Formación Azure B2C

Integración con FastAPI

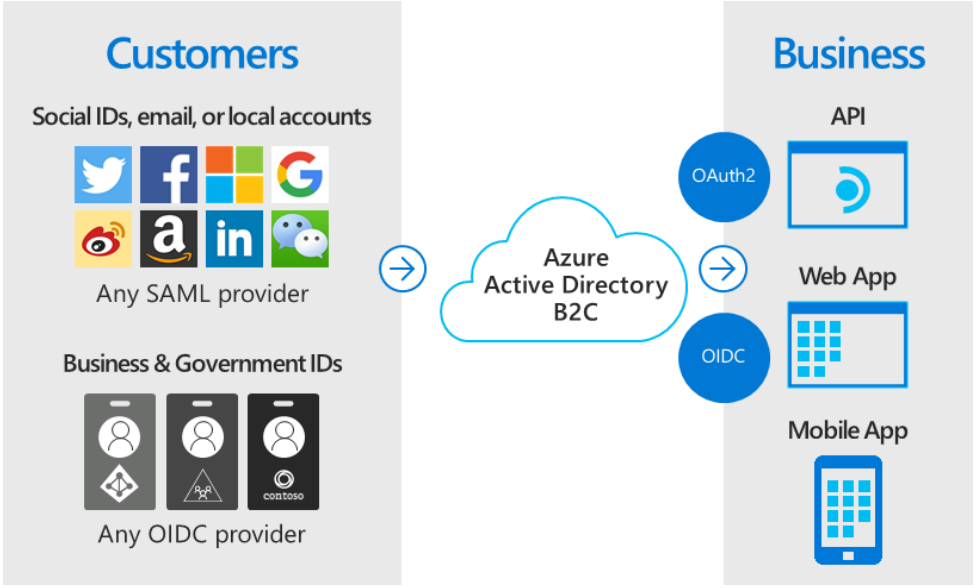
30/04/2024

ÍNDICE

- Conociendo B2C y sus ventajas
- Configuración de Azure
- Integración con FastAPI en Python

1. Conociendo B2C y sus ventajas

Conociendo B2C



- B2C – Bussiness to consumer. Pensado para usuarios finales y sus aplicaciones.

¿Qué tenemos ahora y cómo nos afectaría?

- Tenemos autenticación OAuth2 mediante token con campos extraídos de una tabla usuarios en BD. Esto lo sustituye B2C: es su propio proveedor de tokens.
- En nuestras API usamos API Key propia como capa adicional de seguridad. Se podría mantener.

Ventajas

- Inyección de características: permite **definición personalizada de atributos de usuario, luego accesibles en el token.**
- **Gestión más cómoda** de todos los aspectos desde el portal de Azure. Altas/bajas, permisos...
- Métricas y pago por uso. Disponemos de todas las estadísticas esperables de Azure. Pago por MAU: **50.000 usuarios/mes gratis para P1/P2**

Ventajas

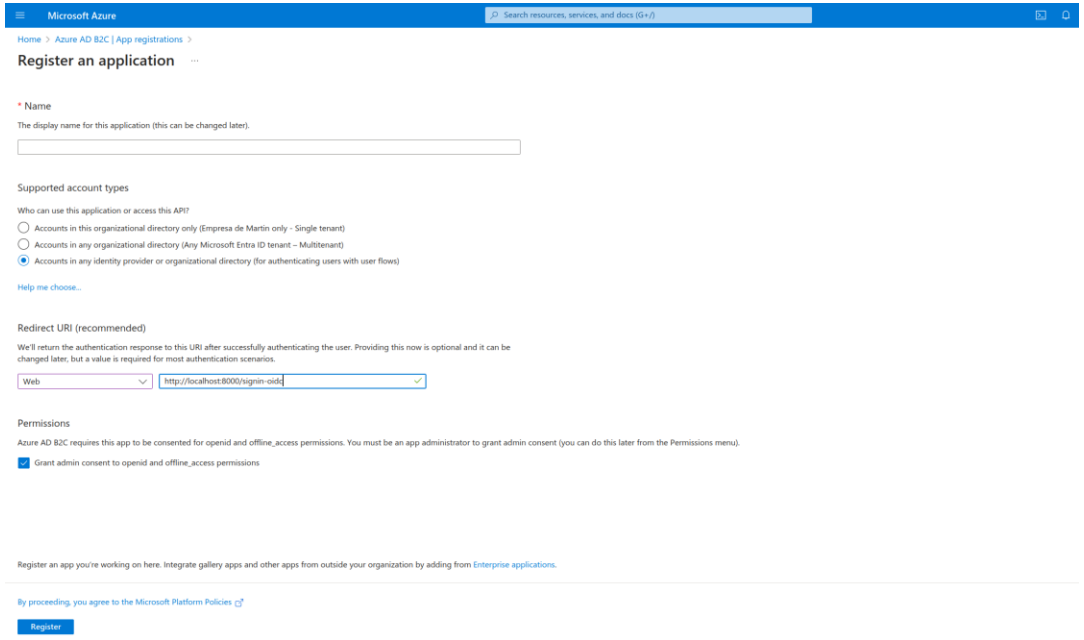
- **Continuidad**: Azure B2C mantiene nuestro esquema de seguridad, con token firmado por ellos: cambios mínimos. Podemos mantener los mismos Scopes.
- **Personalización**: Con los flujos de autenticación, podemos personalizar la experiencia de usuario, con la imagen corporativa que deseemos, permitir que un usuario se registre en el directorio activo, que solo pueda logearse, etc.
- **Seguridad avanzada**: Soporte MFA por email, Autenticator, SMS...adhesión a todos los estándares de seguridad. No conlleva trabajo adicional de nuestra parte, asegurados nuestros endpoints, es "gratis".

Desventajas

- **Bases de datos**. Idealmente, habría que adaptar las BD en lo ya existente. Ya no vamos a tener tablas de usuario que podamos usar como clave foránea en otras. Esto se puede paliar, al menos parcialmente, con atributos en el token que hagan match con columnas de las tablas.
- **Prosigue el abrazo interminable**: sería otro elemento de la infraestructura que pasa a depender de Microsoft y de Azure. Se diría que, sin embargo, nuestras experiencias al respecto son positivas...

2. Configuración de Azure

2. Configuración de Azure



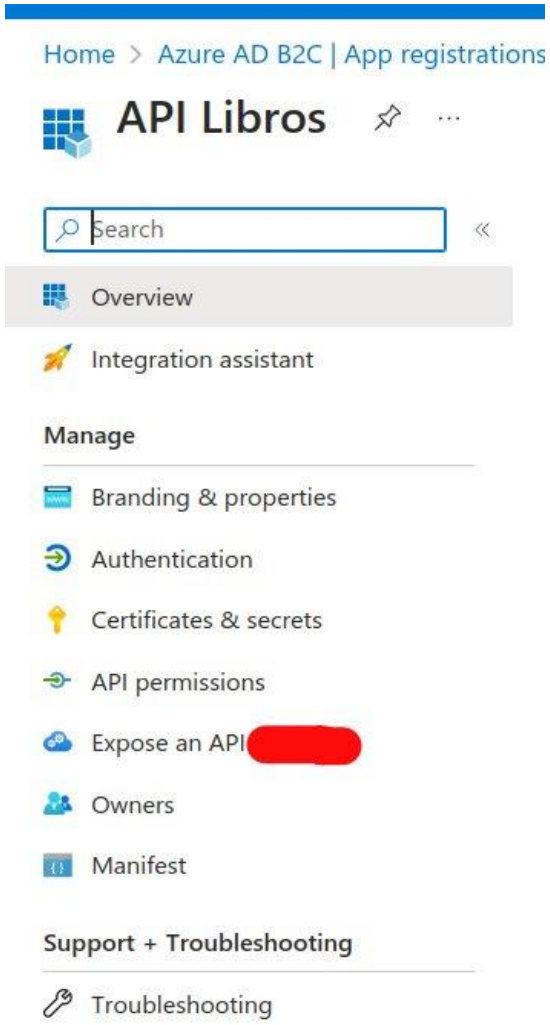
-**Registramos dos aplicaciones** en este menú. Una será web, con redirección http://localhost:8000/signin-oidc, y la otra será SPA (single-page app), redirección http://localhost:8000/oauth2-redirect

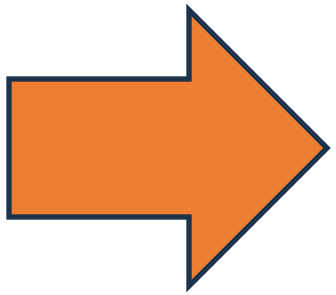
-Por convención la primera se llamará NombreApp y la segunda NombreApp - OpenAPI

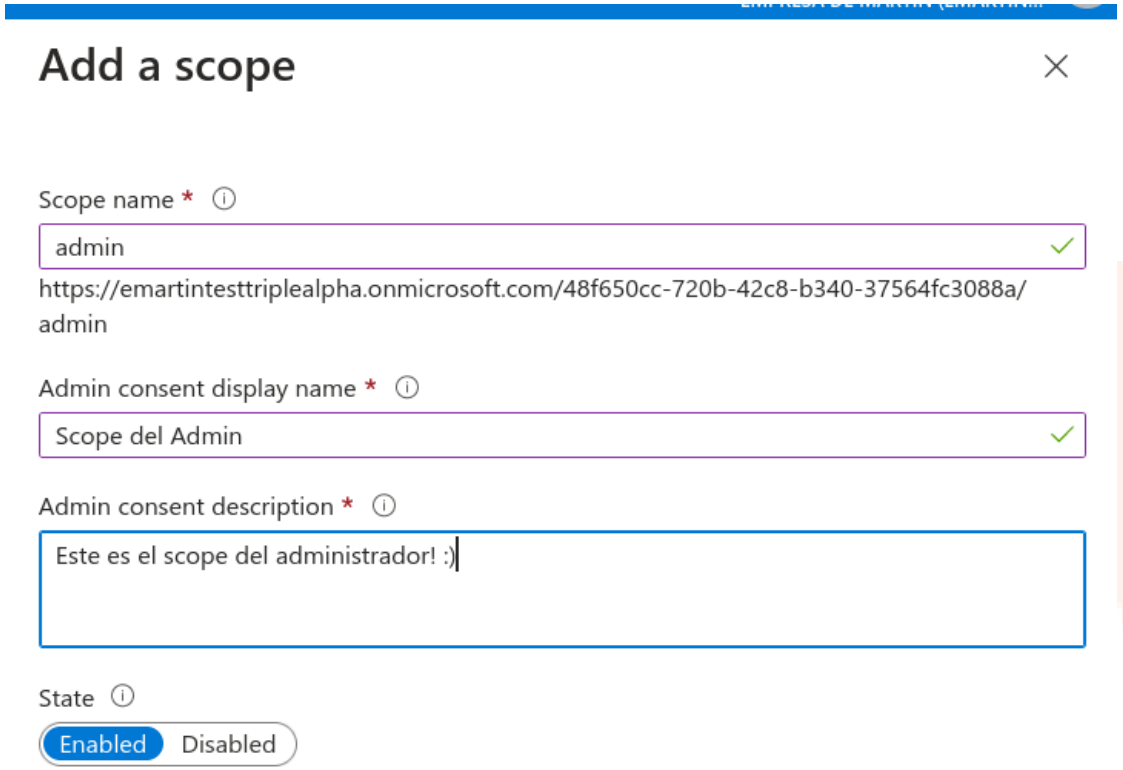
2. Configuración de Azure

- Registradas las aplicaciones, en la vista general veremos el App ID, que anotaremos. Se utilizarán en el archivo .env/Secretos de GitHub. La primera será APP_CLIENT_ID y la segunda OPENAPI_CLIENT_ID.
- A continuación, en la primera aplicación, vamos a Expose API.
- Ya ahí nos sale Expose a Scope. Ahí aceptaremos la URL por defecto que Azure nos crea, y añadiremos al final el Scope, por ejemplo /adm**in**

2. Configuración de Azure





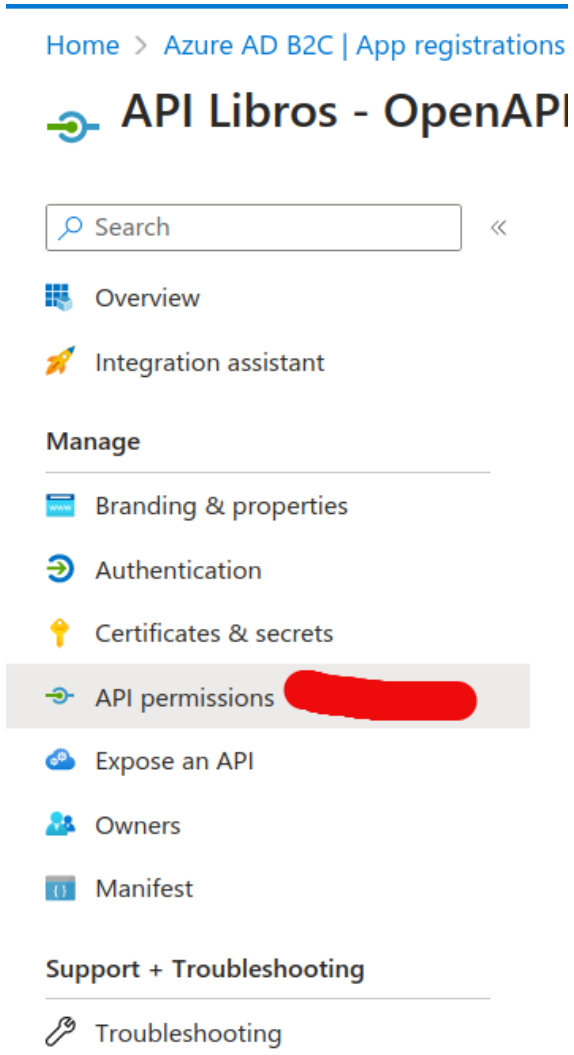


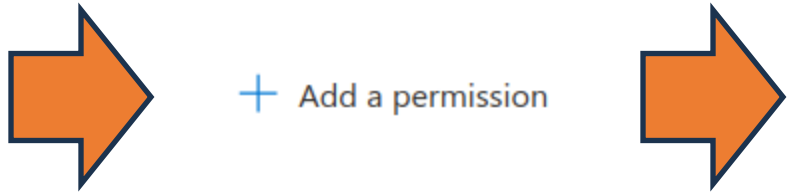
Si es la primera vez, nos sugerirá una URL con una larga cadena de números. La aceptamos.

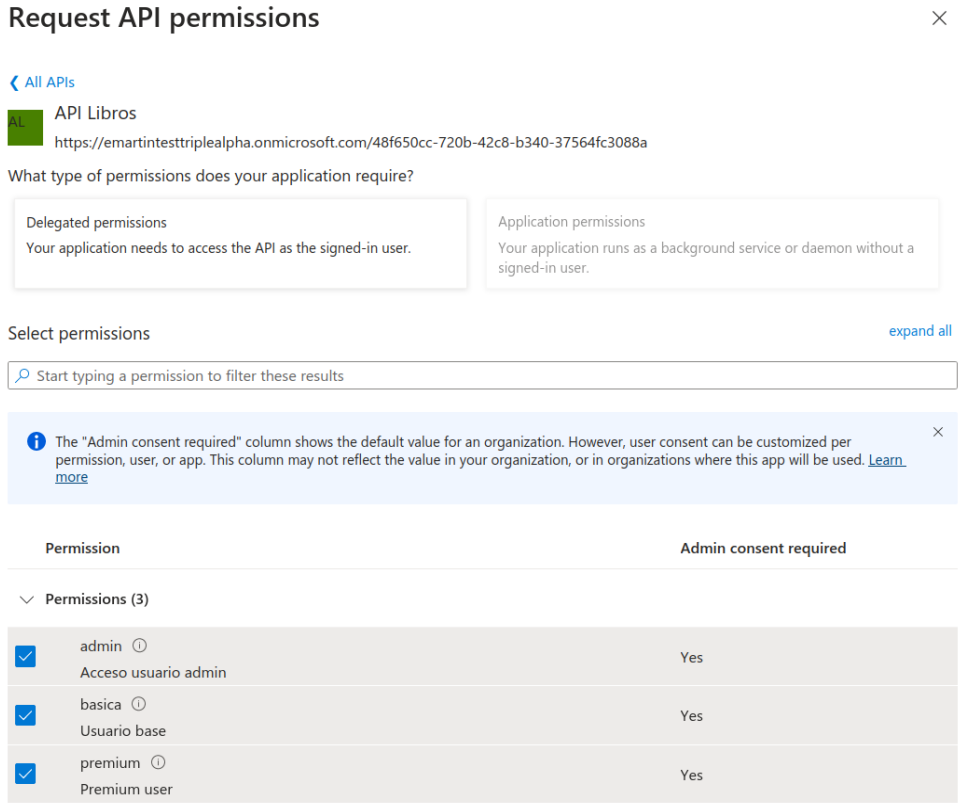
2. Configuración de Azure

- En la segunda App, vamos a API Permissions.
- Ahí nos salen todas las aplicaciones que haya creadas como API en el Registro. En este caso, sería solo la primera que acabamos de crear.
- Ya salen los Scopes que hemos creado. Los marcamos y les damos permiso. Con esto, OpenAPI (es decir, nuestra FastAPI), cuando solicite un token a Azure, va a llevar uno de los Scopes que se han autorizado.

2. Configuración de Azure

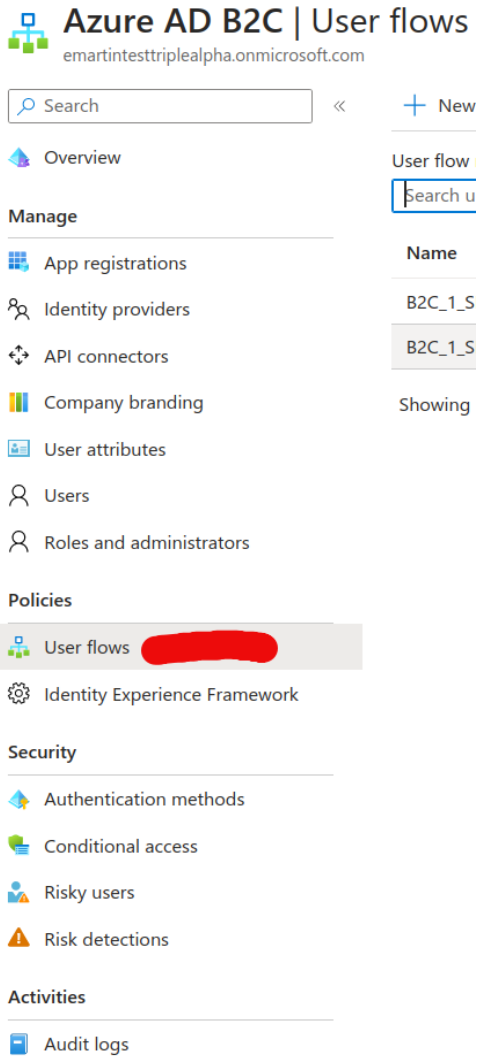






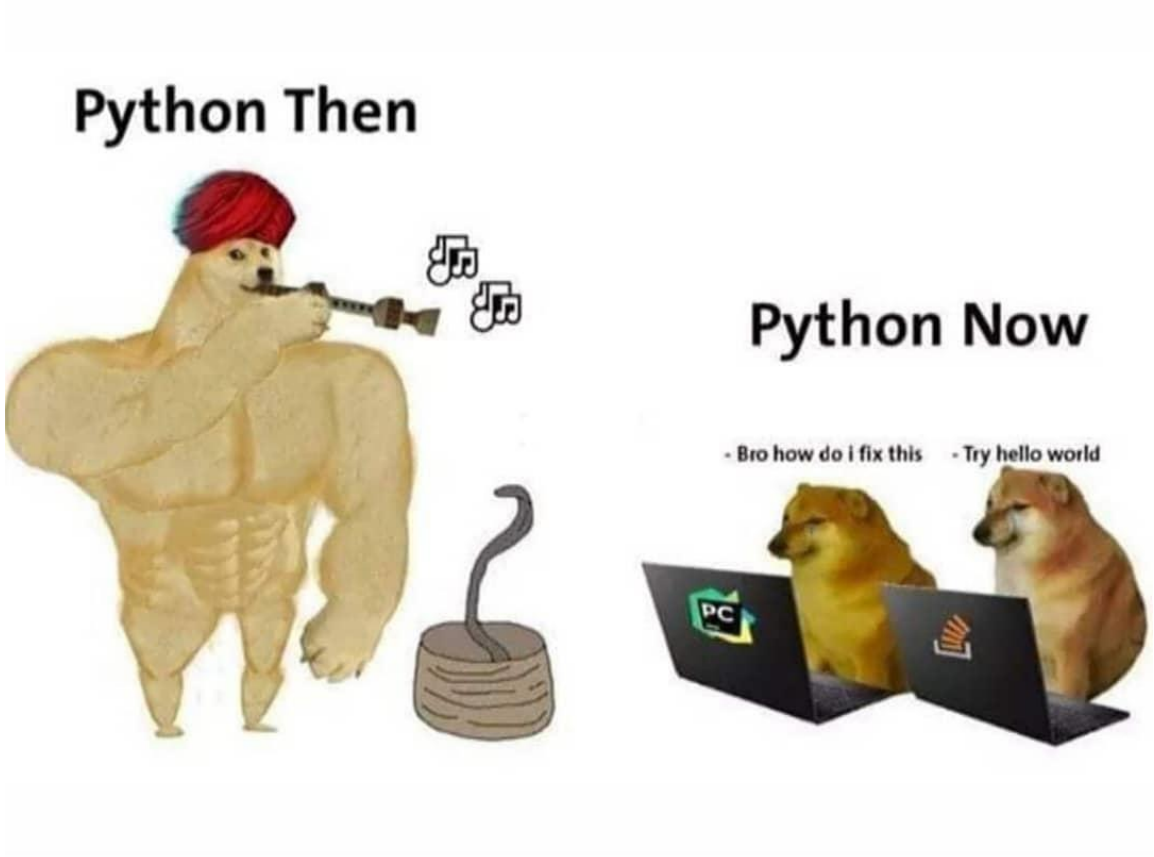
Veremos que los permisos coinciden con los Scopes declarados.

2. Creación de flujos



- Ahí se crean los flujos de usuario, lo que va a ver al iniciar sesión.
- Es tan simple como pinchar ahí y escoger el tipo, Signup/Signin, solo Signin...**y anotar el nombre, pues lo usaremos en el .env. Todos empiezan por B2C.**
- Podemos personalizar la marca corporativa desde las opciones del flujo, una vez creado. Admite imágenes, CSS y HTML personalizados...
- Es posible crear flujos totalmente personalizados, sin usar las plantillas que da Azure.

3. Implementación en Python con fastapi-azure-auth



3. Implementación en Python con fastapi-azure-auth

```
APP_CLIENT_ID=ID primera app
OPENAPI_CLIENT_ID=ID segunda app
TENANT_NAME=Dominio de azure
AUTH_POLICY_NAME=Nombre del flujo
```

Variables env necesarias.

```python
class AzureSettings(BaseSettings):

    TENANT_NAME: str = ""
    APP_CLIENT_ID: str = ""
    OPENAPI_CLIENT_ID: str = ""
    AUTH_POLICY_NAME: str = ""

    @computed_field
    @property
    def SCOPE_NAME(self) -> str:
        return f'https://{self.TENANT_NAME}.onmicrosoft.com/{self.APP_CLIENT_ID}'

    @computed_field
    @property
    def SCOPES(self) -> dict:
        return {
            f'{self.SCOPE_NAME}/admin': "Acceso a todos los endpoint", f'{self.SCOPE_NAME}/premium': "Acceso a los endpoint premium",     f'{self.SCOPE_NAME}/basi
        }

    @computed_field
    @property
    def OPENID_CONFIG_URL(self) -> dict:
        return f'https://{self.TENANT_NAME}.b2clogin.com/{self.TENANT_NAME}.onmicrosoft.com/{self.AUTH_POLICY_NAME}/v2.0/.well-known/openid-configuration'

    @computed_field
    @property
    def OPENAPI_AUTHORIZATION_URL(self) -> dict:
        return f'https://{azure_settings.TENANT_NAME}.b2clogin.com/{azure_settings.TENANT_NAME}.onmicrosoft.com/{azure_settings.AUTH_POLICY_NAME}/oauth2/v2.0/autho

    @computed_field
    @property
    def OPENAPI_TOKEN_URL(self) -> dict:
        return f'https://{azure_settings.TENANT_NAME}.b2clogin.com/{azure_settings.TENANT_NAME}.onmicrosoft.com/{azure_settings.AUTH_POLICY_NAME}/oauth2/v2.0/token

    model_config = SettingsConfigDict(
        extra='allow',
        env_file='.env',
        env_file_encoding='utf-8',
        case_sensitive=True
    )

azure_settings = AzureSettings()
```

Creamos una clase que lee los .env para construir las rutas que se precisan.

Los Scopes que va a ver el usuario de Swagger/FastAPI se crean como diccionario en el campo SCOPES. Es la única forma válida de pasarlos.

3. Implementación en Python con fastapi-azure-auth

```python
azure_scheme = B2CMultiTenantAuthorizationCodeBearer(
    app_client_id=azure_settings.APP_CLIENT_ID,
    openid_config_url=azure_settings.OPENID_CONFIG_URL,
    openapi_authorization_url=azure_settings.OPENAPI_AUTHORIZATION_URL,
    openapi_token_url=azure_settings.OPENAPI_TOKEN_URL,
    scopes=azure_settings.SCOPES,
    validate_iss=False,
)
```

Se crea un esquema de configuración. Es lo que realmente luego va a pasar los parámetros al módulo Security de FastAPI. B2CMultiTenant es estático, de manera que solo podemos tener un scheme.

```python
app = FastAPI(
    title="Pruebas y aprendizaje. API",
    description=description,
    contact={'email': 'jkniffki@triplealpha.in'},
    swagger_ui_oauth2_redirect_url='/oauth2-redirect',
    swagger_ui_init_oauth={
        'usePkceWithAuthorizationCodeGrant': True,
        'clientId': azure_settings.OPENAPI_CLIENT_ID,
    }
)
```

Las dos últimas líneas de Swagger son necesarias para para nuestro app en el fichero main.

3. Implementación en Python con fastapi-azure-auth

```python
router = APIRouter(
    tags=["Libros"],
    dependencies=[Security(azure_scheme, scopes=['premium'])],
    default_response_class=ORJSONResponse
)
```

Aseguramos nuestros rutas de la API con el módulo Security de FastAPI y el esquema, y con scopes le indicamos qué debe haber en el token.

```python
async def get_user_from_azure_token(token: Annotated[str, Security(azure_scheme)]):
    return {"scopes":token.scp, "username":token.name}
```

Con esta función se captura el token. Podemos extraer la información que queramos de él: en este caso uso username para buscar los libros que tiene un usuario en la BD y cuyo nombre coincida con el almacenado en el token. Así se podría enlazar Azure B2C con nuestra BD cuando no sea posible modificar esta última.