Tabla de contenido

Qué es JWT y para que sirve?

JWT (JSON Web Token) es un estándar abierto que define una forma compacta y autónoma de transmitir información de forma segura entre dos partes como un objeto JSON. Se utiliza comúnmente para implementar autenticación y autorización en aplicaciones web y APIs.
Algunas características y ventajas de JWT:

- Compacto: Debido a que está en formato JSON puede ser enviado a través de una URL, un POST parameter o dentro de un header HTTP. Más compacto que un token basado en XML.
- Autónomo (Self-contained): Contiene toda la información necesaria sobre el usuario autenticado, evitando consultas extras a la base de datos.
- Seguro: Puede ser firmado digitalmente (con HMAC, RSA o ECDSA) para garantizar que no ha sido modificado.
- Portable: Los tokens JWT son independientes del proveedor, por lo que se pueden usar entre múltiples hosts.
- Escalable: Permite a frontend y backend escalar de forma independiente.
- Decodificable: Los JWTs son completamente decodificables, lo que ayuda a depurar problemas.

Anatomía de un JWT:

- Header: Metadatos del algoritmo usado para firmar el token (tipicamente HS256 o RS256).
- Payload: Datos del token, como usuario, tiempo de expiración, etc.
- Firma: Firma digital de los datos del token para verificar su integridad.

JWT es útil para implementar autenticación sin estado (stateless) ya que el servidor no necesita almacenar un estado de sesión por cada usuario. Simplemente verifica la firma y payload del JWT para autenticar las peticiones entrantes.
JWT permite intercambiar de forma segura información de autenticación y autorización implementando un estándar abierto e independiente de la plataforma.

Diagrama de la solución:

```
app.py
└─> app_factory.py

app_factory.py
├─> config.py
├─> models.py
├─> routes.py
├─> api.py
└─> gestores.py

routes.py
├─> auth.py
│    ├─> login()
│    └─> login_jwt()
├─> personas.py
│    ├─> index()
│    │    └─> gestor_personas.obtener_pagina()
│    ├─> create()
│    │    └─> gestor_personas.crear()
│    ├─> update()
│    │    └─> gestor_personas.editar()
│    └─> delete()
│         └─> gestor_personas.eliminar()
└─> macros.py
     ├─> paginado()
     └─> modal()

models.py
└─> Persona
    ├─> guardar()
    └─> borrar()

gestores.py
├─> gestor_personas.py
│    ├─> obtener_pagina()
│    ├─> obtener()
│    ├─> crear()
│    ├─> editar()
│    └─> eliminar()
├─> gestor_lugares.py
│    └─> consultar()
├─> gestor_generos.py
│    └─> obtener_todos()
└─> gestor_comun.py

api.py
├─> PersonasResource
│    └─> gestor_personas
```

```
└─> LugaresResource
    └─> gestor_lugares
```

Implementación:

Para implementar JWT a la solución actual es necesario la instalación del modulo Flask-JWT-Extended

```
pip install Flask-JWT-Extended
```

Módulo app_factory.py:

Configurar la extensión Flask-JWT-Extended y crear en el modulo auth la instancia para el CSRF e instancialo en la creación de la app, con el fin de poder excluir algunos métodos post de dicha protección. Por lo que, el modulo routes_personas.py ahora importará el objeto csrf delarado en auth.py e inicializado aquí.

...

```
from flask_jwt_extended import JWTManager
from modules.auth import csrf

...

def create_app():
    csrf.init_app(app)
    jwt = JWTManager(app)
```

Módulo auth.py:

Este modulo concentra la autenticación de los usuarios, por lo que se agrega un nuevo enpoint para solicitor el token mediant eel método POST, enviando como request en formato json username y password con los valores correspondientes a un usuario válido, por ejemplo:

```
{
    "username":"user1",
    "password": "pass1"
}

@auth_bp.route('/login-jwt', methods=['POST'])
@csrf.exempt
def login_jwt():
    username = request.json.get('username', None)
    password = request.json.get('password', None)
```

```
    user = User.query.filter_by(username=username).first()
    if user and user.check_password(password):
        access_token = create_access_token(identity=username)
        return jsonify(access_token=access_token)
    else:
        return jsonify({"msg":"Credenciales inválidas"}),401
```

El decorador @csrf.exempt indica que este endpoint está excluye el uso de la protección CSRF

Se obtiene del método POST los valores enviados para username y password y se utiliza la misma lógica que el login estándar declarado mas arriba.

Si el usuario y password son válidos, se creará un Access token, el cual formará parte del response (respuesta) del endpoint, permitiendo de esta manera poder ser pasado como parámetro en las demás llamadas protegidas por el token (API lugares.py y personas.py)

De no ser válido el usuario y/o password se retornará un código http 401 y el mensaje indicando que las credenciales no son válidas.

Ejemplo del response:


```
{
    "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTY5MjczMzY3Myw
ianRpIjoiZTU1MzczZmMtNmM0ZC00ZDAzLTgxMjQtNDNkNWQ2OTIwNzM5IiwidHlwZSI6ImFjY2VzcyIs
InN1YiI6InVzZXIxIiwibmJmIjoxNjkyNzMzNjczLCJleHAiOjE2OTI3MzQ1NzN9.OW9ESfcOFjXQmXtF
641cS1GxI6IXZ48M9Rbaf2gUI_g"
}
```


Módulos API personas.py y lugares.py

Para utilizar el usuario autenticado por JWT en las APIs de lugares.py y personas.py, se puede hacer lo siguiente:

Proteger los recursos API con @jwt_required para forzar autenticación JWT:

```
from flask_jwt_extended import jwt_required, get_jwt_identity

class PersonasResource(Resource):

    @jwt_required()
    def get(self, persona_id=None):
```

…

El decorador `@jwt_required()` se implementa en cada método declarado (get, post, delete, put)

Si la lógica del negocio lo requiere, se puede obtener la identidad del usuario autenticado con get_jwt_identity()

```
from flask_jwt_extended import jwt_required, get_jwt_identity

class PersonasResource(Resource):

  @jwt_required()
  def get(self):
    # obtener usuario desde JWT
    current_user = get_jwt_identity()
    ...
```

De esta forma se consigue asociar las llamadas a las APIs protegidas con JWT al usuario autenticado actualmente.

Se mantiene la lógica de negocio desacoplada en los gestores y se pasa el usuario para aplicar lógica específica.

Tiempo de vida del Token / expiración

Existen varias opciones para configurar el tiempo de vida o expiración de los tokens JWT en Flask-JWT-Extended:

1. Al crear el token, se puede especificar el tiempo de expiración en segundos:

   ```
   access_token = create_access_token(identity=username,
   expires_delta=timedelta(minutes=30))
   ```

   Esto creará un token que expira en 30 minutos.

2. Mediante la configuración de la aplicación Flask:

   ```
   app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1)
   ```

   Establece el tiempo de expiración default en 1 hora para todos los tokens.

3. Usando los decoradores jwt_optional y fresh_jwt_required, se puede requerir tokens frescos:

   ```
   @jwt_optional
   @fresh_jwt_required
   def protected():
     # solo ejecutará si el token no ha expirado
   ```

4. Al configurar JWTManager, establecer token_in_blacklist_callback para revocar tokens expirados:

```
jwt = JWTManager(app)

@jwt.token_in_blacklist_loader
def check_if_token_revoked(decrypted_token):
  # consultar si token está en lista negra
  return revoked_tokens_set.is_member(decrypted_token['jti'])
```

5. Rotar periódicamente la firma secreta de la aplicación con JWT_SECRET_KEY para forzar expiración de tokens antiguos.

Existen múltiples opciones para controlar la expiración y rotación de tokens JWT según los requerimientos de seguridad de la aplicación.