



University of Piraeus
School of Information and Communication Technologies
Department of Digital Systems

Postgraduate Program of Studies
MSc Digital Systems Security

OpenID Connect & SSO in Mobile Networks (5G)

Supervisor Professor: Ch. Xenakis

Name-Surname	E-mail	Student ID.
Stylianou Ioannis	jsm@hotmail.gr	mte2126

Piraeus
26/10/2022

Περίληψη

This project aims to design and implement a state-of-the-art mechanism for authentication and authorization in mobile networks and specifically 5G, complying with the CAPIF standard, as well as allowing single sign-on between CAPIF instances of different providers. The implementation revolves around the CAPIF framework, which facilitates communication between applications and network functions [3], and attempts to do so in a secure manner with additional functionality. The goal is to implement the authentication and authorization aspects of the CAPIF framework in an enhanced manner using OpenID Connect on top of OAuth2.0. OpenID Connect adds an identity layer on top of the OAuth2.0 method of the framework and allows for single sign-on capabilities between the CAPIF instances of different providers. The identity aspect that is offered can be used for continuous authentication and log generation.

Abstract

Next-generation communication networks offer faster speeds and lower latency as well as higher capacity and bandwidth. While previous communication networks were focused on mobile phones, the future of networking starting from 5G, allows for an outburst of innovation and technological advancement of most industries, notably healthcare, manufacturing and automation. In order to fulfill the aforementioned requirements, technologies such as Software Defined Networking (SDN), Network Functions Virtualization (NFV), edge computing and virtualization are introduced, which make the future of networking vastly different from existing 4G networks [1].

Due to the differentiation of 5G technology and architecture, the properties and security mechanisms of existing networks are hereby insufficient. Moreover, by adopting many different technologies, 5G networks not only take advantage of the benefits, but inherit the inherent both existing and future security challenges of those technologies as well [2].

Table of Content

Relevant Technologies	1
OAuth 2.0.....	1
OpenID Connect	1
OpenID vs OAuth 2.0	2
Keycloak	2
CAPIF framework.....	2
Network Exposure Function	3
Proposed Solution	5
Motivation – Aim.....	5
Design	5
Use Cases	5
States of the NetApp	7
Message Flow	7
Implementation	10
Tools	10
Scenarios	11
Code	12
Initialization	12
Wrapper Functions.....	13
Continuous Authentication	15
NEF API Consumption	15
Configuration	18
Testing.....	19
Verification & Validation	19
References.....	21

Relevant Technologies

OAuth 2.0

OAuth is an abbreviation for “Open Authorization”. This standard allows a website/app to access resources on behalf of a user. It is important to note that this protocol solely handles the authorization, and has nothing to do with authentication. OAuth 2.0 functions using access tokens, identifiers that represent the authorization for access on behalf of the user. Access tokens are commonly JSON Web Tokens (JWT) format and have an expiration date.

In the OAuth protocol the following roles are specified:

- Client Application

An application requesting access to resources of the server. (ex. A website requesting access to a user’s google account)

- Resource Owner

The user (person or application) that owns the data to be shared. (ex. The owner of said google account)

- Resource Server

The server that provides the resources. (ex. Google)

- Authorization Server

The server that permits the Client Application to access the Owner’s resources. The resource and authorization server can be the same.

OpenID Connect

OpenID Connect is an identity layer on top of Auth2.0 [4]. OpenID connect allows clients to use single sign-on to access different parties. This protocol passes on user information to different parties, allowing the user to provide a single set of credentials to access multiple applications. For example, a user can own a Google account, and sign in using said account on any application that supports OpenID connect.

OpenID vs OAuth 2.0

OpenID and Auth2.0 are commonly misconceived due to the false notion that authentication and authorization are synonyms.

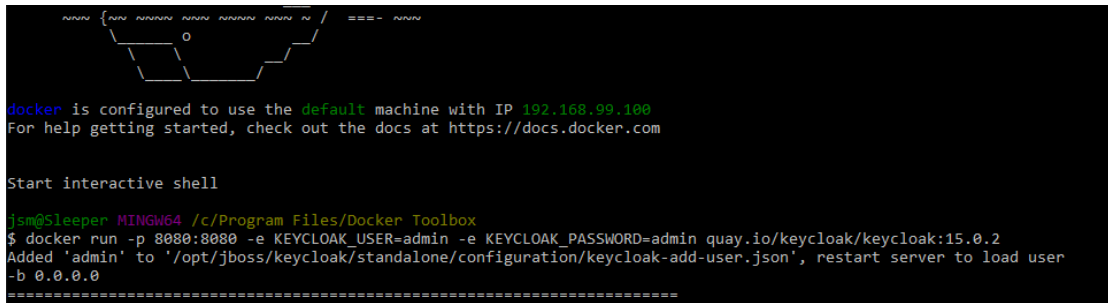
- Authentication describes the process of identifying a user.
- Authorization describes the process of giving permission to a user to access a certain resource or functionality.

OAuth2.0 covers authorization only, whereas OpenID Connect also handles client identification and authentication.

Keycloak

Docker is an open-source platform that supports developing and running applications in an isolated environments (user spaces) that are called software containers.

Keycloak is an open-source Identity and Access Management (IAM) solution that implements OpenID connect and supports single sign-on. Keycloak can be run inside a docker container using the command shown in Fig. 1.

A terminal window showing the execution of a Docker command to run Keycloak. The terminal output includes a Docker configuration notice, a shell prompt, and the execution of a 'docker run' command with various flags and environment variables. The command is: `$ docker run -p 8080:8080 -e KEYCLOAK_USER=admin -e KEYCLOAK_PASSWORD=admin quay.io/keycloak/keycloak:15.0.2`. The output shows that the user 'admin' was added to the configuration and the server is ready to start.

```
docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

Start interactive shell

jsm@Sleeper MINGW64 /c/Program Files/Docker Toolbox
$ docker run -p 8080:8080 -e KEYCLOAK_USER=admin -e KEYCLOAK_PASSWORD=admin quay.io/keycloak/keycloak:15.0.2
Added 'admin' to '/opt/jboss/keycloak/standalone/configuration/keycloak-add-user.json', restart server to load user
-b 0.0.0.0
=====
```

Figure 1 - Keycloak Execution in Docker

Keycloak will be playing the role of Authorization Server in OAuth2.0's protocol.

CAPIF framework

Many different technologies are introduced in 5G networks, which not only take advantage of their benefits, but also inherit any current or future security risks they may have [2]. In order to help combat these issues, 3GPP has released the Common API Framework (CAPIF), which defines a security architecture with distinct features and mechanisms.

The Common API Framework is composed of the following entities, the architecture and inter-communication of which is shown in Fig. 2.

- CAPIF Core Function (CCF)
- API Exposing Function (AEF)
- API Invoker

Key Functional Entities

- **CAPIF Core Function (CCF)** is a repository of all, PLMN and 3rd party, service APIs
 - allows discovery of the stored APIs by the API invokers and AEFs
 - authenticates and authorizes API invokers for use of the service APIs
 - logging and charging the API invocations
- **API Exposing Function (AEF)** is the provider of the services as APIs
 - validates the authorization of the API Invokers
 - provides the service to the API invoker
 - logs the invocations on the CCF and requests charging for the service.
- **API Invoker** is typically the applications that require service from the service providers
 - discovers the service APIs from the CAPIF Core Function
 - seeks authorization for API invocations
 - avails the services provided by the AEFs

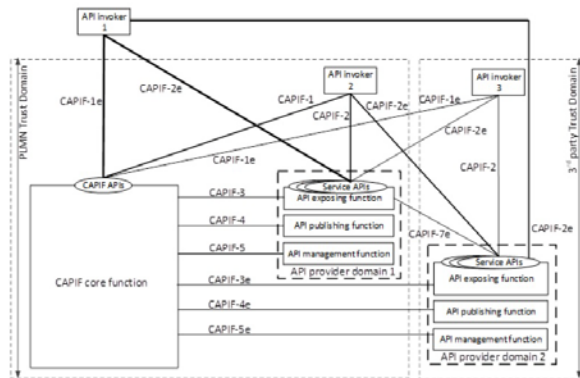


Figure 2. CAPIF Key Functional Entities (credit: 3GPP).

The Common API Framework has the following key features, according to specification TS 23.222:

- On-boarding/off-boarding invoker
- Register/un-register APIs
- API discovery
- Support 3rd party domains
- Function federation to support distributed deployments
- Event subscriptions and notifications
- Entity authentication/authorization
- Enable secure communication

The CAPIF framework has three methods proposed for authentication and authorization:

- Pre-Shared Keys (PSK)
- Public Key Infrastructure Certificates (PKI)
- OAuth 2.0 Tokens

Network Exposure Function

The Network Exposure Function (NEF) is one of the 5G network's capabilities. The NEF allows applications to subscribe to network changes, allowing for secure,

robust and developer-friendly API exposure in a way that encourages openness. The NEF emulator built under the scope of H2020 EVOLVED-5G is an open emulator of the NEF that can run inside a virtual machine. This emulator allows the developer to create different UEs, cells and paths they UEs will follow, showing their movement on an interactive map, as in Fig. 3. The NEF emulator will be used to emulate the 5G network that will be accessed securely by the invoker through the NetApp.

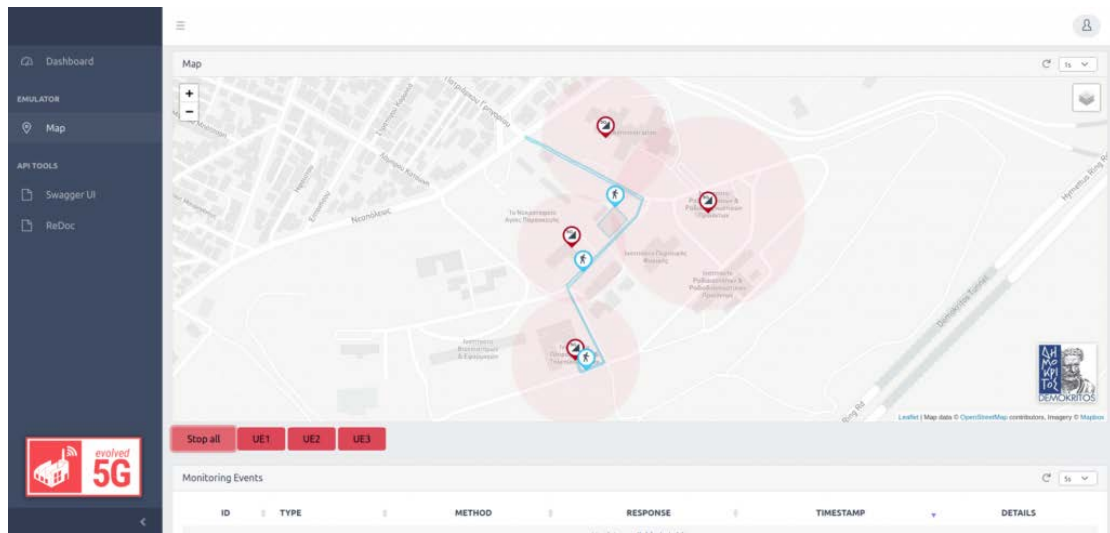


Figure 3 - NEF Emulator

Proposed Solution

Motivation – Aim

The proposed solution complies with the CAPIF framework, and specifically the OAuth2.0 method of authentication. Furthermore, OpenID Connect is used on top of OAuth2.0 as an enhancement [6], adding an identity layer and allowing for SSO between CAPIF instances, therefore between providers. OpenID Connect can not only be an enhancement of the authentication and authorization mechanisms of CAPIF [6] but cellular-based OIDC has been tested to reduce latency for up to 88.3% compared to existing authentication and authorization mechanisms [5]. This implementation can run inside a docker container, therefore employing any security monitoring and prevention mechanisms that serve different systems on mobile networks (ex. container security, cloud security) [3]. Using the identity layer provided by OAuth2.0, any logs generated by the system are tied to an identity, meaning that machine learning algorithms can be adopted for security in order to prevent 0-day attacks, especially when recent algorithms scale well with the amount of data provided, and telecom networks can produce a great amount of data [3]. Acting as a CAPIF enhancement, the implementation already provides many benefits. When considering the possibility of SSO, by simply integrating with CAPIF instead of 5G networks, the implementation can act as an authentication proxy, picking the correct IdP the invoker belongs to, and allowing API consumption of any provider's CAPIF instance, effectively providing automated SSO between providers.

Design

Use Cases

The NetApp is an implementation of the Authorization and Authentication aspects of the CAPIF framework. As such, this system provides functionality for authenticating user equipment (UEs), de-authenticating user equipment as well as authorizing user equipment. Processes such as onboarding and offboarding user equipment described in the CAPIF framework are not included in the scope of the System, and can be handled directly by the Identity Provider.

User Equipment includes all equipment that needs to connect to 5G networks. The identity provider instance describes different Keycloak clients within the same realm.

NEF describes the Network Exposure Function of different 5G Networks. Part of the system's functionality is to route the UE request to the correct Identity provider or Network Exposure Function that corresponds to the respective UE.

The Authentication process involves communication between UE and the System, which then contacts the corresponding IdP in order to validate the UE's credentials and obtain a session OAuth 2.0 Token through OpenID Connect.

The Logout process involves the UE and corresponding IdP, in order to deactivate the UE's active token and clear the session variable. Continuous authentication occurs without UE interaction and de-authenticates a UE in case of misuse of the NetApp. As such, it includes functionality from the Logout process. De-authentication of UEs encapsulates both aforementioned scenarios.

The Authorization process which is invoked when the UE attempts to access a protected resource involves the UE and both the IdP corresponding to the UE as well as the NEF corresponding to the Provider of the protected resource. This includes token introspection from the IdP. In case the UE is attempting to access a protected resource of another provider, the system will introspect the token against the UE's corresponding IdP Keycloak instance and allow access to said protected resource. Fig. 4 represents this scenario in a wholistic use-case perspective.

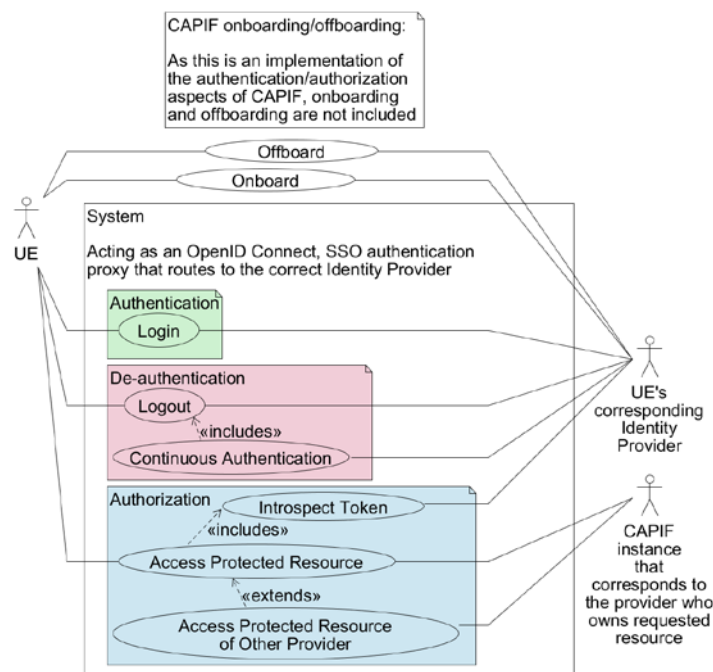


Figure 4. Use Case Diagram

States of the NetApp

There are 3 distinct actions that invoke the NetApp's functionality, and 6 total functionalities/states of the System.

The /login action begins the Login process which contains the request data and session information variables. After initial checks that there is credential data in valid form, the credentials are verified and the IdP is contacted to authenticate the UE and generate an OAuth 2.0 token through OpenID Connect. The token and UE's corresponding provider info is stored inside the session variable. If the credentials are valid the UE is authenticated, otherwise there is a login failure response sent to the UE and the flow is concluded.

The /logout action contains the corresponding provider from the session variable and the corresponding Keycloak IdP instance's connector class. It is confirmed that the UE is currently authenticated before continuing using the authorize state. In case the UE is authorized and the flow continues, the UE is logged out and becomes de-authenticated. In case it is not authorized, there is an appropriate response sent to the UE and the flow is terminated.

The Access NEF endpoint action begins the access protected resource process. In case the request is invalid due to missing endpoint or invalid method used, the de-authenticate state begins. Otherwise, the authorized state begins to confirm that the UE is authorized. In case the UE is authorized the flow continues and the NEF request state is invoked. Fig. 5 represents this scenario from the machine's states perspective.

Message Flow

The UE sends an HTTP request to the /login endpoint along with credential data in JSON form. The credentials are sent to the UE's corresponding IdP. In case the credentials are valid the IdP returns an access token which is then forwarded to the UE. Otherwise, an exception is thrown from the IdP and an "invalid credentials" response is sent to the UE.

The UE sends a request to access a protected resource. In case the method is not allowed or the endpoint does not exist, the UE is de-authenticated due to misuse and an appropriate response is sent. Otherwise, the following occur: The token is sent to the

corresponding IdP for introspection and the token status is returned to the NetApp. In case the token is active a resource request is sent towards the corresponding provider of said resource. The provider's response is forwarded back to the UE. In case the token is inactive a "not authenticated" response is sent to the UE.

The UE sends a request to logout to the NetApp. The token is sent to the IdP for introspection. In case the token is active, the user is authorized to perform said action, is logged out, and a "logged out" response is sent back to the UE. In case the token is inactive, a "not authenticated" response is sent to the UE.

Any subscription created using the access protected resource message flow can then generate async callback notifications from the corresponding NEF that provider of said resource directly to the UE or any other address specified in the JSON body during the subscription creation request. Fig. 6 represents this scenario from the message flow perspective.

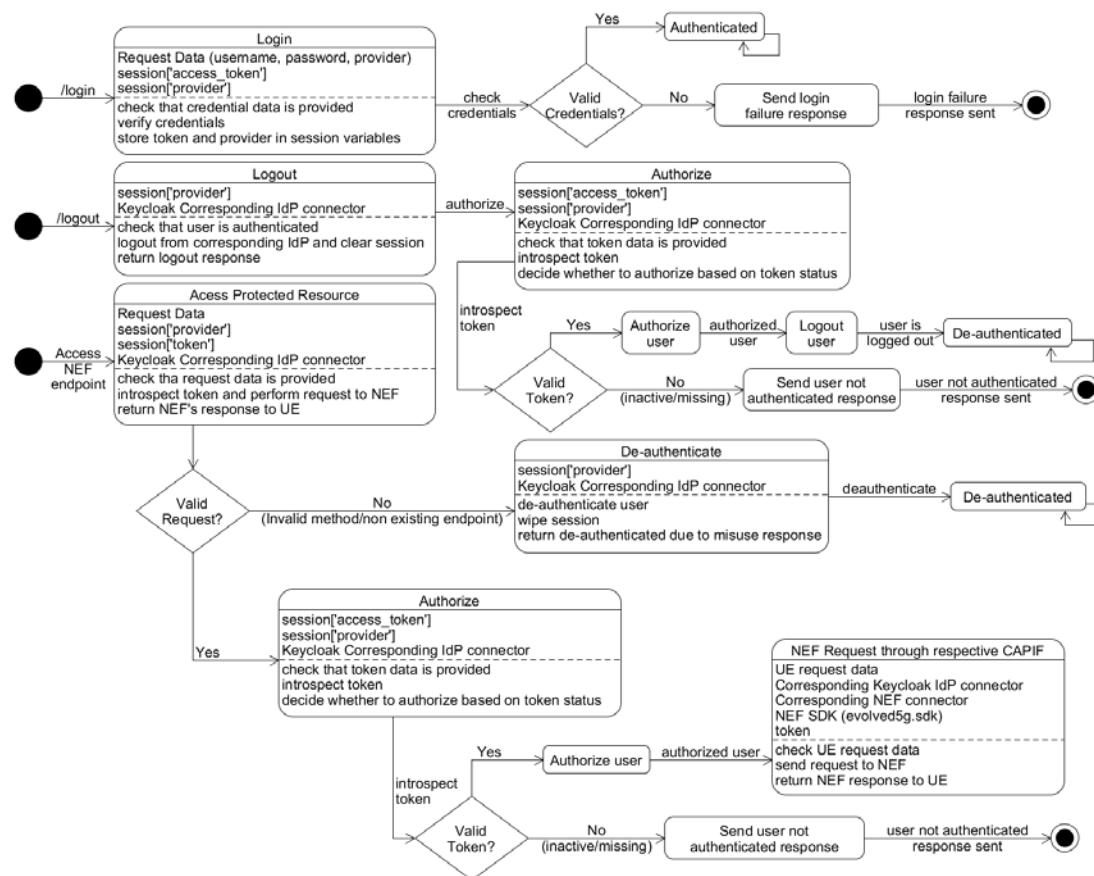


Figure 5. Machine State Diagram

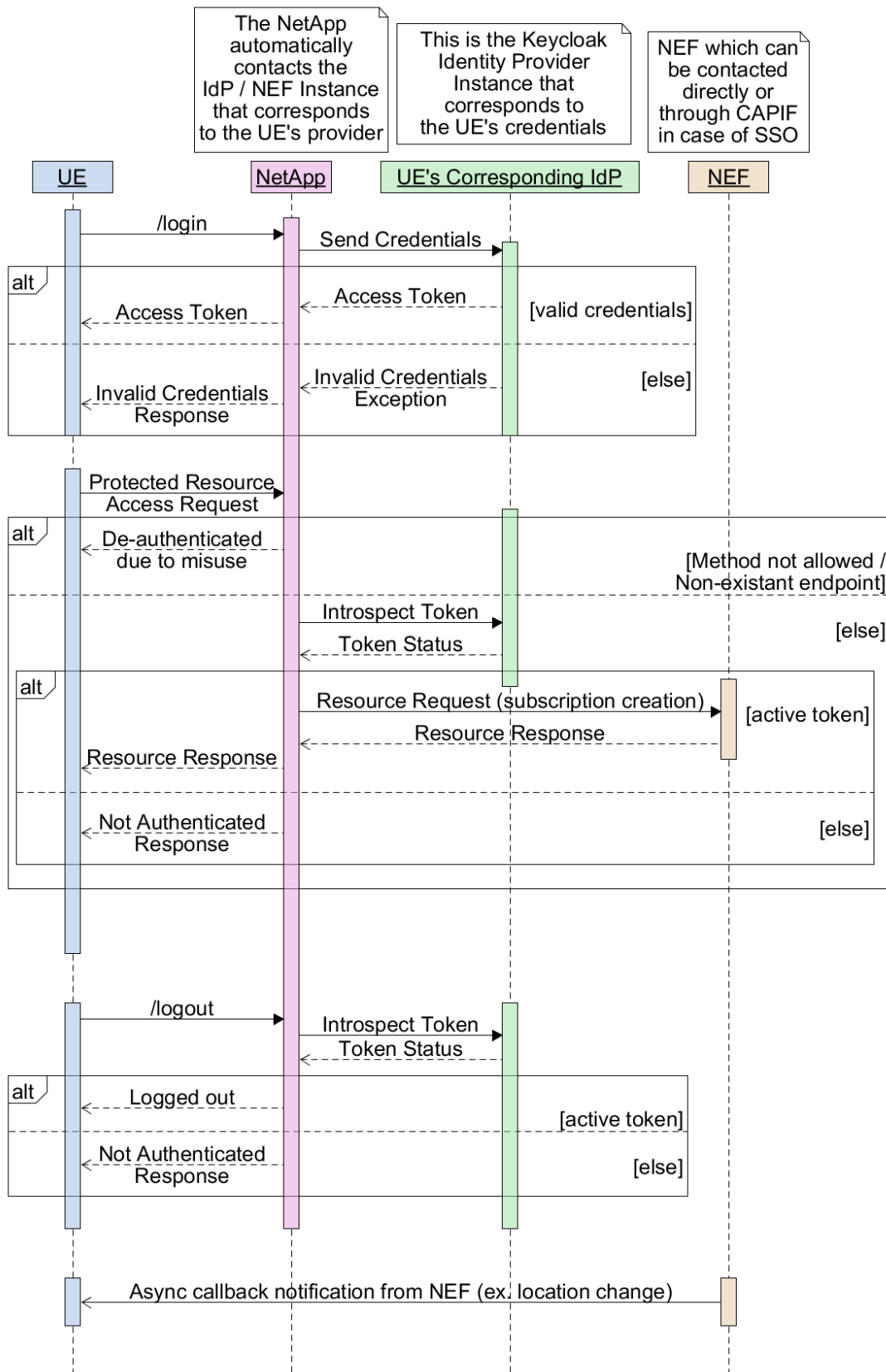


Figure 6. Activity Diagram

Implementation

Tools

The implementation is a Python (flask) API running inside a docker container. Keycloak open source IAM is running in another container, communicating with the API. The NetApp provides endpoints for authentication and consumption of the NEF Emulator APIs. A 3rd party can only consume the NEF Emulator APIs after authentication and authorization, otherwise access is declined. The python libraries used can be found in the file “requirements.txt” provided, and they are flask, keycloak-client, requests, evolved5g, datetime, configparser. The “emulator_utils.py” file is also imported in order to communicate with the NEF emulator. The Dockerfile provided ensures the proper setup of the container of the NetApp, as shown in Fig. 7. The NetApp was originally developed and tested on a Windows 10 Host, while the NEF Emulator was hosted inside an Ubuntu 20.04 Virtual Machine. The NetApp has been fully containerized. Docker Toolbox is used to run the NetApp container as well as the Keycloak container. For communication between the NetApp and Keycloak, the python library keycloak-client is used, while the Evolved5G SDK is used for communication between the NetApp and the NEF Emulator.

```
FROM python:3.9
# FROM base_image:version
# install dependencies

WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

# copy all files and folders of the NetApp Python project into the image
COPY NetApp-v2.py /app
COPY emulator_utils.py /app
COPY config.json /app

#execute commands in the container
CMD [ "python", "NetApp-v2.py", "--config", "container", "--host=0.0.0.0" ]
ENTRYPOINT [ "python3", "-u", "NetApp-v2.py" ]
```

Figure 7 – Dockerfile

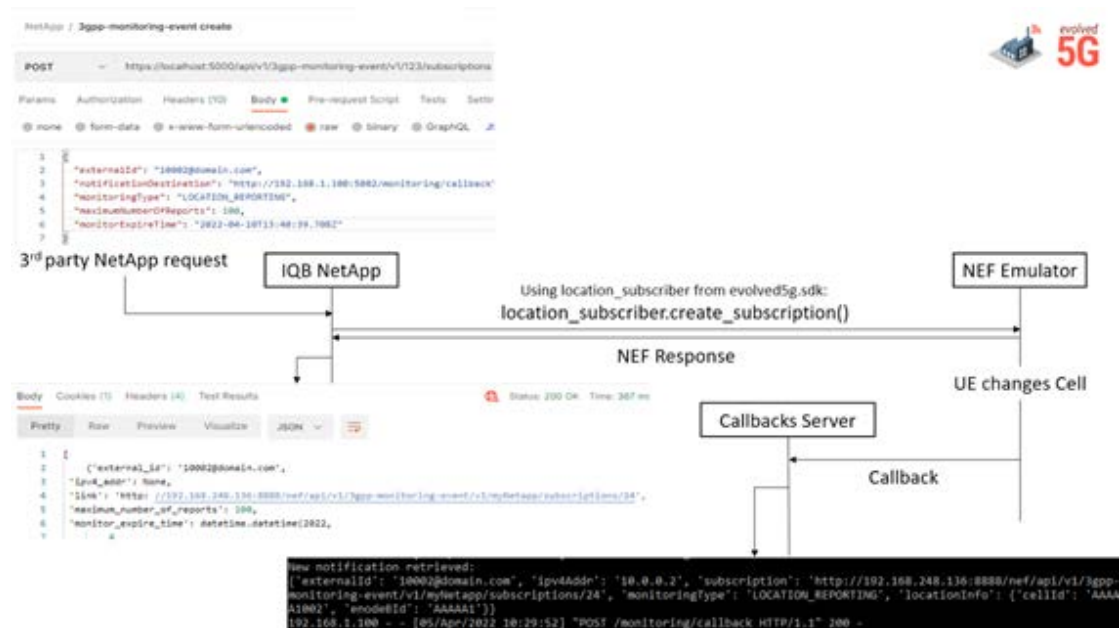


Figure 9 - NEF API Consumption

Code

Initialization

In order to initialize variables, the CONFIG file is parsed, depicted in Fig. 10.

```

import configparser
with open('config.json', 'r') as jsonfile: CONFIG=json.load(jsonfile)

~
apiRoot = CONFIG['apiRoot']
nefEMU = CONFIG['nefEMU']
selfURL = CONFIG['selfURL']
keycloakURL = CONFIG['keycloakURL']
unittests = CONFIG['unittest']

nefToken = ''
global nefHeaders

# Initiate Flask App
app = Flask(__name__)
app.secret_key = os.urandom(24)
app.emuUsername = CONFIG['emuUsername']
app.emuPassword = CONFIG['emuPassword']

```

Figure 10 - Parsing Configuration

There are functions to check the availability of components as well as to test connection to the NEF emulator, as shown in Fig. 11.


```

#check emu availability
def check_emus():
    try:
        response = requests.get(apiRoot, verify=False) #SEC removed certificate checking by adding verify=False
        print ("-----dummy API is accessible-----\n")
    except Exception as e:
        print ("-----dummy API not accessible-----\n")
    try:
        response = requests.get(nefEMU, verify=False) #SEC removed certificate checking by adding verify=False
        print ("-----EMU is accessible-----\n")
        print ("Logging in...")
        log_in_NEF_Emu()
    except Exception as e:
        print ("-----EMU is not accessible-----\n")

#IQB NetApp Authentication
def log_in_NEF_Emu():
    try:
        global nefHeaders
        credentials = {'username': app.emuUsername, 'password': app.emuPassword}
        print (credentials, end="\n\n")
        nefResponse = requests.post(nefEMU+'api/v1/login/access-token', data=credentials)
        token = nefResponse.json()
        nefToken = token
        print("nefToken:")
        print(nefToken, end="\n\n")
        nefHeaders = {"Authorization": token['token_type'] + ' ' + token['access_token']}
        print("nefHeaders:")
        print(nefHeaders, end="\n\n")
        nefResponse = requests.post(nefEMU+'api/v1/login/test-token', headers=nefHeaders)
        result = nefResponse.json()
        print("The NEF response:")
        print(result, end="\n\n")
    except Exception as e:
        raise e

```

Figure 11 - Connectivity Tests

After connection to NEF and Keycloak is tested, the provider IdPs stored in the CONFIG file are imported, and a Keycloak connector for each provider is created, as shown in Fig. 12.

```

#Configure clients for KeycloakOpenID
confproviders = CONFIG['providers']
providers = {}
for i in confproviders:
    providers[i] = {"keycloakRealmName": confproviders[i]['keycloakRealmName'], "keycloakSecretKey" : confproviders[i]['keycloakSecretKey']}

kc_oidc = {}
for i in providers:
    kc_oidc[i] = KeycloakOpenID(server_url=keycloakURL, client_id = i, realm_name = providers[i]['keycloakRealmName'], client_secret_key = providers[i]['keycloakSecretKey'])

```

Figure 12 - Providers Initialization

Wrapper Functions

Wrappers or decorators are functions that take other functions as an argument, therefore allowing the execution of code right before endpoint consumption in an elegant way. By defining the wrapper functions shown in Fig. 13, any endpoint can be wrapped using a wrapper function with the @ notation. The wrapper function's code will execute right before the wrapped function.

```

'''
~
Wrappers
-----
'''

#authorization
def require_oauth(f):
    @wraps (f)
    def decorated_function(*args, **kwargs):
        if 'access_token' not in session:
            return Response("User not authenticated.",status=401,mimetype="application/json")
        if kc_oidc[session['provider']].introspect(session['access_token']['access_token'])["active"] != True:
            return Response("User token not active.",status=401,mimetype="application/json")
        return f(*args, **kwargs)
    return decorated_function

#input checks
def require_data(f):
    @wraps (f)
    def decorated_function(*args, **kwargs):
        data=None
        try:
            data = request.get_json()
        except Exception:
            return Response("Bad form content",status=500,mimetype='application/json')
        if data == None:
            return Response("Bad request",status=500,mimetype='application/json')
        return f(*args, **kwargs)
    return decorated_function

def require_afId(f):
    @wraps (f)
    def decorated_function(*args, **kwargs):
        if request.args.get('afId') == None:
            return Response("Information incomplete. Missing NetApp identifier",status=500,mimetype="application/json")
        return f(*args, **kwargs)
    return decorated_function

def require_subId(f):
    @wraps (f)
    def decorated_function(*args, **kwargs):
        if request.args.get('subscriptionId') == None:
            return Response("Information incomplete. Missing NetApp identifier",status=500,mimetype="application/json")
        return f(*args, **kwargs)
    return decorated_function

def validate_TrafficInfluSub(f):
    @wraps (f)
    def decorated_function(*args, **kwargs):
        data = request.get_json()
        #note 1 for TrafficInfluSub NorthboundAPIs_NEF
        if not "afAppId" in data and not "trafficFilters" in data and not "ethTrafficFilters" in data:
            return Response("Information incomplete (note 1)",status=500,mimetype="application/json")
        #note 2 for TrafficInfluSub NorthboundAPIs_NEF
        elif not "gpsid" in data and not "ipv4Addr" in data and not "ipv6Addr" in data:
            return Response("Information incomplete (note 2)",status=500,mimetype="application/json")
        return f(*args, **kwargs)
    return decorated_function

```

Figure 13 - Wrapper Functions

An example is the logout endpoint. In order for an invoker to consume this endpoint, they need to be authenticated, which means having a valid OAuth2.0 token in their session. By using @require_oauth right before the logout function, the decorator function will ensure the user is authenticated. This implementation is shown in Fig. 14.

```

@app.route('/logout', methods=['GET'])
@require_oauth
def logout():
    try:
        kc_oidc[session['provider']].logout(session['access_token']['refresh_token'])
        session.clear()
    except Exception as e:
        return Response({'Unknown error occurred'},status=500,mimetype='application/json')
    return Response({'Logged out'},status=200,mimetype='application/json')

```

Figure 14 - Usage of a Wrapper Function

Continuous Authentication

In order to increase the robustness and security of the NetApp, two mechanisms have been implemented to restrict access of invokers by de-authenticating them forcefully. Invokers will be de-authenticated forcefully in case there is an attempt to access a non-existing endpoint, or an attempt to access an existing endpoint using a method that is not allowed as shown in Fig. 15.

```

...
Continuous Authentication |
...
@app.errorhandler(405)
def method_not_allowed(e):
    try:
        kc_oidc[session['provider']].logout(session['access_token']['refresh_token'])
        session.clear()
    except Exception as e:
        pass
    return Response({'De-authenticated due to misuse'},status=403,mimetype='application/json')

@app.route('/', defaults={'path': ''}, methods=['GET', 'HEAD', 'POST', 'PUT', 'DELETE', 'CONNECT', 'OPTIONS', 'TRACE', 'PATCH'])
@app.route('/<path:path>')
def catch_misuse(path):
    try:
        kc_oidc[session['provider']].logout(session['access_token']['refresh_token'])
        session.clear()
    except Exception as e:
        return Response({'Unknown error occurred'},status=500,mimetype='application/json')
    return Response({'De-authenticated due to misuse'},status=200,mimetype='application/json')

```

Figure 15 - Continuous Authentication

NEF API Consumption

Depending on the combination of endpoint and method, the correct function is invoked in order to consume NEF APIs, depicted in Fig. 16.

```

'''
-----
Intermediary Role Fulfillment
-----
'''

@app.route('/api/v1/3gpp-monitoring-event/v<version>/scsAsId/subscriptions', defaults={'subscriptionId': None}, methods=['GET', 'POST'])
@app.route('/api/v1/3gpp-monitoring-event/v<version>/scsAsId/subscriptions/<subscriptionId>', methods=['GET', 'PUT', 'DELETE'])
@require_oauth
def monitoring(version, scsAsId, subscriptionId):
    data = request.get_json()
    if request.method == 'GET':
        try:
            nefResponse = requests.get(nefEMM+'/api/v1/3gpp-monitoring-event/v'+version+'/'+scsAsId+'/subscriptions', json = data, params = request.args, headers=nefHeaders)
            if nefResponse.status_code == 204:
                return Response('', status=204, mimetype='application/json')
            else:
                return nefResponse.json()
        except Exception as e:
            raise e
        ...
    if subscriptionId == None:
        result = location_read_all_subscriptions()
    else:
        result = location_read_subscription(subscriptionId)
    return Response(str(result), status=200, mimetype='application/json')
    elif request.method == 'POST':
        status, result = location_create_subscription(data)
        return Response(str(result), status=200, mimetype='application/json')
    elif request.method == 'DELETE':
        status, result = location_delete_subscription(subscriptionId)
        return Response(str(result), status=200, mimetype='application/json')
    elif request.method == 'PUT':
        status, result = location_update_subscription(data, subscriptionId)
        return Response(str(result), status=200, mimetype='application/json')
    else:
        return Response('Bad Request', status=500, mimetype='application/json')
@app.route('/api/v1/3gpp-as-session-with-qos/v<version>/scsAsId/subscriptions', defaults={'subscriptionId': None}, methods=['GET', 'POST'])
@app.route('/api/v1/3gpp-as-session-with-qos/v<version>/scsAsId/subscriptions/<subscriptionId>', methods=['GET', 'PUT', 'DELETE'])
def qos(version, scsAsId, subscriptionId):
    data = request.get_json()
    if request.method == 'GET':
        try:
            nefResponse = requests.get(nefEMM+'/api/v1/3gpp-monitoring-event/v'+version+'/'+scsAsId+'/subscriptions', json = data, params = request.args, headers=nefHeaders)
            if nefResponse.status_code == 204:
                return Response('', status=204, mimetype='application/json')
            else:
                return nefResponse.json()
        except Exception as e:
            raise e

```

Figure 16 - Function Selection for NEF Communication

The functions invoked shown in Fig. 16 are 5 distinct functions shown throughout Fig. 17-21 that prepare and perform each request using the evolved5g NEF SDK.

```

'''
-----
NEF SDK Functionality
-----
'''

def location_read_all_subscriptions():
    netapp_id = CONFIG['netappId']
    host = emulator_utils.get_host_of_the_nef_emulator()
    token = emulator_utils.get_token()
    location_subscriber = LocationSubscriber(host, token.access_token)
    try:
        all_subscriptions = location_subscriber.get_all_subscriptions(netapp_id, 0, 100) #skip, limit
        print('\n', all_subscriptions, '\n')
        return all_subscriptions
    except ApiException as ex:
        if ex.status == 404:
            print("No active transcriptions found")
            return "No active transcriptions found"
        else: #something else happened, re-throw the exception
            raise

```

Figure 17 - Retrieve All Subscriptions

```

def location_create_subscription(data):
    expire_time = (datetime.datetime.utcnow() + datetime.timedelta(days=1)).isoformat() + "Z"
    netapp_id = CONFIG['netappId']
    host = emulator_utils.get_host_of_the_nef_emulator()
    token = emulator_utils.get_token()
    location_subscriber = LocationSubscriber(host, token.access_token)
    try:
        subscription = location_subscriber.create_subscription(
            netapp_id=netapp_id,
            external_id=data["externalId"],
            notification_destination=data["notificationDestination"],
            maximum_number_of_reports=data["maximumNumberOfReports"],
            monitor_expire_time=data["monitorExpireTime"]
        )
        print(subscription)
        return True, subscription
    except ApiException as ex:
        if ex.status == 409:
            print("\nThere is already an active subscription for UE with external id", data["externalId"], '\n')
            return False, "There is already an active subscription for UE with external id " + data["externalId"]
        else: #something else happened, re-throw the exception
            raise

```

Figure 18 - Create a Subscription

```

def location_read_subscription(subscription_id):
    netapp_id = CONFIG['netappId']
    host = emulator_utils.get_host_of_the_nef_emulator()
    token = emulator_utils.get_token()
    location_subscriber = LocationSubscriber(host, token.access_token)
    try:
        subscription = location_subscriber.get_subscription(netapp_id, subscription_id)
        print('\n', subscription, '\n')
        return subscription
    except ApiException as ex:
        if ex.status == 404:
            print("No active transcriptions found")
            return "No active transcriptions found"
        else: #something else happened, re-throw the exception
            raise

```

Figure 19 - Retrieve Subscription Info

```

def location_delete_subscription(subscription_id):
    netapp_id = CONFIG['netappId']
    host = emulator_utils.get_host_of_the_nef_emulator()
    token = emulator_utils.get_token()
    location_subscriber = LocationSubscriber(host, token.access_token)
    try:
        subscription = location_subscriber.delete_subscription(netapp_id, subscription_id)
        print("Deleted subscription with id: " + subscription_id)
        return True, subscription
    except ApiException as ex:
        if ex.status == 404:
            print("No active transcriptions found")
            return False, "No active transcriptions found"
        else: #something else happened, re-throw the exception
            raise

```

Figure 20 - Delete a Subscription

Testing

The system under test is the local deployment of the NEF Emulator, NetApp, Keycloak SaaS, a 3rd party (Postman) and a test server to receive callbacks.

The requests made through Postman shown in Fig. 23 have been saved in a collection and exported in .json format. The file NetApp.postman_collection.json that accompanies this report can be imported in Postman in order to test the NetApp functionality.

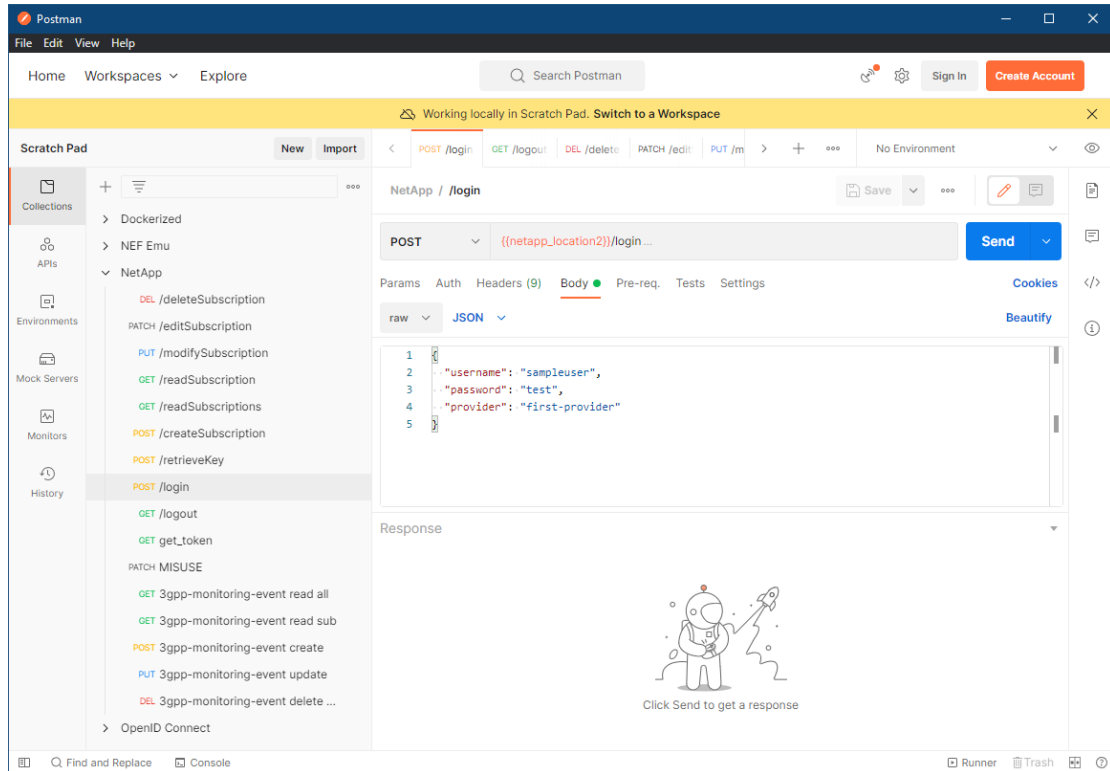


Figure 23 - Postman Requests Collection

Verification & Validation

Several unit tests have been implemented to check the NetApp's functionality, as shown in Fig. 24. These tests ensure that the system behaves properly, allowing invokers to authenticate and be authorized to consume NEF APIs only if they should. Unauthorized access should be blocked and incorrect / malicious usage of the endpoints should result in the de-authentication of a user.

References

- [1] X. Ji *et al.*, “Overview of 5G security technology,” *Science China Information Sciences*, vol. 61, no. 8, pp. 1–25, 2018.
- [2] I. Ahmad, T. Kumar, M. Liyanage, J. Okwuibe, M. Ylianttila, and A. Gurtov, “Overview of 5G security challenges and solutions,” *IEEE Communications Standards Magazine*, vol. 2, no. 1, pp. 36–43, 2018.
- [3] Q. Tang, O. Ermis, C. D. Nguyen, A. De Oliveira, and A. Hirtzig, “A Systematic Analysis of 5G Networks With a Focus on 5G Core Security,” *IEEE Access*, vol. 10, pp. 18298–18319, 2022.
- [4] N. Sakimura, J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore, “Openid connect core 1.0,” *The OpenID Foundation*, p. S3, 2014.
- [5] C.-Y. Li *et al.*, “Transparent AAA security design for low-latency MEC-integrated cellular networks,” *IEEE Transactions on Vehicular Technology*, vol. 69, no. 3, pp. 3231–3243, 2020.
- [6] A. M. Sanchez, A.-S. Charismiadis, D. Tsolkas, D. A. Guillen, and J. G. Rodrigo, “Offering the 3GPP Common API Framework as Microservice to Vertical Industries,” in *2022 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, 2022, pp. 363–368.