FastAPI sur Azure Container Instance : Implémentation de l'authentification AAD et du protocole TLS

maelys.rimbert

November 2023

Table des matières

1 Introduction

Ce tutoriel concerne les API créées avec FastApi, utilisant Uvicorn et hébergées dans une instance de conteneur Azure.

Nous verrons, dans un premier temps, comment mettre en place l'authentification Azure en local avec FastAPI. Dans un second temps, nous verrons comment implémenter le protocole TLS afin de pouvoir utiliser notre authentification dans le contexte d'une instance de conteneur Azure.

Les points de terminaison résultants ne pourront être utilisés que par un utilisateur authentifié et appartenant au locataire de l'API. Pour d'autres cas d'usages, se référer à la documentation fournie dans la section **Sources**.

2 Implémenter et utiliser l'authentification avec fastapi-azure-auth

Prérequis :
* Python
* Paquet fastapi-azure-auth
* Paquet uvicorn

2.1 Configuration sur Azure [1]

2.1.1 Côté Backend

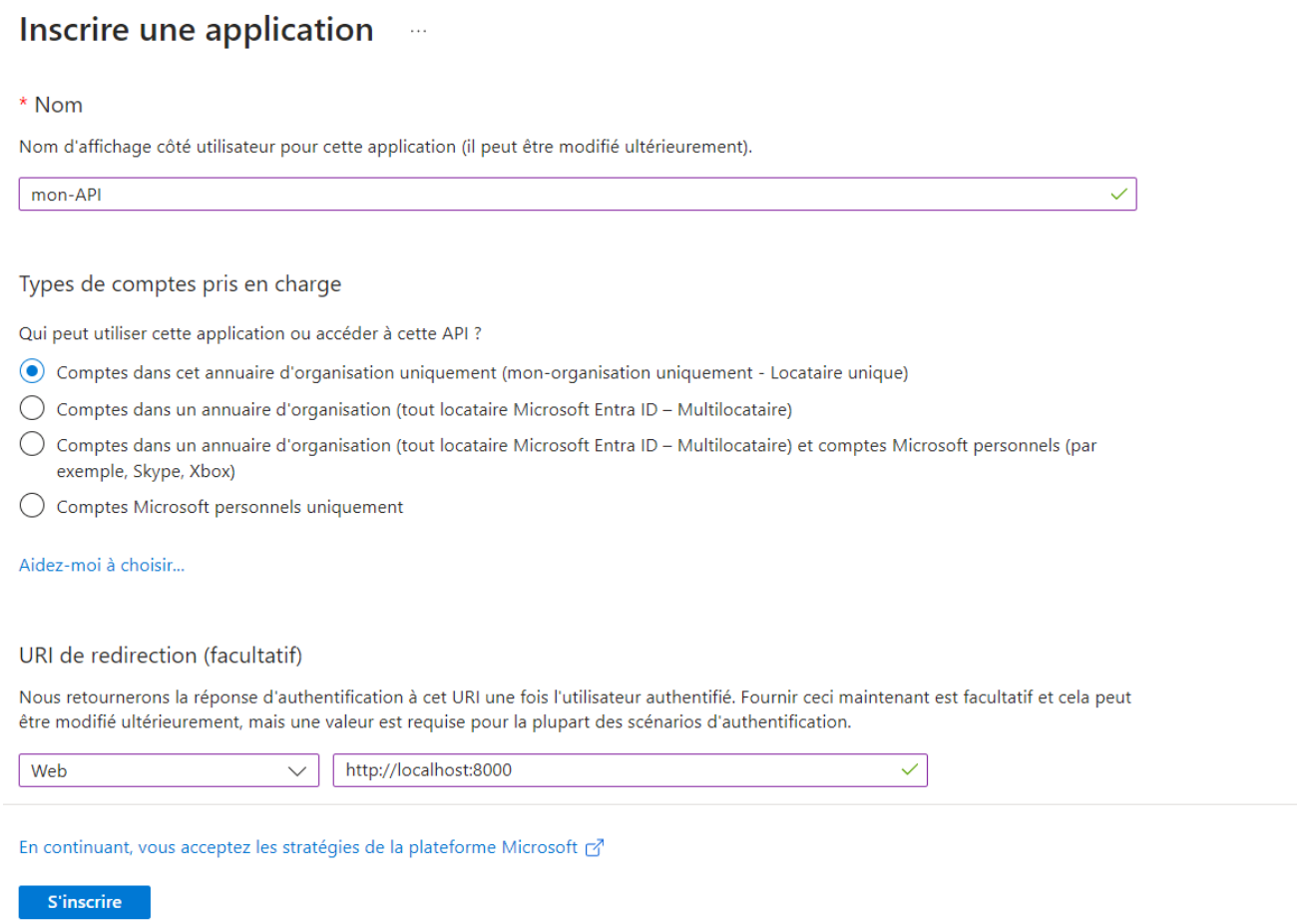
Inscrire l'application : Rendez vous sur Azure -> Azure Active Directory -> App registrations [3], et créez une nouvelle inscription.

Donnez-lui un nom qui corresponde au projet ; Azure présentera ce nom à l'utilisateur.

— Types de comptes pris en charge : Locataire unique

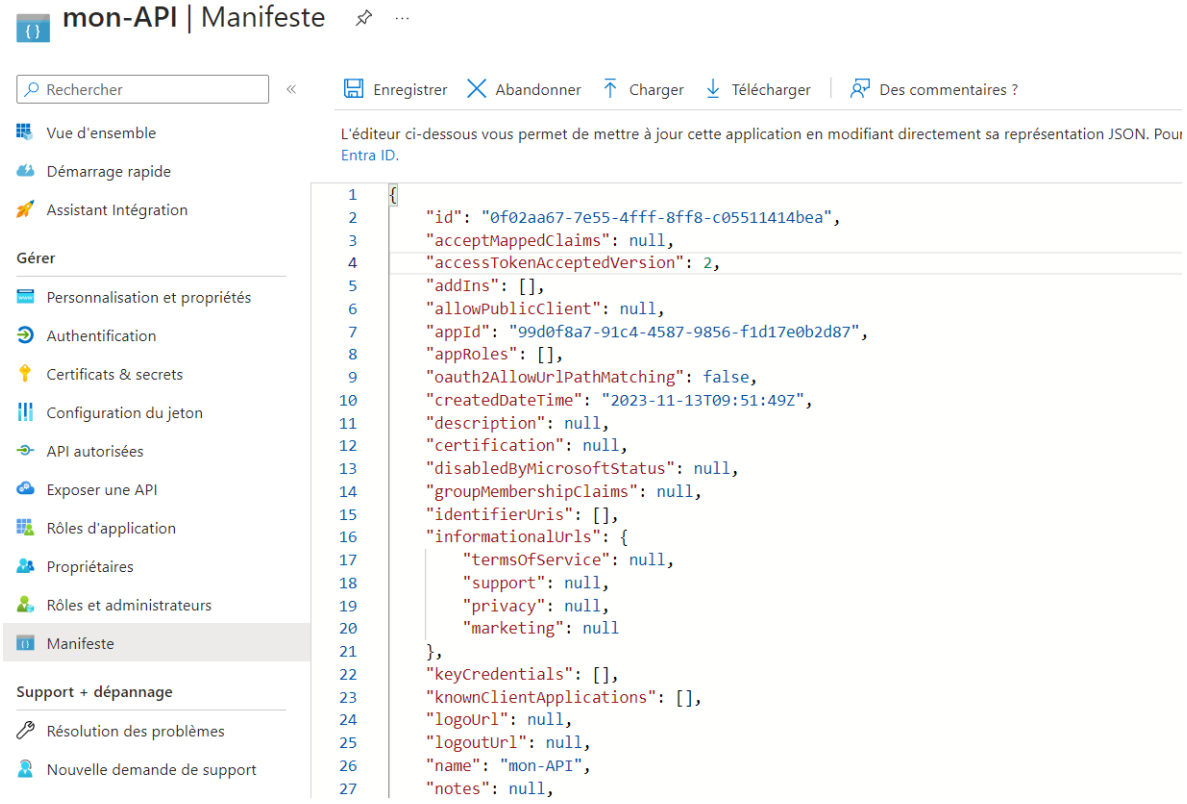
— URI de redirection : Choisissez "Web" et "http ://localhost :8000" comme valeur

Appuyez sur 'S'inscrire'.



Changer la version du token à "v2" : Dans le menu de gauche, cliquez sur Manifeste et trouvez la ligne "accessTokenAcceptedVersion". Changez sa valeur de "null" à "2".

Appuyez sur 'Enregistrer'.



Noter les identifiants d'application Retournez à la "Vue d'ensemble" depuis le menu de gauche.

Copiez l'ID d'application (client) en tant que APP_CLIENT_ID, et l'ID de l'annuaire (locataire) en tant que TENANT_ID, nous en aurons besoin plus tard. Il est préférable de stocker ce genre de variables dans un fichier .env :

.env

```
TENANT_ID=
APP_CLIENT_ID=
OPENAPI_CLIENT_ID=
```

Ajouter une étendue d'application :

— Allez sur "Exposer une API" dans le menu de gauche.

— Cliquez sur "+ Ajouter une étendue".

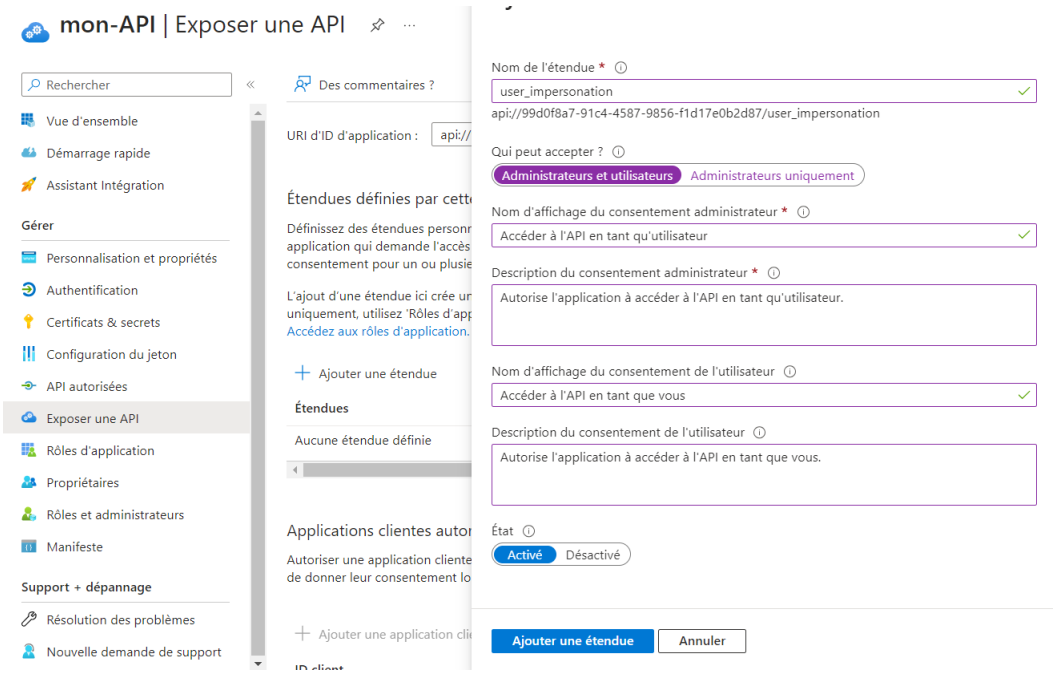
— Laissez l'URI ID suggéré et cliquez sur "Enregistrer et continuer".

Ajoutez une étendue nommée user_impersonation qui peut être acceptée par les administrateurs et les utilisateurs.

Vous pouvez utiliser les descriptions suivantes :

Accéder à l'API en tant qu'utilisateur
Autorise l'application à accéder à l'API en tant qu'utilisateur.

Accéder à l'API en tant que vous
Autorise l'application à accéder à l'API en tant que vous.



Votre API est désormais accessible côté backend.

2.1.2 Côté Swagger

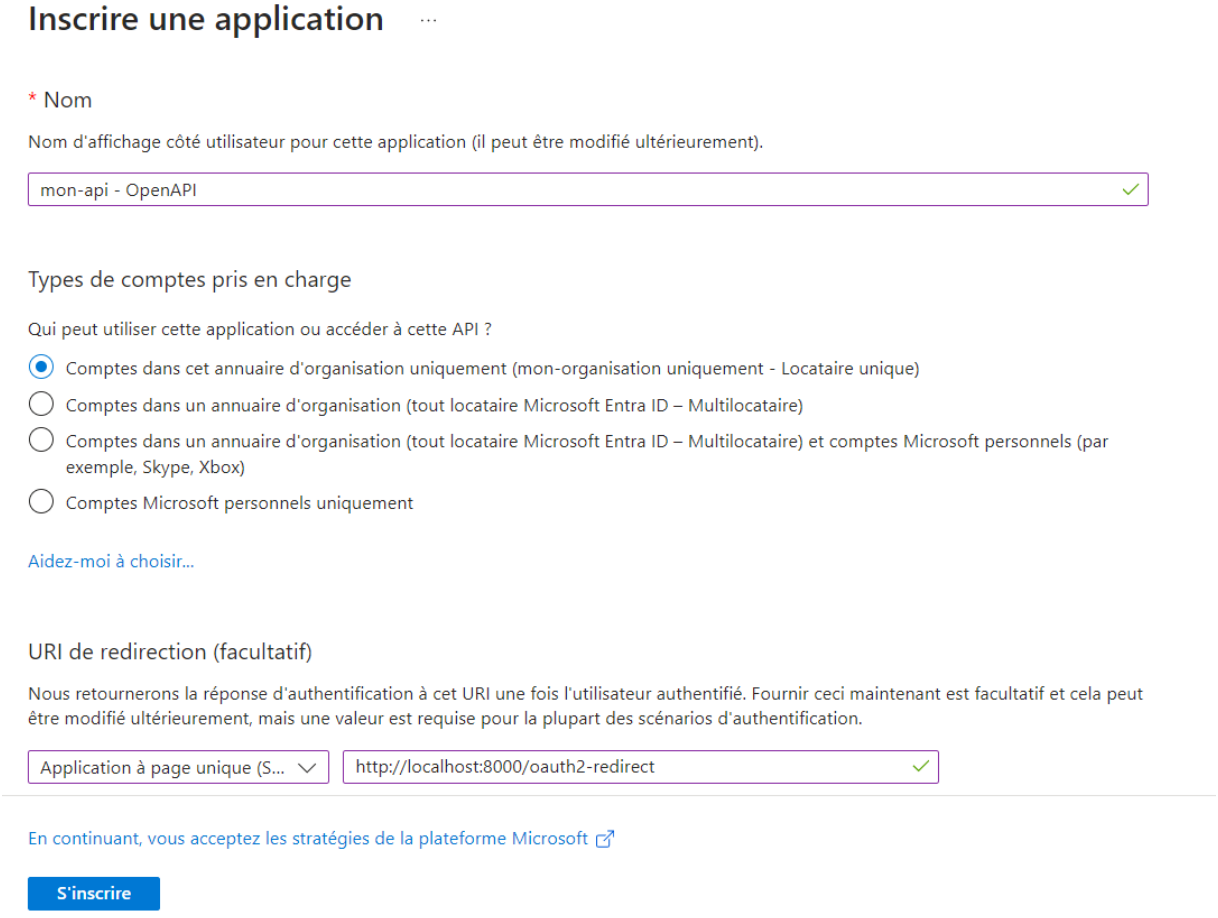
Afin de pouvoir utiliser l'API directement avec le Swagger, il est nécessaire de créer une nouvelle application.

Inscrire l'application : Comme pour la première partie, il faut créer une inscription d'application pour OpenAPI. Rendez vous sur Azure -> Azure Active Directory -> App registrations [3], et créez une nouvelle inscription.

Donnez-lui le même nom qu'à la première, mais en ajoutant " - OpenAPI" à la fin.

— Types de comptes pris en charge : Locataire unique

— URI de redirection : Choisissez "Application à page unique (SPA)" et "http ://localhost :8000/oauth2-redirect" comme valeur



Changer la version du token à "v2" : Comme précédemment, dans le menu de gauche, cliquez sur Manifeste et trouvez la ligne "accessTokenAcceptedVersion". Changez sa valeur de "null" à "2".

Appuyez sur "Enregistrer".

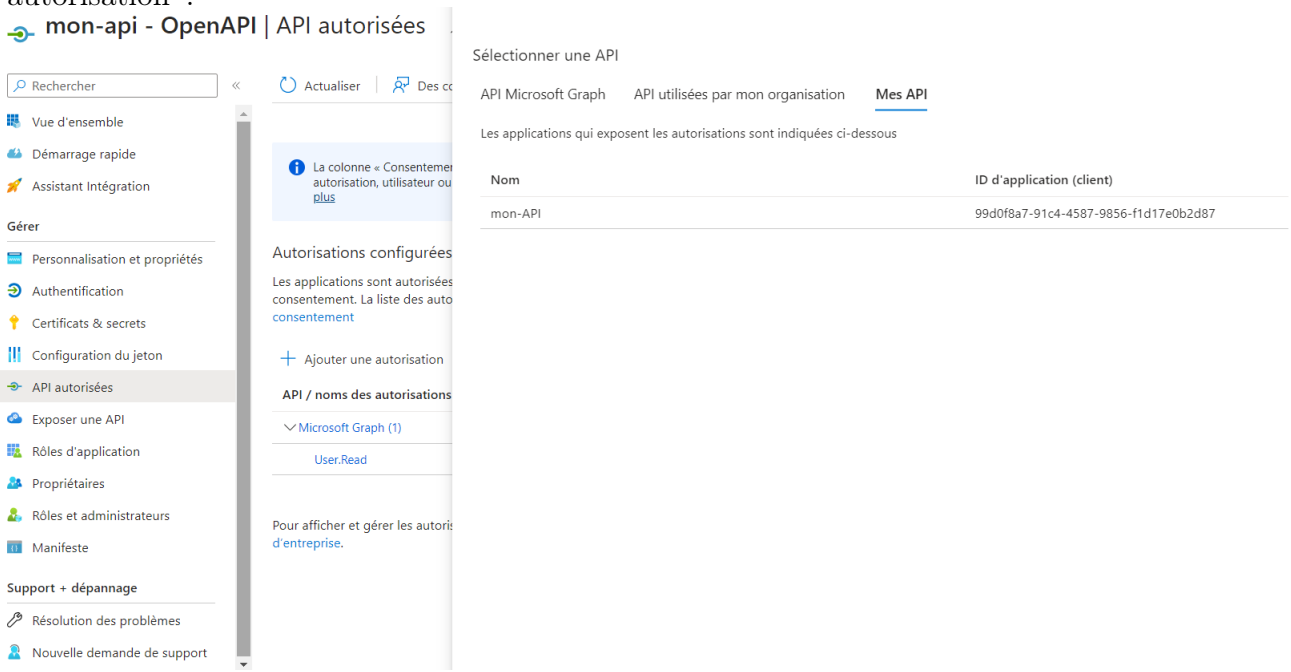
Noter l'identifiant d'application : Retournez à la "Vue d'ensemble" depuis le menu de gauche.

Copiez l'ID d'application (client) en tant que OPENAPI_CLIENT_ID.

```
.env

TENANT_ID=
APP_CLIENT_ID=
OPENAPI_CLIENT_ID=
```

Autoriser OpenAPI à communiquer avec le backend : Il faut ajouter des permissions à l'application OpenAPI. Dans le menu de gauche, sélectionnez "API autorisées", puis "Ajouter une autorisation".



Sélectionnez votre API (ici "mon-API"), puis l'autorisation "user_impersonation" créée plus tôt, et enfin "Ajouter des autorisations".

Il ne reste plus qu'à configurer notre FastAPI pour utiliser l'authentification Azure.

2.2 Configuration sur FastAPI [2]

Pour illustrer cette documentation, nous utiliserons une API très simple et basée sur un unique fichier main.py. Bien entendu, le processus est le même pour une API de plus grande envergure.

2.2.1 Mise en place

Commencez par créer un fichier .env contenant les variables récupérées plus tôt :

```
APP_CLIENT_ID=
TENANT_ID=
OPENAPI_CLIENT_ID=
```

Créez votre fichier main.py :

```
from fastapi import FastAPI
import uvicorn

app = FastAPI()


@app.get("/")
async def root():
    return {"message": "Hello World"}

if __name__ == '__main__':
    uvicorn.run('main:app', reload=True)
```

Lancez l'application et vérifiez que tout fonctionne sur http://localhost:8000/docs.
Attention : Votre application doit utiliser le port configuré dans Azure Active Directory !

2.2.2 Ajouter les paramètres

Ajoutez une classe "Settings" qui contiendra les paramètres de votre application. Le code suivant utilise le paquet "dotenv" pour récupérer les valeurs dans le fichier .env, mais vous pouvez utiliser la méthode que vous préférez.

```
from os import getenv

from dotenv import load_dotenv
from pydantic import AnyHttpUrl

_ = load_dotenv()

class Settings():
    BACKEND_CORS_ORIGINS: list[str | AnyHttpUrl] = [
        'http://localhost:8000',
        '127.0.0.1'
        ]
    OPENAPI_CLIENT_ID: str = getenv('OPENAPI_CLIENT_ID')
    APP_CLIENT_ID: str = getenv('APP_CLIENT_ID')
    TENANT_ID: str = getenv('TENANT_ID')

settings = Settings()
```

2.2.3 Configurer CORS

Configurons CORS. Ajoutez le code suivant dans votre main.py :

```
from fastapi.middleware.cors import CORSMiddleware

if settings.BACKEND_CORS_ORIGINS:
```

```
    app.add_middleware(
        CORSMiddleware,
        allow_origins=[str(origin)
                       for origin in settings.BACKEND_CORS_ORIGINS],
        allow_credentials=True,
        allow_methods=['*'],
        allow_headers=['*'],
    )
```

2.2.4 Configurer FastAPI

Ajoutons quelques paramètres à notre application FastAPI :

```
app = FastAPI(
    swagger_ui_oauth2_redirect_url='/oauth2-redirect',
    swagger_ui_init_oauth={
        'usePkceWithAuthorizationCodeGrant': True,
        'clientId': settings.OPENAPI_CLIENT_ID,
    },
)
```

Le paramètre swagger_ui_oauth2_redirect_url pour la redirection doit être le même que celui qui est configuré dans Azure Active Directory. Le swagger_ui_init_oauth est un ensemble de propriétés standards OpenAPI [4].

Nous avons utilisé 2 propriétés : usePkceWithAuthorizationCodeGrant, qui est le flux d'authentification ; clientId est l'ID d'application (client), qui permettra de compléter automatiquement un champ pour les utilisateurs finaux.

2.2.5 Implémenter FastAPI-Azure-Auth

Importez SingleTenantAzureAuthorizationCodeBearer de fastapi_azure_auth et configurez-le. Pour cela, vous pouvez utiliser la classe Settings :

```
from fastapi_azure_auth import
    SingleTenantAzureAuthorizationCodeBearer

class Settings(BaseSettings):
    BACKEND_CORS_ORIGINS: list[str | AnyHttpUrl] = [
        'http://localhost:8000',
        '127.0.0.1'
        ]
    OPENAPI_CLIENT_ID: str = getenv('OPENAPI_CLIENT_ID')
    APP_CLIENT_ID: str = getenv('APP_CLIENT_ID')
    TENANT_ID: str = getenv('TENANT_ID')
    SCOPE_DESCRIPTION: str = "user_impersonation"

    @property
    def SCOPE_NAME(self) -> str:
        return f'api://{self.APP_CLIENT_ID}/{self.SCOPE_DESCRIPTION}
            '
```

```
    @property
    def SCOPES(self) -> dict:
        return {
            self.SCOPE_NAME: self.SCOPE_DESCRIPTION,
        }

azure_scheme = SingleTenantAzureAuthorizationCodeBearer(
    app_client_id=settings.APP_CLIENT_ID,
    tenant_id=settings.TENANT_ID,
    scopes=settings.SCOPES,
)
```

On passe le "app_client_id" en tant qu'ID d'application Backend, le "tenant_id" en tant qu'ID de locataire, et pour finir, nos étendues. Nous y reviendrons plus tard.

2.2.6 Charger la configuration OpenID au démarrage

En ajoutant le point de terminaison on_event('startup'), il est possible de charger la configuration OpenID immédiatement, plutôt que lors de l'authentification du premier utilisateur. Ce n'est pas obligatoire, mais un peu plus rapide. Après 24 heures, la configuration sera considérée comme périmée, et se mettra à jour lors de la prochaine requête utilisateur. Il est également possible d'utiliser des tâches d'arrière-plan pour effectuer le rafraîchissement, si vous préférez.

```
@app.on_event('startup')
async def load_config() -> None:
    """
    Load OpenID config on startup.
    """
    await azure_scheme.openid_config.load_config()
```

2.2.7 Ajouter l'authentification aux points de terminaison

Dans FastAPI, il y a deux façons d'ajouter des dépendances : Depends() ou Security(). Security() a une propriété supplémentaire appelée "scopes". FastAPI-Azure-Auth supporte les deux méthodes, mais en utilisant Security() vous pouvez également verrouiller vos vues API selon l'étendue.

Faisons cela :

```
from fastapi import FastAPI, Security

@app.get("/", dependencies=[Security(azure_scheme)])
async def root():
    return {"message": "Hello World"}
```

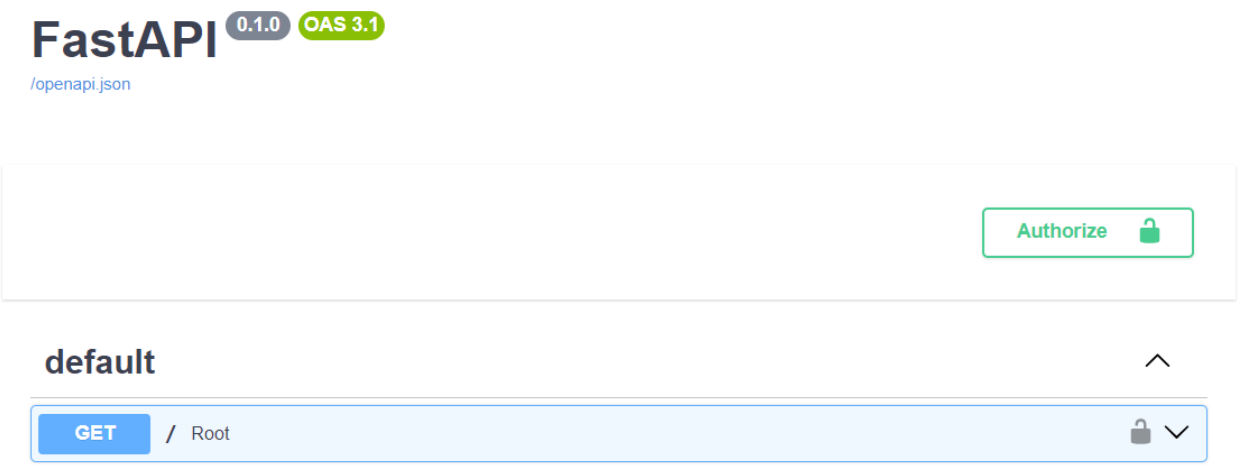
Facultatif : Comme précisé, Security() accepte un paramètre "scopes" (étendues). Si vous le souhaitez, vous pouvez verrouiller votre API pour qu'elle ne soit accessible qu'avec certaines étendues. Par exemple, pour que seules les étendues nommées "read_messages" aient accès à l'API :

```
@app.get("/", dependencies=[Security(azure_scheme, scopes=['
    read_messages'])])
```

Votre API est maintenant sécurisée ! Voyons comment l'utiliser.

2.3 Utilisation

Lancez votre application. Rendez vous sur la documentation OpenAPI : `http://localhost:8000/docs`. Vous verrez un nouveau bouton "Authorize". Avant de cliquer dessus, vous pouvez essayer d'utiliser l'API : ça ne fonctionne pas car vous n'êtes pas authentifié.



Testons l'authentification. Cliquez sur le bouton "Authorize". Cochez la case de l'étendue et laissez le "Client secret" vide. Cliquez à nouveau sur "Authorize" et accordez les permissions demandées.

Une fois authentifié, tentez à nouveau d'utiliser votre API : ça fonctionne !

2.4 S'authentifier sur un conteneur Azure

Si vous tentez de déployer votre API sur un conteneur Azure, vous allez vous heurter à un problème : Azure requiert une connexion HTTPS pour accéder à l'API, or notre application ne fonctionne qu'en HTTP pour le moment.

3 Implémenter le protocole TLS afin d'obtenir une adresse HTTPS

Afin de sécuriser notre API, il faut implémenter le protocole TLS [5].

```
Prérequis :
    * Une API fonctionnant sur http://localhost:8000
    * Azure Active Directory
    * Azure CLI > 2.0.55 ou Azure Cloud Shell
```

3.1 Conteneur "side-car" avec nginx et un certificat auto-signé [6]

Cette méthode permet d'utiliser un certificat auto-signé que l'on "cache" dans un conteneur auxiliaire appelé "side-car". C'est une méthode qui convient pour des tests ou du développement, mais qui peut poser problème sur le long terme, car elle nécessite une intervention manuelle en cas d'expiration du certificat. De plus, nous utiliserons un certificat auto-signé, ce qui n'est pas la meilleure solution ; vous pouvez utiliser un certificat délivré par une autorité de certification de la même manière.

3.1.1 Création du certificat

Nous allons créer un certificat "ssl.crt" et sa clé "ssl.key" avec mkcert [7]. Vous pouvez passer cette partie si vous possédez déjà un certificat valide, ou si vous utilisez une méthode de génération différente.

Sur Windows, vous pouvez installer mkcert avec la commande :

```
choco install mkcert
```

Récupérez l'adresse IP publique de votre conteneur. L'idéal est d'avoir un FQDN ou un DNS qui soit statique. Exécutez les commandes indiquées sur la documentation de mkcert :

```
mkcert -install
mkcert 4.207.117.238 mon-fqdn.azurecontainer.io mon-dns.com
   localhost 127.0.0.1 ::1
```

Remplacez les valeurs d'exemple par les différentes adresses de votre conteneur. Si vous utilisez une IP publique, en cas de changement, il faudra recréer le certificat.

mkcert a créé 2 fichiers en ".pem". Renommez-les "ssl.crt" et "ssl.key" (pour celui qui se termine en "-key.pem") pour plus de clarté.

3.1.2 Mise en place du conteneur auxiliaire [6]

Nous allons créer un groupe de conteneurs avec un conteneur d'application et un conteneur side-car exécutant un fournisseur SSL/TLS.

```
Prérequis :
    * Une image dans un groupe de ressources sur azure (ou une image
        préexistante, comme aci-helloworld)
```

Vous configurerez un groupe de conteneurs composé de deux conteneurs :

— Un conteneur d'applications qui exécute notre API

— Un conteneur side-car exécutant l'image Nginx publique, configuré pour utiliser TLS

Dans cet exemple, le groupe de conteneurs expose uniquement le port 443 pour Nginx avec son adresse IP publique. Nginx achemine les requêtes HTTPS vers l'application web, qui écoute en interne sur le port 80. Vous pouvez adapter l'exemple pour les applications de conteneur qui écoutent sur les autres ports.

Créer un fichier de configuration Nginx : Dans cette section, vous allez créer un fichier de configuration afin que Nginx utilise TLS. Commencez par copier le texte suivant dans un nouveau fichier nommé "nginx.conf". Dans "location", veillez à définir "proxy_pass" avec le port approprié pour votre application. Dans cet exemple, nous avons défini le port 80 pour le conteneur.

```
# nginx Configuration File
# https://wiki.nginx.org/Configuration

# Run as a less privileged user for security reasons.
user nginx;

worker_processes auto;

events {
  worker_connections 1024;
}

pid         /var/run/nginx.pid;

http {

    #Redirect to https, using 307 instead of 301 to preserve post
       data

    server {
        listen [::]:443 ssl;
        listen 443 ssl;

        server_name localhost;

        # Protect against the BEAST attack by not using SSLv3 at all
           . If you need to support older browsers (IE6) you may
           need to add
        # SSLv3 to the list of protocols below.
        ssl_protocols               TLSv1.2;

        # Ciphers set to best allow protection from Beast, while
           providing forwarding secrecy, as defined by Mozilla -
           https://wiki.mozilla.org/Security/Server_Side_TLS#Nginx
        ssl_ciphers                 ECDHE-RSA-AES128-GCM-SHA256:ECDHE
           -ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:
```

```
    ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:
    DHE-DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-
    SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:
    ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-
    ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-
    AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-
    DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-
    SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-
    SHA384:ECDHE-RSA-RC4-SHA:ECDHE-ECDSA-RC4-SHA:AES128:
    AES256:RC4-SHA:HIGH:!aNULL:!eNULL:!EXPORT:!DES:!3DES:!MD5
    :!PSK;
ssl_prefer_server_ciphers  on;

# Optimize TLS/SSL by caching session parameters for 10
   minutes. This cuts down on the number of expensive TLS/
   SSL handshakes.
# The handshake is the most CPU-intensive operation, and by
   default it is re-negotiated on every new/parallel
   connection.
# By enabling a cache (of type "shared between all Nginx
   workers"), we tell the client to re-use the already
   negotiated state.
# Further optimization can be achieved by raising
   keepalive_timeout, but that shouldn't be done unless you
   serve primarily HTTPS.
ssl_session_cache    shared:SSL:10m; # a 1mb cache can hold
   about 4000 sessions, so we can hold 40000 sessions
ssl_session_timeout  24h;


# Use a higher keepalive timeout to reduce the need for
   repeated handshakes
keepalive_timeout 300; # up from 75 secs default

# remember the certificate for a year and automatically
   connect to HTTPS
add_header Strict-Transport-Security 'max-age=31536000;
   includeSubDomains';

ssl_certificate      /etc/nginx/ssl.crt;
ssl_certificate_key  /etc/nginx/ssl.key;

location / {
    proxy_pass http://localhost:80; # TODO: replace port if
       app listens on port other than 80

    proxy_set_header Connection "";
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $remote_addr;
```

```
        }
    }
}
```

Encoder les secrets et le fichier de configuration au format Base64 : Encodez le fichier de configuration Nginx, le certificat TLS/SSL et la clé TLS au format Base64.

```
cat nginx.conf | base64 > base64-nginx.conf
cat ssl.crt | base64 > base64-ssl.crt
cat ssl.key | base64 > base64-ssl.key
```

Créer un fichier YAML : Copiez le fichier YAML suivant dans un nouveau fichier nommé "deploy-aci.yaml". Entrez le contenu des fichiers encodés au format Base64 à l'endroit indiqué sous "secret". Au cours du déploiement, ces fichiers sont ajoutés à un volume secret dans le groupe de conteneurs. Dans cet exemple, le volume secret est monté sur le conteneur Nginx.

```
api-version: 2019-12-01
location: westus
name: app-with-ssl
properties:
  containers:
  - name: nginx-with-ssl
    properties:
      image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
      ports:
      - port: 443
        protocol: TCP
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
      volumeMounts:
      - name: nginx-config
        mountPath: /etc/nginx
  - name: my-app
    properties:
      image: mcr.microsoft.com/azuredocs/aci-helloworld
      ports:
      - port: 80
        protocol: TCP
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
  volumes:
  - secret:
      ssl.crt: <Enter contents of base64-ssl.crt here>
```

```
      ssl.key: <Enter contents of base64-ssl.key here>
      nginx.conf: <Enter contents of base64-nginx.conf here>
    name: nginx-config
  ipAddress:
    ports:
    - port: 443
      protocol: TCP
    type: Public
  osType: Linux
tags: null
type: Microsoft.ContainerInstance/containerGroups
```

Déployer le groupe de conteneurs : Sur Azure CLI ou Azure Cloud Shell, créez un groupe de ressources avec la commande az group create :

```
az group create --name monGroupeDeRessources --location europe
```

Déployez le groupe de conteneurs avec la commande az container create, en transmettant le fichier YAML en tant qu'argument.

```
az container create --resource-group <myResourceGroup> --file deploy
  -aci.yaml
```

3.2 Modifier l'application FastAPI pour utiliser du HTTPS

Pour pouvoir utiliser le HTTPS, ou même accepter les requêtes venant du conteneur, il faut ajouter les adresses voulues aux origines CORS. Précédemment, dans notre fichier main.py, dans la classe Settings, nous avons écrit :

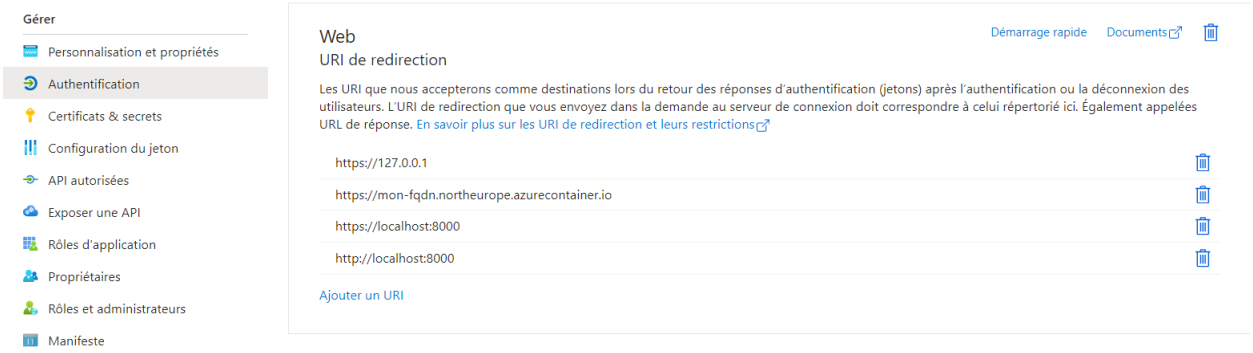
```
BACKEND_CORS_ORIGINS: list[str | AnyHttpUrl] = [
        'http://localhost:8000',
        '127.0.0.1'
        ]
```

A cette liste, il faut ajouter l'adresse https ://localhost :8000, et éventuellement https ://127.0.0.1.

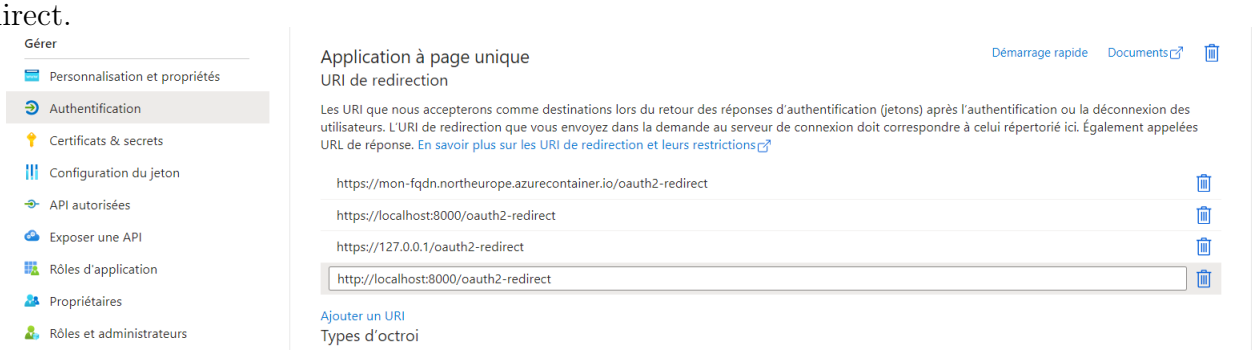
```
BACKEND_CORS_ORIGINS: list[str | AnyHttpUrl] = [
        'http://localhost:8000', 'https://localhost:8000',
        'http://127.0.0.1', 'https://127.0.0.1'
        ]
```

3.3 Modifier l'application sur Azure pour utiliser du HTTPS

Sur Azure, rendez vous sur la page de votre application [3] -> Authentification -> URI de redirection. Indiquez les adresses https que vous souhaitez utiliser (adresse IP publique, FQDN, DNS...) en version http et https.



Faites de même sur l'application OpenAPI, en rajoutant à chaque adresse le suffixe /oauth2-redirect.



Votre application est désormais fonctionnelle et utilisable en HTTPS : il ne reste qu'à mettre à jour l'image et le conteneur !

4 Sources

[1] Documentation fastapi-azure-auth:https://intility.github.io/fastapi-azure-auth/single-tenant/azure_setup/

[2] Documentation fastapi-azure-auth :https://intility.github.io/fastapi-azure-auth/single-tenant/fastapi_configuration/

[3] Applications inscrites sur Azure :https://portal.azure.com/#blade/Microsoft_AAD_IAM/ActiveDirectoryMenuBlade/RegisteredApps

[4] Configuration OpenAPI :https://swagger.io/docs/open-source-tools/swagger-ui/usage/oauth2/

[5] BD explicative sur les protocoles TLS, SSL, le HTTP et HTTPS :https://howhttps.works/why-do-we-need-https/

[6] Activer un point de terminaison TLS dans un conteneur side-car :https://learn.microsoft.com/fr-fr/azure/container-instances/container-instances-container-group-ssl

[7] mkcert pour la génération de certificats auto-signés :https://github.com/FiloSottile/mkcert