives a unique encrypted token that is valid for a specified session time.It is more efficient and can handle more concurrent requests. Let's use a simple case for an API call using Okta authentication, illustrated in the next figure.

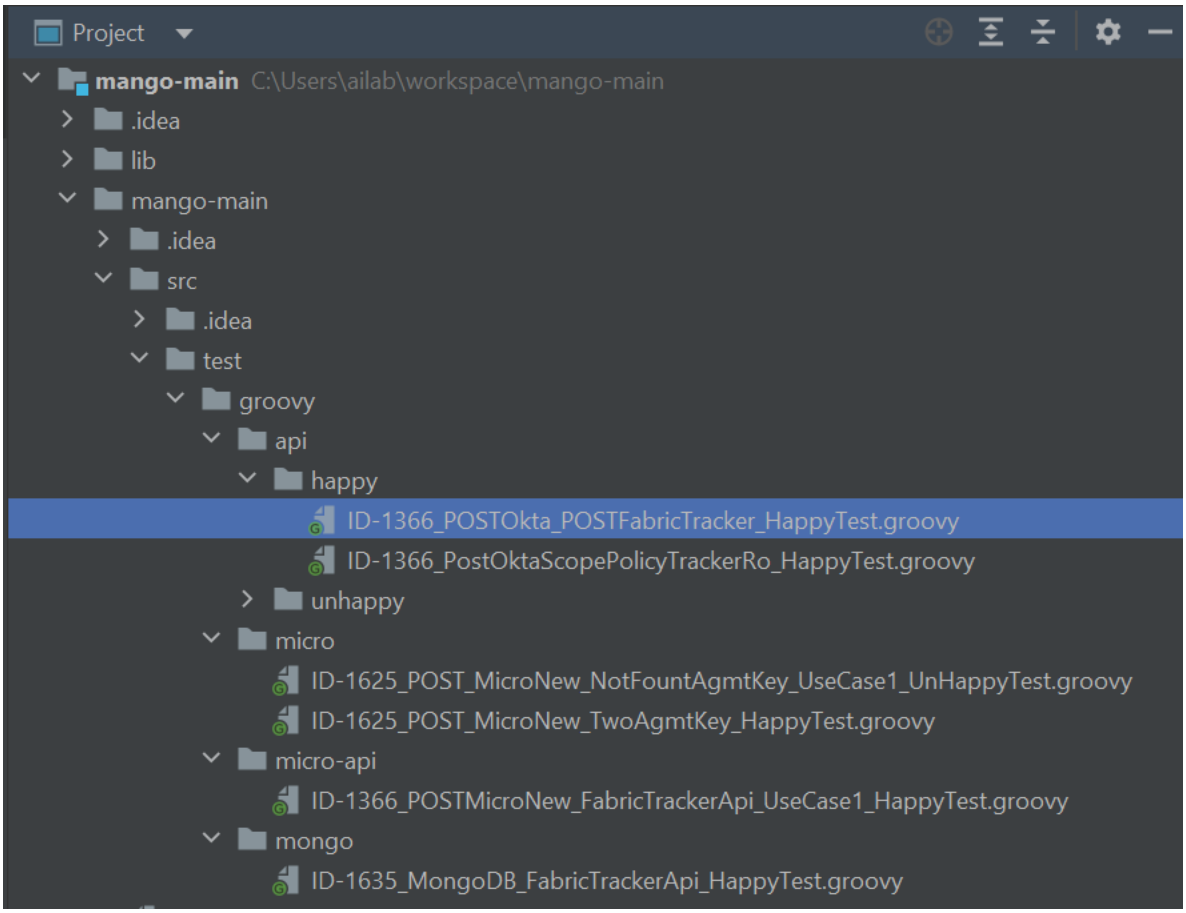

Figure 5.3: Okta Token Groovy structure

Having the following Groovy script:

```
import okhttp3.*
import org.json.JSONObject
import org.assertj.core.api.SoftAssertions

Random random = new Random()
int randomInt = random.nextInt(10000000) + 1
randomString = randomInt.toString()
String adminSys = "IDK"
String key = "00" + randomString + "|||" + adminSys
String code = "CODE"

//Define necessary variables for Okta
def url_okta = "https://oktatokengeneration/aus456788"
def usr = "username"
```

```
def pswd = "password"

RequestBody okta = RequestBody.create(null, " ")

OkHttpClient client = new OkHttpClient()
Request request_okta = new Request.Builder().
url(url_okta).header("Authorization",
Credentials.basic(usr, pswd)).
header("Content-Type", "application/json").
header("Content-Type",
"application/x-www-form-urlencoded").post(okta).build()

Response response_okta = client.newCall(request_okta).execute()
String responseBody_okta = response_okta.body().string()
int statusCode_okta = response_okta.code()

SoftAssertions softly = new SoftAssertions()
softly.assertThat(statusCode_okta).isEqualTo(200)
softly.assertAll()

log.info("Response for OktaI Call :" + responseBody_okta)
log.info("Status code forOkta Call:" + statusCode_okta)

//Save bearer token
JSONObject rspOkta = new JSONObject(responseBody_okta)
String access_token = rspOkta.getString("access_token")
String token_type = rspOkta.getString("token_type")
String expires_in = rspOkta.getString("expires_in")
String scope = rspOkta.getString("scope")

softly.assertThat(token_type).isEqualTo("Bearer")
softly.assertThat(expires_in).isEqualTo("3600")
softly.assertThat(scope).isEqualTo("tracker")
softly.assertAll()
```

We are using library okhttp3, and making a POST with null
to our Okta token request, having Content-Type:
application:x-www-form-urlencoded AND application:json Also access token value is what we are interested in for the future calls, as we will see in our next code section:

```
//API Call using Okta Token

//Define necessary variables for Fabric API POST Call
def url_api = "https://trcakerapi
```

```
JSONObject sys = new JSONObject()
sys.put("code", code)

JSONObject tracking = new JSONObject()
tracking.put("key", key)
tracking.put("sys", sys)

JSONObject tracker = new JSONObject()
tracker.put("tracking", tracking)
RequestBody trackingBody = RequestBody.create(MediaType.parse
("application/json"),tracker.toString())

Request request_tracker = new Request.Builder().url(url_api).
header("Authorization", "Bearer" + access_token).
header("Content-Type", "application/json").post(trackingBody).build()

Response response_tracker = client.newCall(request_tracker).execute()

String responseBody_tracker = response_tracker.body().string()
int statusCode_tracker = response_tracker.code()

softly.assertThat(statusCode_tracker).isEqualTo(200)
softly.assertAll()

log.info("Response for POST Tracker API Call :" + responseBody_tracker)
log.info("Status code for POST Tracker API Call:" + statusCode_tracker)
```

```
Request request_tracker = new Request.Builder().url(url_api).
header("Authorization", "Bearer" + access_token).
header("Content-Type", "application/json").post(trackingBody).build()
```

we are creating a call request using the api url passing on the header the authorization the access token with the payload set in the code snippet:

```
JSONObject sys = new JSONObject()
sys.put("code", code)

JSONObject tracking = new JSONObject()
tracking.put("key", key)
tracking.put("sys", sys)

JSONObject tracker = new JSONObject()
tracker.put("tracking", tracking
```

- Auth 2.0 OAuth is an open standard for authorization that provides a way for users to grant access to their data without sharing their passwords. OAuth is used to authenticate users and authorize access to a system or service.

  OAuth 2.0 authentication in REST API is a great option for applications that need to access user data from other services, such as Google, Facebook, Twitter, or any other external service. It allows users to grant access to their data without having to share their username and password with the application.

  In addition the Okta authentication that we have already seen, there is an extra level of security to have a user token generated for the request access data. The next figure illustrates Groovy scripts using Auth 2.0.

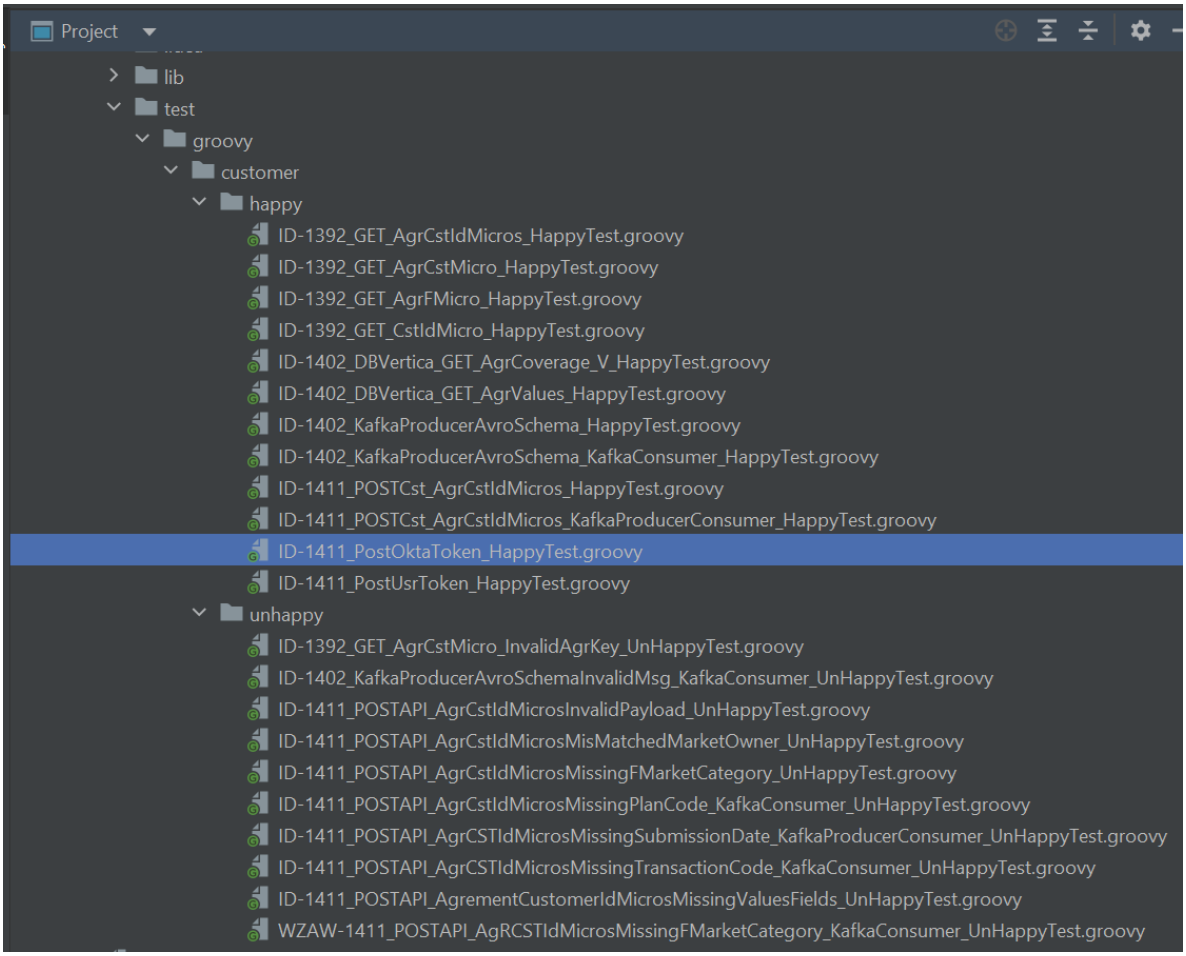

Figure 5.4: Auth 2.0

```
import okhttp3.*
import org.json.JSONArray
```

```
import org.json.JSONObject
import org.assertj.core.api.SoftAssertions
import org.slf4j.*
import java.util.*

//Define necessary variables for User Token
def url_user = "https://usertoken
def username_user = "username"
def password_user = "password"
def client_id = "client_id"
def client_secret = "client_secret"

JSONObject bodyUsr = new JSONObject()

bodyUsr.put("client_id", client_id)
bodyUsr.put("client_secret", client_secret)
bodyUsr.put("grant_type", "")
bodyUsr.put("realm", "")
bodyUsr.put("audience", "")
bodyUsr.put("username", "")
bodyUsr.put("password", "")

RequestBody payloadUser =
    RequestBody.create(MediaType.parse("application/json"),
    bodyUser.toString())

Request request_User = new Request.Builder().url(url_user).
header("Authorization", Credentials.basic(username, password)).
header("Content-Type", "application/json").post(payloadUser).build()

OkHttpClient client = new OkHttpClient()
Response response_user = client.newCall(request_User).execute()
String responseBody_user = response_user.body().string()
int statusCode_user = response_user.code()

SoftAssertions softly = new SoftAssertions()

softly.assertThat(statusCode_user).isEqualTo(200)
softly.assertAll()

log.info("Response for User Call :" + responseBody_user)
log.info("Status code for User Call:" + statusCode_user)

//Save id_token
JSONObject rspUser = new JSONObject(responseBody_user)
String user_access_token = rspUser.getString("access_token")
```

Let us analyze our code. To generate a user token, we need to set as a payload:

```
bodyUsr.put("client_id", client_id)
bodyUsr.put("client_secret", client_secret)
bodyUsr.put("grant_type", "")
bodyUsr.put("realm", "")
bodyUsr.put("audience", "")
bodyUsr.put("username", "")
bodyUsr.put("password", "")
```

then we build our request using the url and passing on the header credentials with the payload set above:

```
Request request_User = new Request.Builder().url(url_user).
header("Authorization", Credentials.
basic(username, password)).
header("Content-Type", "application/json").
post(payloadUser).build()
```

Making the call will result in having obtained the user access token that we are going to use in our future Api calls using oauth 2.0 as follows:

```
import org.assertj.core.api.SoftAssertions
import org.slf4j.*
import java.util.*
import java.text.DateFormat
import java.text.SimpleDateFormat
import okhttp3.*
import org.json.JSONArray
import org.json.JSONObject

Date now = new Date()
SimpleDateFormat date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
String currentDateComplete = date.format(now)
String dateEDT = currentDateComplete + ".100 EDT"

//Define necessary variables for Okta
def url_okta = "https://oktatoke"
def usr = "username"
def pswd = "password"
RequestBody okta = RequestBody.create(null, " ")

OkHttpClient client = new OkHttpClient()
```

```
Request request_okta = new Request.Builder().
url(url_okta).header("Authorization",
Credentials.basic(usr, pswd)).
header("Content-Type", "application/json").header("Content-Type",
"application/x-www-form-urlencoded").post(okta).build()

Response response_okta = client.newCall(request_okta).execute()
def responseBody_okta = response_okta.body().string()
int statusCode_okta = response_okta.code()

SoftAssertions softly = new SoftAssertions()

softly.assertThat(statusCode_okta).isEqualTo(200)
softly.assertAll()

log.info("Response for OktaI Call :" + responseBody_okta)
log.info("Status code for Okta Call:" + statusCode_okta)

//Save bearer token
JSONObject rspOkta = new JSONObject(responseBody_okta)
String access_token = rspOkta.getString("access_token")

//Define necessary variables for User Token
def url_user = "https://usertoken"
def username_user = "user"
def password_user = "pswd"
def client_id = "client_id"
def client_secret = "client_secret"

JSONObject bodyUser = new JSONObject()

bodyUser.put("client_id", client_id)
bodyUser.put("client_secret", client_secret)
bodyUser.put("grant_type", "")
bodyUser.put("realm", "")
bodyUser.put("audience", "")
bodyUser.put("username", "m")
bodyUser.put("password", "")
RequestBody payloadUser = RequestBody.create(MediaType.
parse("application/json"), bodyUser.toString())

Request request_User = new Request.Builder().url(url_user).
header("Authorization", Credentials.Credentials.
basic(username_user, password_user)).
header("Content-Type", "application/json").
post(bodyUser).build()
```

```
Response response_user = client.newCall(request_User).execute()
String responseBody_user = response_user.body().string()
int statusCode_user = response_user.code()

//Save id_token
JSONObject rspUser = new JSONObject(responseBody_user)
String user_access_token = rspUser.getString("access_token")

//Define necessary variables for the Agreement Customer
String agmtKey = "1111"
String adminSystem = "AAAA"

def url_agr_cst_micro = "https://agrcstmicro" + "/" + agmtKey + "|||" +
    adminSystem
def username_agr = "agrcst"
def password_agr = pswd2025

Request request_agrKey = new Request.Builder().url(url_agr_cst_micro).
header("Authorization", Credentials.
basic(username_agr, password_agr)).
header("Content-Type", "application/json").build()

Response response_agrKey = client.newCall(request_agrKey).execute()
String responseBody_agrKey = response_agrKey.body().string()
int statusCode_agrKey = response_agrKey.code()

log.info("Response for agreementKey:" + responseBody_agrKey.toString())
log.info("Status code for agreementKey:" + statusCode_agrKey)

JSONObject jsonResponseAgreementCustomer = new
    JSONObject(response_agrKey)

JSONArray agreements = jsonResponseAgreementCustomer.
getJSONArray("agreements")
JSONObject customer = agreements.getJSONObject(0).
getJSONArray("agreementCustomers").getJSONObject(1)

String roleType = customer.getString("roleType")
String memberGUIDAgreementCustomer = customer.getString("memberGUID")

//Define necessary variables for Customer Id
def url_customer_id_micro = "https://cstid +
"/" + memberGUIDAgreementCustomer
def username_customerIds = "username"
def password_customerIds = "password"
```

```
Request request_customerId = new Request.Builder().
url(url_customer_id_micro).
header("Authorization", Credentials.
basic(username_customerIds, password_customerIds)).
header("Content-Type", "application/json").build()

Response response_customerId = client.newCall(request_customerId).
execute()
String responseBody_customerId= response_customerId.body().string()
int statusCode_customerId = response_customerId.code()

log.info("Response for CustomerId:" + responseBody_customerId.toString())
log.info("Status code for CustomerId:" + statusCode_customerId)

JSONObject jsonResponseCustomerId = new
    JSONObject(responseBody_customerId)

JSONArray customers = jsonResponseCustomerId.
getJSONArray("customers")
String memberGUIDCustomerId = customers.getJSONObject(0).
getString("memberGUID")

softly.assertThat(memberGUIDAgreementCustomer).
isEqualTo(memberGUIDCustomerId)
softly.assertAll()

//POST Cst
def url_api = "https://cstapi" + "/" + memberGUIDAgreementCustomer +
    "/agreements/" + agmtKey + "|||" + adminSystem

log.info("Calling Fabric API:" + url_api)

// template
String cts = """
{
  "transaction":{
    "code":"CODE",
    "date":"${dateEDT}"
  },
  "product":{
    "item":{
      "plan":"PLAN",
      "category":"Allo"
    }
  }
```

```
}
"""

log.info(cts)
RequestBody cstPayload=RequestBody.create(MediaType.
parse("application/json"), cts)

Request request_cst = new Request.Builder().url(url_api).
header("Authorization", "Bearer " + access_token).
header("Content-Type", "application/json").header("Accept", "*/*").
header("Accept-Encoding","gzip, deflate, br").
header("Connection","keep-alive").
header("userToken", user_access_token)
.post(cstPayload).build()

Response response_cst= client.newCall(request_cst).
execute()
String responseBody_cst = response_cst.body().string()
int statusCode_cst= response_cst.code()

log.info("Response for POST Cst Call :" + responseBody_cst)
log.info("Status code for POST Cst Call:" + statusCode_cst)
```

Our final call request is having as payload:

```
{
  "transaction":{
    "code":"CODE",
    "date":"${dateEDT}"
  },
  "product":{
    "item":{
      "plan":"PLAN",
      "category":"Allo"
    }
  }
}
```

an on the header level the 2 layers of authorization: Okta token and user token:

```
Request request_cst = new Request.Builder().url(url_api).
header("Authorization", "Bearer " + access_token).
header("Content-Type", "application/json").
header("Accept", "*/*").header("Accept-Encoding","gzip, deflate, br").
header("Connection","keep-alive").
```

```
header("userToken", user_access_token).post(cstPayload).build()
```