Flexible, Risk Aware Authentication System - Bank Use-Case - Final report

Diogo Matos - 102848, David Araújo - 93444

March 15, 2024

Table of Contents

- Table of Contents
- Introduction
- Services
  - Description
  - Roles/Agents and respective platforms
  - Operations
  - Authentication
    - Home Banking
    - Bank Officer Portal
    - Account Manager Portal
  - Access Control
  - Information Flow Control
  - Implementation
- Resource Servers
- The IdP
  - Endpoints
    - Authentication Endpoint
    - Resource Access Authorization Endpoint
    - Authentication Revoke Endpoint
    - Registration Endpoint
    - Login Endpoint
  - Basic flow
  - Service configuration file
  - Authentication
    - Password
    - EOTP
    - HOTP
    - TOTP
    - Portuguese Citizen Card
  - Risk based on context
  - Authorization

Introduction

The objective of this report is to present the architecture, description of the services, general authentication and authorization flows, a general risk tracking model, and the implementation of a Identity Provider (IdP) and three services that authentication and authorization play a crucial crucial role.

We start out by presenting the IdP, its general goal, its authentication and authorization features, and a quick overview of its implementation. Next, we go through every service providing a brief overview of their functionalities, policies and implementation. Then, we finish with some conclusions.

Services

Description

Authentication and authorization are paramount in banking services. The various parties involved in typical processes may have access to private information and/or perform critical operations. Excessive user access or identity theft can lead to significant economic impacts. Another aspect that is clearly evident in the case of banks is the conflict of interests, where an employee with sufficient privileges can benefit themselves or others associated with them.

Roles/Agents and respective platforms

In this project, we implement three platforms, each accessible to a set of users with a given role. The following are the roles (the agents): - *Client* - bank client that has a active account in the bank; - *Officer* - bank employee that works in one of the banks branch providing costumer support; - *Account Manager* - bank employee responsible to manage a set of client accounts.

From those three roles we get three distinct services, respectively: - *Home Banking* - *Bank Officer Portal* - *Account Manager Portal*

Operations

A bank operation is vast and complex, so we've selected a set of operations that will be available to the agents through their respective platforms. The following features will be implemented (note that their labels will be used later): - Home banking: - View loans, accounts movements, balances and general information; - Transfer money to another account. - Bank Officer Portal: - View all accounts on the local branch. It can see all information related to the bank itself and high-level balances; - Make a loan request to a given client from a fixed set of pre-defined plans; - Request a new account creation (pending acceptance by the selected manager). - Account management portal: - View all information about the accounts under management; - Accept loan requests (Must bide bank rate policies); - Delete accounts; - Change account branch.

Authentication

Home Banking

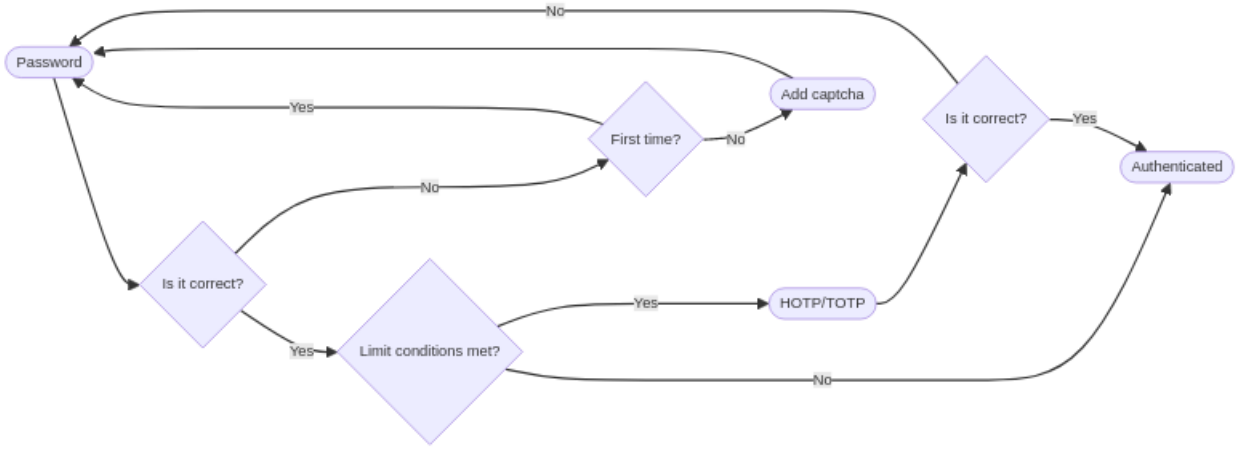

Figure 1: Home Banking authentication flow.

In normal cases the password is enough for the user to login, the full flow is followed when these limit conditions are met: - The IP from which the user is trying to login was never seen before; - Number of failed logins in the last 24 hours exceeds the set limit.

Bank Officer Portal

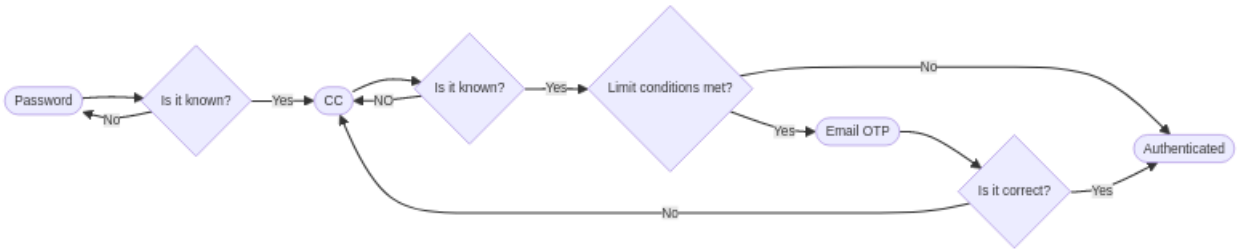

Figure 2: Bank Officer Portal authentication flow.

In normal cases the Citizens Card (CC) is enough for the user to login, the full flow is followed when these limit conditions are met: - The IP from which the user is trying to login was never seen before. Ensuring that if the CC is stolen and the access is not made from the bank's officer computer. - Every 1 week.

No logins in the platform can be made on a weekend.

Account Manager Portal

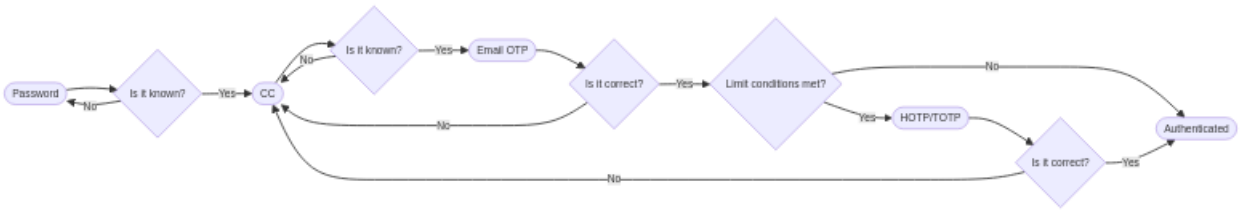

Figure 3: Account Manager Portal authentication flow.

The CC and the Email OTP are always requires. MFA fatigue can be ignored in this case because the user is too critical. The HOTP/TOTP is only required when these limit conditions are met: - It's a weekend or every day from 7pm to 7am - The IP from which the user is trying to login was never seen before; - In the last 24 hours the second step in the authentication (Email OTP) failed.

For each agent, distinct authorization criteria must be followed. Where possible, access control will be enforced by the IdP itself. In cases where this is not possible, the resource server will perform access control based on the information it stores and the access token provided by the client.

Access Control

	View account movements	View account balances	Make Payments	Request Account Creation	Transfer Money	Update Client Info	View Account Info	Request Loan	Accept Loan Request	Delete Account	Change Account Branch
Client	x	x	x	x	x	x	x	x	-	-	-
Bank Officer	-	x	-	x	-	-	x	x	-	-	-
Account Manager	x	x	-	x	-	-	x	x	x	x	x

Have access: **x** | Doesn't have access: **-**

More limitations: - A *bank officer* can't perform any action on accounts that he/she is titular; - An *account manager* can't manage account where he/she is titular; - An account deletion cannot be fully performed without explicit consent from the *client*. The client shall verify the action inside the Home Banking platform; - A *client* when performing either a payment or a transfer, must present a set of digits corresponding to four random-generated index of a multi-channel code that was given to the user when the account was created via email; - The *account manager* has a margin to adjust the loan request before accepting it, but it always must follow the bank's rate policies.

Information Flow Control

Regarding account management, information flow control is paramount since read/write operations should not be allowed to all agents at every action they take part in. We can implement the Bell-La Padula MLS Model to specify controls where it is not possible to read up or write down the hierarchical tree.

In actions such as *Request Account Creation* and *Request Loan*, the information follows the same pattern.

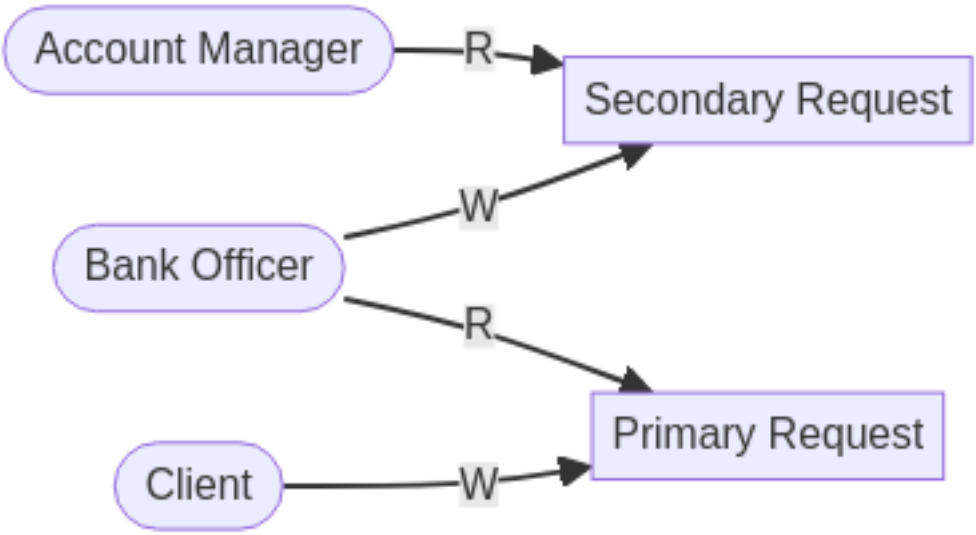

Figure 4: Request Account Creation and Request Loan information flow.

When performing other actions such as *Change Account Branch*, the information flows from the top management to the bottom, and to ensure integrity of the action, the most appropriate model to implement is the Biba Integrity Model which ensures that data can only be written down and read above.

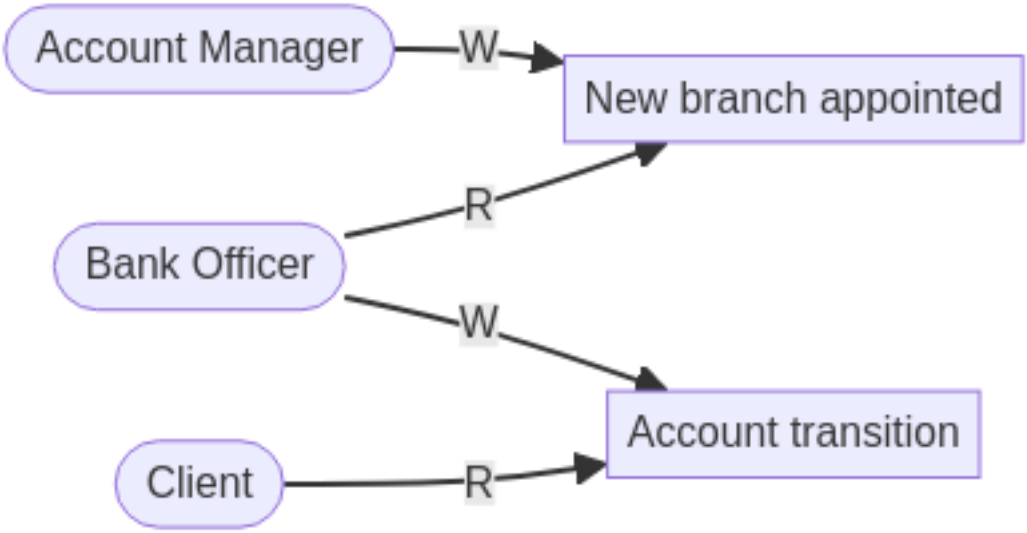

Figure 5: Change Account Branch information flow.

Implementation

The services are implemented using Flask and are located in the `/services` directory.

Resource Servers

To support the services, there's three resource servers: - *Account information server* - stores and provides all the information about the bank's accounts. - *Payment server* - enables clients to perform, upon request, money transfers and payments. - *Loan server* - manages loan-related information.

These are small Flask servers implemented in the `/resource_servers` directory.

The IdP

The goal of the IdP is to identify and authenticate users and limit their access to resources using OAuth 2.0.

Service/application registration in the IdP should be configured through a configuration file without the need for any additional code, within the capabilities of the IdP.

Endpoints

Authentication Endpoint

- URL: `/oauth/authorize`
- Method: `GET`, `POST`
- Arguments: client_id (str), response_type (str), scope (str)
- Example: (GET) `http://127.0.0.1:5000/oauth/authorize?response_type=code&client_id=<client_id>&scope=`
- Returns: redirect to registered redirect Url with authorization_code (str)
- Description: A `GET` request to this endpoint is used by clients to initiate the OAuth 2 authentication process. When a client wants to authenticate a user, it redirects the user's browser to this endpoint. The IdP presents the user with a login page (redirection to `/` with `next` URL argument pointing back to this endpoint). After successful authentication, the user is redirected is presented with a consent form. The response to this form represents the usage of the `POST` method, the result of this request when the user consents de access, the browser is redirected back to the client with an authorization code (a random unique string) as the `code` URL argument.

Resource Access Authorization Endpoint

- URL: `/oauth/token`
- Method: `POST`
- Arguments: code (str), client_id (str), client_secret (str), scope (str), include_refresh_token (int), grant_type (str)
- Example: `curl -u <client_id>:<client_secret> -XPOST http://127.0.0.1:5000/oauth/token -F grant_type=authorization_code -F scope=profile -F code=<authorization_code>`
- Returns: access_token (JWT), refresh_token (str optional)
- Description: This endpoint is used by clients to exchange the authorization code obtained from the authentication process for an access token and a refresh if the `gran_type` is equal to `authorization_code`. The refresh token is only included if the `include_refresh_token` argument is set to a number different from 0. If the `gran_type` is equal to `refresh_token`, the `code` argument is not the authorization code but the refresh token. This refresh token is then validated and a new access token is provided. Note that, as discussed later in Authorization, the refresh token can only be requested once, so when `gran_type` is equal to `refresh_token` it never generates a new refresh token.

Authentication Revoke Endpoint

- URL: `/oauth/revoke`
- Method: `POST`
- Arguments: access_token (str), client_id (str), client_secret (str)
- Example: ...
- Returns: status

- Description: The services, when the user logs off or something happens that must finish the current session, the services (the clients) should send a request to this endpoint in order to revoke the access token.

Registration Endpoint

- URL: `/registration`
- Method: `POST`, `GET`
- Argument: client_id (str), client_secret (str), email (str)
- Example: `curl <client_id>:<client_secret> -XPOST http://127.0.0.1:5000/register --header "Content-Type: application/json" --data '{"email":"manel@xpto.pt"}'`
- Return: status
- Description: We're talking about services (see next section) that shall not be accessible by everyone without a first intent from the bank. So, the user registry should be integrated within authorized services. This endpoint receives a `POST` request with the credentials of the client and information about the user. The user will receive an email with a URL that will send him/her to a IdP page (this same endpoint but using a GET request) where the registration process will be finished (e.g. password and MFA setup) by the user itself. Some more details are provided in Authorization.

Login Endpoint

- URL: `/`
- Method: POST
- Argument: client_id (str), client_secret (str), ... tbd
- Example:
- Return: status
- Description: We're talking about services (see next section) that shall not be accessible by everyone without a first intent from the bank. So, the user registry should be integrated within authorized services. This endpoint shall receive from the service very little information related with the bank and basic user information. The user will receive an email with a URL that will send him/her to a IdP page where the registration process will be finished (e.g. password and MFA setup).

Basic flow

Figure 6: Authentication/authorization flow.

Service configuration file

Each service can be registered and configured upon the IdP load from a `YAML` formatted file like the one bellow.

The main fields, and what varies in significance between the services, are the `levels` and `limit-conditions` fields. The `levels` field defines the supported authentication methods for the service and ensures that the user authenticates using these methods. This setup allows users to specify different methods for each service.

The `limit-conditions` field, as detailed in the various flows in Authentication, specifies the events that trigger the system to enforce re-authentication for a user. This approach enables administrators to precisely define the conditions that trigger these events, thereby enhancing security.

```yaml
name: "Bank Managing for managers"
client_id: ccccccccccccccccccccccccc
client_secret: BkcHH9B4CX0IdrJwofdMAIf0gMoPjdykvPzINW1gPZzDxtHt
uri: http://127.0.0.1:5002/
redirect_uris:
  - http://127.0.0.1:5002
auth:
  # mandatory authentication step
  levels:
    - cc
    - eotp

  # condition authentication step
  limit-conditions:
    key: eotp
    limit: days=1
```

```
    behavior: hotp

token_lifetime: 14400 # seconds (4 hours)
internal_authorization:
  - create_clients
authorization:
  - 1 # access accounts
  - 2 # access loans
```

The following parameters can be customized: - `name` - Name of the service; - `client_id` - Client identifier; - `client_secret` - Client secret; - `uri` - Service URI; - `redirect_uris` - Redirection URIs used to redirect after a successful GET request to `/authorize`. This current version just support one redirect URL; - `auth` - Configuration of the authentication (login) flow. The `levels` has all the login steps. In `limit_condition` one condition can be defined that will be checked apart from the default ones; - `token_lifetime` - Lifetime of the access tokens; - `internal_authorization` - Set of identifiers of internal authorization, i.e. actions directly targeted at the IdP; - `authorization` - Set of identifiers of resource servers that this service is authorized to access.

Each client is identified by; - *client_id* - random string (in the client config for convenience). - *client_secret* - random string (in the client config for convenience). - *issued_at* - timestamp. - *expired_at* - timestamp.

Also, has attached to it the following metadata: - *client_name* - string - *client_uri* - string (URI) - *grant_types* - list of strings. This version only support 'authorization_code' and 'refresh_token'. - *redirect_uris* - list of strings. - *user_auth_method* - dict with the authentication flow. - *token_endpoint_auth_method* - string. Only supports 'client_secret_basic'. - *authorization* - list of integers. List of authorized resource servers - *internal_authorization* - list of strings. List of internal resource accesses. - *token_lifetime* - integer.

Authentication

As almost everything in the IdP, the authentication is configurable. In the client `yaml` configuration file, a field for authentication is specified in order to provide the IdP with the authentication flow.

Overall, the IdP support four authentication methods.

Password

Password authentication is a widely used method for verifying the identity of users attempting to access a system or service. In this process, users are required to provide a unique combination of characters. Upon login, the system hashes the entered password and compares it to the stored hash value to determine authentication. Password authentication relies on the assumption that only authorized users possess the correct password, making it a fundamental yet often vulnerable. To enhance security, password need to have at least 12 alphanumeric characters, and its advised to be complemented with at least one of the authentication methods.

To use password for authentication, use `password` in the configuration file.

EOTP

Email One-Time Password is the simplest type of OTP. Upon login the server was the user to input a code that was sent to the user's e-mail.

In the employed implementation, the OTP is a random number between 100000 and 999999 (6 digits) and it's sent to the user's email using Google's Gmail API.

To use EOTP for authentication, use `eotp` in the configuration file.

HOTP

HMAC-based One-Time Password is a type of OTP algorithm first introduced in 2005 and it is pretty much connected with the first implementations of OAUTH. Basically, both the authentication server (the IdP in

our case) and the user have a token (OTP) generator synchronized by a counter. This generator is initialized using mainly three parameters: - OTP size; - Secret key; - Hash function; - Initial counter value;

We're using, respectively: - 6; - 8 byte random value encoded in base64; - SHA-1; - 2 for the client, 1 for the server;

On the client-side, every time he/she needs a OTP, the counter should be incremented and the generated OTP provided to the Server. On the server-side, if the counters are synchronized, the value return by the generator should be the same. But of course, there's no way to make sure both counters are synchronized, so the client between each login, can increment the counter some times. The server only stores the value of the counter of the last login and upon a new one, it will check if the provided OTP is equal to one of next `n` iterations (we use 1000), updating the stored counter to the value that matches. The case, where the generator can go out of sync is if the user increments the counter more than `n` times between logins.

HOTPs can be generated by physical or software solutions. Nowadays, this job is made by a MFA application, usually installed in a mobile device, capable to manage multiple OTPs for different services. We've followed that path and during the registration process, a QR Code is provided for the user to scan using an authenticator app, enabling the user to setup the OTP very easily. A MFA authentication app like Aegis (an open-source solution) can be used.

To use HOTP for authentication, use `hotp` in the configuration file.

TOTP

Time-based One-Time Password is a type of OTP algorithm first introduced in 2008 and it is pretty much like the previously presented HOTP. But this solution completely the synchronization issues, by introducing time into the algorithm. Basically, both the authentication server (the IdP in our case) and the user have a token (OTP) generator synchronized by a counter that is updated periodically. The TOTP generator is initialized by the following parameters: - Initial counter value; - Time interval to calculate a new counter value.

We're using, respectively: - 0; - 30 seconds.

For the user to setup TOTP in a MFA authentication app, the IdP provides a QR Code that when scanned by the app it setups the TOTP generator. A MFA authentication app like Aegis (an open-source solution) can be used.

To use HOTP for authentication, use `totp` in the configuration file.

Portuguese Citizen Card

For this authentication method to work, the user needs the Autenticação.Gov plugin, a smart card reader and the `cc_middleware` API.

The `cc_middleware` acts as a resource server but is only accessed by the IdP indirectly using the user's browser. To access it an access token generated by the IdP needs to be provided, this JWT token has a 1 minute lifetime and has the identity of the user and an `access_whitelist` (see more in Authorization) with the "cc" identifier in it. This means that the IdP gave permission for the browser to access the middleware. The `cc_middleware` has only one endpoint `/sign` that signs a given message using the Citizen Card (CC) and returns both the signature and the certificate of the CC. This is used to make a bridge between the browser and the CC.

Ideally, the registration should be setup from scratch with the CC associated with it. Here we're letting the user during the registration setup, to provide his/her CC identification. The registration step with the CC goes as follows: 1 - The IdP renders the `html` page directly with the request to the `cc_middleware` with the token and random 64 bit challenge encoded in `base64`; 2 - The user, when ready, clicks on the button to continue; 3 - The browser sends a POST request to `/sign` of the local `cc_middleware`; 4 - The `cc_middleware` uses the CC to sign the challenge and to retrieve its certificate; 5 - The browser sends back to the IdP, same URL but with a POST, the signature and the certificate. 6 - The IdP verifies the signature and checks if the certificate was issued by one of the entity certificates that it trusts. These certificates are

from the certification authority of the “Autenticação do Cartão de Cidadão”. 7 - The IdP, if the verification is successful, stores the certificate associated with the user.

The authentication flow is pretty much the same, but it uses directly the previously stored certificate to verify the signature.

To use Citizen Card for authentication, use `cc` in the configuration file.

Risk based on context

Each user, based on their role, follows a specific set of authentication protocols. This segregation of roles is based on their inherent risk.

This risk is determined by the context and history of their interactions with the app. Certain events during their interaction can trigger the application to ask for re-authentication. This is done cyclicly until the user is properly authentication according to the services flow.

These triggers, as detailed in Authentication flows, are defined as *limit conditions*. One such condition is the IP addresses from which the user connects to the application. If a user connects from a never-before-seen IP address, they will be prompted to take an extra step in re-authenticating.

```
@staticmethod
def get_next_level(login_current_step, levels):
  return levels[(levels.index(login_current_step) + 1) % len(levels)]
```

The context is stored for each user and updated with each new authentication event. These context events include the following keys: - User identification - Timestamp of the authentication event - IP address from which the event was triggered - Successfulness of the event - Authentication method used

This context needs to be dynamic and adapt according to the role and service the user is using since trigger events are fixed for each service but differ between them. The only limit condition shared by all services is the detection of a never-before-seen IP address for a given user.

```
    @staticmethod
    def next_authentication_step(user: User, current_ip: str, client_id: str):
      # 1. Check IPs

      # 2. Check service specific flow
      flow = getattr(Context, user.role + "_flow")
      flow(user, conditions, levels)
      ...

    @staticmethod
    def client_flow(user, conditions):
      ...

    @staticmethod
    def officer_flow(user, conditions):
      ...

    @staticmethod
    def manager_flow(user, conditions):
      ...
```

After detecting such an IP address, the role is used to call the service-specific limit condition check. If the limit is triggered, the service will request the authentication method specified in the `behavior` for the user to comply.

Authorization

Access authorization to resources is done on a two-staged check, one at IdP level and another at the resource server level. Nevertheless, the final decision is always made by the resource server. A client, when in possession of a authentication code, can request a access token. In that request it specifies its credentials (`client_id` and `client_secret`) and the scope that should match the scope of the authorization code. The idp support one scope, `profile`, that provides enough information (`email` and `role`) about the user for the resource server to decide if the person requesting a resource is authorized or not. Based on the client requesting an access token, the IdP will provide also a access whitelist of resource servers IDs that can be accessed by the that user. All that information is introduced in a JWT token used as access token.

A JWT access token has the following attributes: - `email` - `role` - `access_whitelist` - `iat` - `exp`

Example:

```
{
  "alg": "RS256",
  "typ": "JWT"
},
{
  "email": "client@xpto.com",
  "role": "client",
  "access_whitelist": [
    1,
    2,
    3
  ],
  "iat": 1715520974,
  "exp": 1715524574
}
```

When a resource server receives a request, it will check the token integrity (signature, validity, etc.) and check both the authorization at user level, based on `email` and `role`, and also at service layer, based on the `access_whitelist`. If both pass, the access is granted. So, we can say that we've followed a more authentication/identification driven approach with the IdP.

There is also another special type of authorization that we call "internal authorization" that takes place when a service wants to manage in some way the IdP itself. In our use case, that happens when a new client is added by a manager to the system. The "Bank Managing for manager" service has a special internal authorization named `create_clients` that will enable it to create clients. That request must come with the service credentials and a valid token for extra authorization and, especially, accountability.

Bellow we can see a snippet of a service IdP configuration file with both authorization fields, `authorization` is mandatory and it must contain valid IDs of resource servers, i.e. must be unique and the resource servers themselves must know their IDs.

```
internal_authorization:
  - create_clients
authorization:
  - 1
  - 2
```

The access tokens have a lifetime (i.e. an expiration time), this value can be adjusted in the configuration file for each client service in the field `token_lifetime` in seconds. The value used for each client should be the result of a analysis of the service risk and the overall processes behind it that shall be facilitated.The access tokens can be refreshed. To do so, the first access token request must have the `include_refresh_token` field set to a value different than 0. While the token is still valid, with a margin of 1/10th of the token lifetime, the client can request a new one with a refreshed *issued at* and *expires_at* time. Due to the nature of the use cases we've decided to take a more conservative path in terms of the usage of refresh tokens. So, a refresh

token can only be requested, and consequently generated, on the first access token request, i.e. using the authorization grant. In another words, tokens can only be refreshed once, after that, the user behind the client must login again. This limit by a lot the impact of a refresh token robbery, and still does not impact much the service because if the service is not critical at all, the service simply can have tokens with a large lifetime. For example, on average banks branches opened for costumer support have 7 hour schedules, the tokens for our bank managing services have a 4 hour lifetime, if we include the refreshed token, it sums up to 8 hours. On the other hand, the home banking tokens will make sure that the user cannot have a session active for more than 20 minutes (two tokens of 10 minutes).

The reason why we prefer to provide two tokens, being the second one the product of refresh of the first, is that, usually services just verify and consequently request tokens after a request from the user. So, for example, by giving a token with 20 minutes with no refresh, the token will always be valid for 20 minutes, while two with 10 minutes makes sure that after the first expires the user is still active (due to the small refresh window) and, certainly, there's the need to refresh the token

In terms of requests are only three. First, the user is redirected to a URL like this one:

```
http://127.0.0.1:5000/oauth/authorize?response_type=code&client_id=<client_id>&scope=profile
```

The user will be prompted to introduce its credentials (see authentication) and if everything is successful, the user will be redirected back to the service with a URL argument names `code` that is the authorization code.

Next, the service used the authorization code along with its credentials (`client_id`, `client_secret`) to request an access token:

```
curl -u <client_id>:<client_secret> -XPOST http://127.0.0.1:5000/oauth/token -F include_refresh_token=1
```

The access token can later be refreshed using:

```
curl -u <client_id>:<client_secret> -XPOST http://127.0.0.1:5000/oauth/token -F grant_type=refresh_toker
```