Gestión de Sesiones con JWT y Control de Acceso RBAC

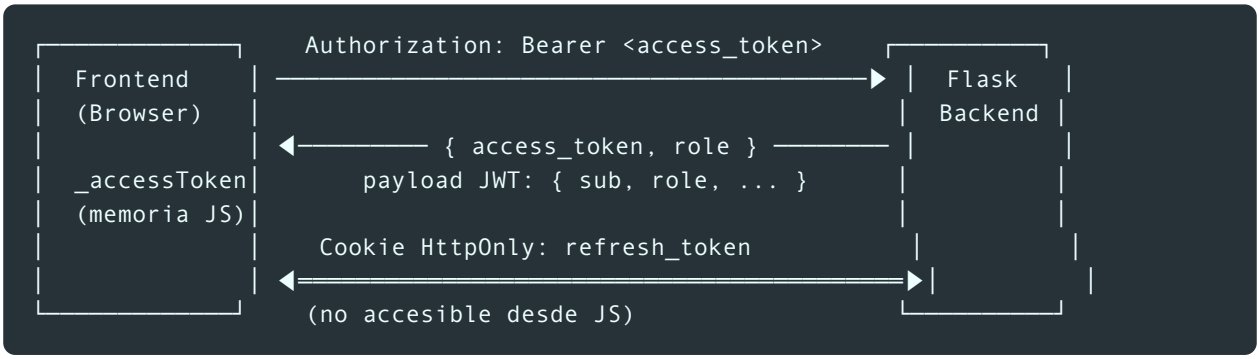
Repositorio: github.com/alejandroquinonesgamez/medical_register
Rama: `dev`
Commit JWT: `0e7cf4f` — *feat(auth): integración JWT con access token en memoria y refresh token HttpOnly*
Commit RBAC: `b29605c` — *feat(auth): RBAC con roles admin/user, decorador require_role y documentación completa*

---

1. Descripción de la solución implementada

La aplicación médica implementa un sistema de autenticación stateless basado en **JWT (JSON Web Tokens)** con control de acceso basado en roles (**RBAC**), sustituyendo el mecanismo anterior de sesiones Flask + CSRF.

Esquema de doble token



Token	Duración	Almacenamiento	Transporte
Access token	15 min	Memoria JavaScript (`_accessToken`)	Cabecera `Authorization: Bearer`
Refresh token	7 días	Cookie `HttpOnly` (inaccesible desde JS)	Cookie automática del navegador

Decisiones de seguridad:

- **Access token solo en memoria JS**: no se guarda en `localStorage` → mitiga XSS.
- **Refresh token como cookie HttpOnly**: JavaScript no puede leerlo → protegido contra XSS.
- **CSRF eliminado**: la cabecera `Authorization: Bearer` no se envía automáticamente en peticiones cross-origin → CSRF no aplica.

- **Token blacklist**: al hacer logout, el `jti` del refresh token se añade a una blacklist en BD → revocación real.
- **Rol embebido en el JWT**: el claim `role` viaja en el access token → sin consultas extra a BD para verificar permisos.

Sistema de roles (RBAC)

Rol	Asignación	Permisos
admin	Primer usuario registrado (automático). Puede promover otros usuarios	Acceso total: datos propios + administración (DefectDojo, WSTG, gestión de roles)
user	Todos los usuarios registrados después del primero	Solo sus propios datos: perfil, peso, IMC, estadísticas

---

2. Gestión de la sesión y control de acceso

2.1 Generación del JWT (backend)

Al hacer login o registro, el servidor genera dos tokens. El **access token** incluye el claim `role` directamente en el payload:

```
# app/jwt_utils.py
import secrets
from datetime import datetime, timezone
import jwt
from .config import JWT_CONFIG

def create_access_token(user_id, username, role="user"):
    now = datetime.now(timezone.utc)
    payload = {
        "sub": str(user_id),       # Identificador del usuario
        "username": username,
        "role": role,               # ← Rol para RBAC ("admin" o "user")
        "type": "access",
        "jti": secrets.token_urlsafe(16),  # ID único para revocación
        "iat": now,
        "exp": now + JWT_CONFIG["access_token_expires"],  # 15 min
    }
    return jwt.encode(payload, _get_secret(), algorithm=JWT_CONFIG["algorithm"])
```

El **refresh token** (larga vida) se establece como cookie HttpOnly:

```
# app/jwt_utils.py
def create_refresh_token(user_id):
    now = datetime.now(timezone.utc)
    payload = {
        "sub": str(user_id),
        "type": "refresh",
        "jti": secrets.token_urlsafe(16),
        "iat": now,
        "exp": now + JWT_CONFIG["refresh_token_expires"],  # 7 días
    }
    return jwt.encode(payload, _get_secret(), algorithm=JWT_CONFIG["algorithm"])
```

Configuración centralizada:

```
# app/config.py
JWT_CONFIG = {
    "secret_key": os.environ.get("JWT_SECRET_KEY", ""),
    "algorithm": "HS256",
    "access_token_expires": timedelta(minutes=15),
    "refresh_token_expires": timedelta(days=7),
    "refresh_cookie_name": "refresh_token",
    "refresh_cookie_path": "/api/auth",
}
```

2.2 Validación del JWT (middleware `require_auth`)

Cada petición protegida pasa por el decorador `require_auth`, que extrae el token de la cabecera `Authorization: Bearer`, verifica su firma, expiración, tipo y blacklist, y almacena el rol en el contexto de Flask:

```
# app/routes.py
def require_auth(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        token = _get_bearer_token()
        if not token:
            return jsonify({"error": "Autenticación requerida"}), 401
        try:
            payload = decode_token(token, expected_type="access")
        except pyjwt.ExpiredSignatureError:
            return jsonify({"error": "Token expirado"}), 401
        except (pyjwt.InvalidTokenError, ValueError):
            return jsonify({"error": "Autenticación requerida"}), 401

        jti = payload.get("jti")
        if jti and current_app.storage.is_token_blacklisted(jti):
            return jsonify({"error": "Autenticación requerida"}), 401

        g.current_user_id = int(payload["sub"])
        g.current_user_role = payload.get("role", "user")  # ← Rol del JWT
        g.jwt_payload = payload
        return func(*args, **kwargs)
    return wrapper
```

2.3 Control de acceso por rol (middleware `require_role`)

El decorador `require_role` se aplica **después** de `require_auth` y verifica que el rol del usuario esté en la lista de roles permitidos:

```
# app/routes.py
def require_role(*allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            user_role = getattr(g, "current_user_role", None)
            if user_role not in allowed_roles:
                return jsonify({"error": "No tienes permisos para realizar esta
acción"}), 403
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

Ejemplo de uso — ruta protegida solo para administradores:

```
@api.route('/admin/users/<int:target_user_id>/role', methods=['PUT'])
@require_auth                  # 1. Verifica JWT válido
@require_role("admin")         # 2. Verifica que role == "admin"
def update_user_role(target_user_id):
    ...
```

2.4 Mapa completo de rutas por rol

Ruta	Método	Acceso	Descripción
`/api/auth/register`	POST	Público	Registro (primer usuario → admin)
`/api/auth/login`	POST	Público	Inicio de sesión
`/api/auth/refresh`	POST	Cookie HttpOnly	Renovar access token
`/api/auth/logout`	POST	Autenticado	Cerrar sesión (blacklist)
`/api/auth/me`	GET	Autenticado	Datos del usuario actual
`/api/user`	GET/ POST	Autenticado	Perfil del usuario
`/api/weight`	POST	Autenticado	Registrar peso
`/api/imc`	GET	Autenticado	IMC actual
`/api/stats`	GET	Autenticado	Estadísticas
`/api/weights`	GET	Autenticado	Historial de pesos
`/api/admin/users/<id>/role`	PUT	**Solo admin**	Cambiar rol de usuario
`/api/defectdojo/*`	GET/ POST	**Solo admin**	Gestión DefectDojo
`/api/wstg/*`	GET/ POST	**Solo admin**	Sincronización WSTG

2.5 Asignación automática de roles en el registro

```
# app/routes.py — endpoint /api/auth/register
role = "admin" if storage.count_auth_users() == 0 else "user"
auth_user = storage.create_auth_user(username, password_hash, role=role)
access_token = create_access_token(auth_user.user_id, auth_user.username, role=auth
_user.role)
```

2.6 Frontend: envío del JWT y gestión del rol

El frontend usa `authenticatedFetch()` para añadir automáticamente el token a cada petición y reintentar con refresh si recibe 401:

```
// app/static/js/auth.js
static async authenticatedFetch(url, options = {}) {
    if (!this._accessToken) {
        const refreshed = await this._refreshAccessToken();
        if (!refreshed) throw new Error('No autenticado');
    }
    options.headers = {
        ...options.headers,
        'Authorization': `Bearer ${this._accessToken}`
    };
    let response = await fetch(url, options);
    if (response.status === 401) {
        const refreshed = await this._refreshAccessToken();
        if (refreshed) {
            options.headers['Authorization'] = `Bearer ${this._accessToken}`;
            response = await fetch(url, options);
        }
    }
    return response;
}
```

El rol se almacena en el objeto `_currentUser` y se expone con métodos auxiliares:

```
// app/static/js/auth.js
static getCurrentRole() {
    return this._currentUser ? this._currentUser.role : null;
}
static isAdmin() {
    return this.getCurrentRole() === 'admin';
}
```

---

3. Ejemplos de funcionamiento

3.1 Registro del primer usuario (admin automático)

Al registrar el primer usuario (`firstuser`), el servidor detecta que no hay usuarios en la BD y le asigna automáticamente el rol `admin` :

```
alejandro@pc247-14 Aplicación Médica % curl -s -X POST "localhost:5001/api/auth/register" -H "Content-Type: application/json" -d '{"username":"firstUser",
"password":"miClave_segura_123"}' | python3 -m json.tool
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxIiwidXNlcm5hbWUiOiJmaXJzdHVzZXIiLCJyb2xlIjoiYWRtaW4iLCJ0eXBlIjoiYWNjZXNzIiwianRpIjoi
em12WXdBdWVNT1dxcG9IV3g2RnZaZyIsImlhdCI6MTc3MTM1NTg4NywiZXhwIjoxNzcxMzU2Nzg3fQ.abJChJdX4y_4EFCWLe9t2iuZesQ-IFBetI5nOvpReCQ",
    "role": "admin",
    "user_id": 1,
    "username": "firstuser"
}
alejandro@pc247-14 Aplicación Médica %
```

Se observa que la respuesta incluye `"role": "admin"` y `"user_id": 1` .

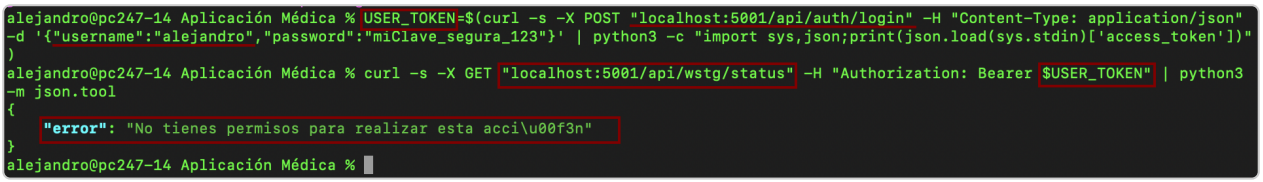
3.2 Registro de un segundo usuario (rol user)

Al registrar un segundo usuario (`alejandro`), al existir ya un usuario en la BD, se le asigna el rol `user`:

Se observa que la respuesta incluye `"role": "user"` y `"user_id": 2`.

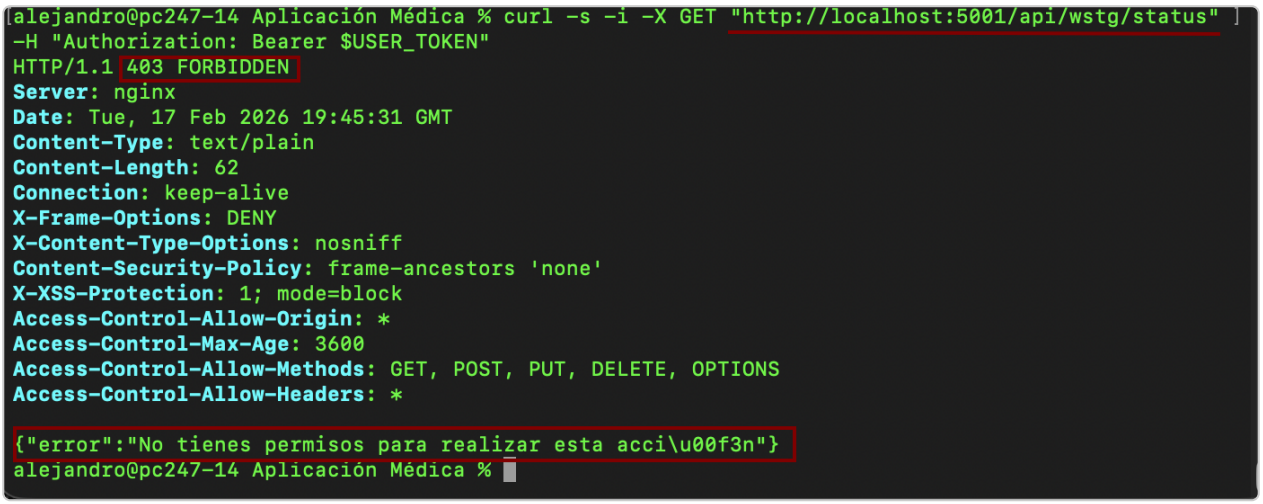
3.3 Acceso denegado por rol insuficiente (403 Forbidden)

El usuario `alejandro` (rol `user`) obtiene un access token e intenta acceder a la ruta de administración `/api/wstg/status`. El middleware `require_role("admin")` deniega el acceso:



La respuesta muestra `"error": "No tienes permisos para realizar esta acción"`.

Con la opción `-i` de curl, se puede verificar el código HTTP **403 FORBIDDEN** en las cabeceras de respuesta:



3.4 Token expirado (401 Unauthorized)

Cuando un access token expira (tras 15 minutos), el servidor responde con **401 UNAUTHORIZED** y el mensaje `"Token expirado"`. Esto demuestra que la validación de expiración funciona correctamente:

```
alejandro@pc247-14 Aplicación Médica % ADMIN_TOKEN=$(curl -s -X POST "localhost:5001/api/auth/login" -H "C
ontent-Type: application/json" -d '{"username":"firstuser","password":"miClave_segura_123"}' | python3 -c
"import sys,json;print(json.load(sys.stdin)['access_token'])")
alejandro@pc247-14 Aplicación Médica % curl -s -i -X GET "http://localhost:5001/api/wstg/status" -H "Autho
rization: Bearer $ADMIN_TOKEN"
HTTP/1.1 401 UNAUTHORIZED
Server: nginx
Date: Tue, 17 Feb 2026 20:24:20 GMT
Content-Type: application/json
Content-Length: 27
Connection: keep-alive
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
Content-Security-Policy: frame-ancestors 'none'
X-XSS-Protection: 1; mode=block
Access-Control-Allow-Headers: *

{"error":"Token expirado"}
alejandro@pc247-14 Aplicación Médica %
```

3.5 Acceso autorizado con rol admin (200 OK)

El usuario `firstuser` (rol `admin`) obtiene un nuevo access token y accede a la misma ruta `/api/wstg/status`. El middleware `require_role("admin")` permite el acceso y devuelve **200 OK**:

```
{"error":"Token expirado"}
alejandro@pc247-14 Aplicación Médica % ADMIN_TOKEN=$(curl -s -X POST "localhost:5001/api/auth/login" -H "C
ontent-Type: application/json" -d '{"username":"firstuser","password":"miClave_segura_123"}' | python3 -c
["import sys,json;print(json.load(sys.stdin)['access_token'])")                                          ]
[alejandro@pc247-14 Aplicación Médica % curl -s -i -X GET "http://localhost:5001/api/wstg/status" -H "Autho]
rization: Bearer $ADMIN_TOKEN"
HTTP/1.1 200 OK
Server: nginx
Date: Tue, 17 Feb 2026 20:27:12 GMT
Content-Type: application/json
Content-Length: 108
Connection: keep-alive
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
Content-Security-Policy: frame-ancestors 'none'
X-XSS-Protection: 1; mode=block
Access-Control-Allow-Headers: *

{"conflicts":0,"last_sync":"2025-12-03T17:45:18.378590","pending_items":0,"synced_items":4,"total_items":4
}
alejandro@pc247-14 Aplicación Médica %
```

3.6 Admin cambia el rol de un usuario

El admin (`firstuser`) promueve al usuario `alejandro` (user_id: 2) al rol `admin` mediante `PUT /api/admin/users/2/role`:

```
alejandro@pc247-14 Aplicación Médica % curl -s -X PUT "localhost:5001/api/admin/users/2/role" -H "Authoriza
tion: Bearer $ADMIN_TOKEN" -H "Content-Type: application/json" -d '{"role":"admin"}' | python3 -m json.tool

{
    "message": "Rol de 'alejandro' actualizado a 'admin'",
    "role": "admin",
    "user_id": 2
}
alejandro@pc247-14 Aplicación Médica %
```

La respuesta confirma: `"Rol de 'alejandro' actualizado a 'admin'"`.

3.7 Verificación del cambio de rol

Tras el cambio de rol, `alejandro` obtiene un nuevo access token (que ahora contiene `"role": "admin"` en el JWT) y accede a la ruta `/api/wstg/status` que antes le denegaba. Ahora recibe **200 OK**:

```
[alejandro@pc247-14 Aplicación Médica % curl -s -X PUT "localhost:5001/api/admin/users/2/role" -H "Authoriza
 tion: Bearer $ADMIN_TOKEN" -H "Content-Type: application/json" -d '{"role":"admin"}' | python3 -m json.tool

 {
     "message": "Rol de 'alejandro' actualizado a 'admin'",
     "role": "admin",
     "user_id": 2
 }
[alejandro@pc247-14 Aplicación Médica % USER_TOKEN=$(curl -s -X POST "localhost:5001/api/auth/login" -H "Con
 tent-Type: application/json" -d '{"username":"alejandro","password":"miClave_segura_123"}' | python3 -c "im
 port sys,json;print(json.load(sys.stdin)['access_token'])")
[alejandro@pc247-14 Aplicación Médica % curl -s -i -X GET "http://localhost:5001/api/wstg/status" -H "Author
 ization: Bearer $USER_TOKEN"
 HTTP/1.1 200 OK
 Server: nginx
 Date: Tue, 17 Feb 2026 20:33:40 GMT
 Content-Type: application/json
 Content-Length: 108
 Connection: keep-alive
 X-Frame-Options: DENY
 X-Content-Type-Options: nosniff
 Content-Security-Policy: frame-ancestors 'none'
 X-XSS-Protection: 1; mode=block
 Access-Control-Allow-Headers: *

 {"conflicts":0,"last_sync":"2025-12-03T17:45:18.378590","pending_items":0,"synced_items":4,"total_items":4}
 alejandro@pc247-14 Aplicación Médica %
```

Esto demuestra que el sistema RBAC es dinámico: al cambiar el rol en la BD, el siguiente token emitido refleja el nuevo rol.

3.8 Tests automatizados (225 pasados)

Se ejecutan 225 tests automatizados que cubren JWT, RBAC, validación de datos y lógica de negocio. Los 3 skipped corresponden a SQLCipher (no disponible en el entorno de test local):

```
alejandro@pc247-14 Aplicación Médica % python3 -m pytest tests/backend -v --tb=short 2>&1 | tail -5
tests/backend/whitebox/test_validation.py::TestEquivalencePartitionsAPI::test_fecha_particion_invalida_antigua PASSED [ 99%]
tests/backend/whitebox/test_validation.py::TestEquivalencePartitionsAPI::test_fecha_particion_valida PASSED [ 99%]
tests/backend/whitebox/test_validation.py::TestEquivalencePartitionsAPI::test_fecha_particion_invalida_futura PASSED [100%]

================== 225 passed, 3 skipped, 1 warning in 27.87s ==================
alejandro@pc247-14 Aplicación Médica %
```

4. Estructura de archivos modificados

Archivo	Cambio principal
`app/jwt_utils.py`	Generación y validación de JWT con claim `role`
`app/config.py`	`JWT_CONFIG` (secreto, algoritmo, tiempos de expiración)
`app/routes.py`	Decoradores `require_auth` y `require_role`, endpoints auth, asignación de roles
`app/storage.py`	Campo `role` en tabla `users`, `count_auth_users()`, `update_user_role()`, token blacklist
`app/__init__.py`	CORS: `Authorization` en lugar de `X-CSRF-Token`
`app/static/js/auth.js`	`authenticatedFetch()`, `isAdmin()`, `getCurrentRole()`, token en memoria
`app/static/js/sync.js`	Uso de `authenticatedFetch()` en lugar de `fetch()` + CSRF
`waf/modsecurity-override.conf`	Exclusión de `Authorization: Bearer` para evitar falsos positivos
`tests/backend/`	225 tests actualizados para JWT + RBAC