Bcrypt y JWT

Programación de Servicios y Procesos

Cristian Fernández

Viernes, 20 de febrero de 2026

Índice

1. Introducción

El objetivo de esta práctica es aplicar las librerías **Bcrypt** y **JWT** en un proyecto en construcción que tiene como objetivo gestionar las actividades y reservas de un gimnasio. Dicho proyecto está configurado de la siguiente manera:

- **Backend API.** El backend está programado en python usando el framework flask con base de datos en MongoDB. Fue mi elección por 2 razones, la primera porque nunca había "programado" en python y la segunda porque flask tiene fama de ser muy agradable de programar.
- **Frontend Escritorio.** El frontend de escritorio está programado en Vue y Electron y la razón es porque las especificaciones lo obligaban.
- **Frontend Móvil.** El frontend móvil está programado en React Native. Las opciones eran ésta o Flutter y me decidí por React por estar ya familiarizado por una práctica anterior y porque es muy sencillo compilarlo en máquina real.

2. ¿Por qué implementarlos?

La razón principal para implementar estas librerías en el proyecto es simple: **SEGURIDAD**.

Bcrypt nos permitirá hashear las contraseñas de los usuarios en la base de datos, dando una capa extra de seguridad en comparación con la encriptación. Si un ataque mal intencionado a la base de datos expusiese la información del usuario, el atacante se encontraría con una contraseña hasheada imposible de descrifrar.

JWT nos posibilitará crear sesiones para los diferentes usuarios de la aplicación en los diferentes frontends mediante la generación de tokens. Dichos tokens, de duración variable, nos permitirán autenticar, autorizar e intercambiar información en los diferentes endpoints de la aplicación.

Las librerías se implementarán en el Backend, y específicamente son las siguientes:

- *flask-bcrypt*
- *flask-jwt-extended*

Ambas se pueden encontrar en https://pypi.org/

3. ¿Qué son Bcrypt y JWT?

Bcrypt es una función de hashing (derivación de claves) diseñada para proteger contraseñas, basada en el cifrado de bloque Blowfish. A diferencia de algoritmos rápidos como SHA256, bcrypt es intencionalmente lento, lo que lo hace altamente resistente a ataques de fuerza bruta y precomputados.

Características principales:

- **Hash, no encriptación.** Bcrypt transforma una contraseña en una cadena de caracteres ininteligible (un hash) de una sola vía. No se puede "desencriptar"para obtener la contraseña original, solo verificarla.
- **Salting (Sal).** Bcrypt genera automáticamente una "sal"(una cadena aleatoria) para cada contraseña antes de hashearla. Esto asegura que, si dos usuarios usan la misma contraseña ("123456"), sus hashes almacenados sean completamente diferentes, protegiendo contra ataques de fuerza bruta.

JWT (JSON Web Token) es un estándar abierto que define una forma compacta y autónoma para transmitir información segura entre partes como un objeto JSON. Es ampliamente utilizado para la autenticación y autorización en aplicaciones web, permitiendo validar la identidad del usuario sin necesidad de almacenar sesiones en el servidor.

Características principales:

- **Autónomo (Stateless).** El token contiene toda la información necesaria sobre el usuario, eliminando la necesidad de consultar la base de datos en cada petición.
- **Firmado digitalmente.** Los JWT se firman (usando secretos o pares de claves pública/privada) para garantizar la integridad y autenticidad de los datos.

4. Implementación

Lo primero que debemos hacer es importar las librerías correspondientes en nuestra API.

```
9   from flask_bcrypt import Bcrypt
10  from flask_jwt_extended import JWTManager, create_access_token
11  from flask_cors import CORS
```

Figura 1: CORS será necesaria para que haya comunicación con los frontend

Empecemos por mostrar el endpoint de registro.

```
#### MÉTODOS POST ####

## REGISTRAR USUARIO EN APP
@app.route('/auth/registro', methods=['POST'])
def registroUsuario():
    coleccion = db['usuarios']
    datos = request.json

    # VALIDACIONES
    try:
        # Intentamos validar los datos obtenidos del registro
        datos = validacionRegistro.validate(datos)
    except Exception as e:
        # Si hay un error se especificará de forma detallada
        return jsonify({"ERROR": "Validación fallida", "detalle": str(e)}), 400

    # COMPROBACIONES
    # Extraemos contraseña y los campos que deben ser únicos
    nombre_usuario = datos.get('nombre_usuario')
    email = datos.get('email')
    dni = datos.get('dni')
    passPlana = datos.get('contraseña')

    # Validación de existencia en formulario de campos obligatorios
    # if not nombre_usuario or not passPlana or not email or not dni:
    #     return jsonify({"ERROR": "Debe rellenar los campos obligatorios (nombre de usuario, contraseña, email, dni)"}), 400

    # Comprobación de existencia en base de datos de los campos únicos anteriores
    usuario_existente = coleccion.find_one({
        "$or": [ # El operador $or devuelve un documento si coincide cualquiera de las condiciones
            {"nombre_usuario": nombre_usuario},
            {"email": email},
            {"dni": dni}
        ]
    })
```

Figura 2: Parte 1 - Parece que ha habido cambios en el sistema de validación...

```
82      # Si ya existe un usuario personalizamos el mensaje según qué campo falló
83      if usuario_existente:
84          if usuario_existente.get('email') == email:
85              mensaje = "Ese email ya está registrado"
86          elif usuario_existente.get('dni') == dni:
87              mensaje = "Ese DNI ya está registrado"
88          else:
89              mensaje = "El nombre de usuario ya está en uso"
90
91          return jsonify({"ERROR": mensaje}), 400
92
93      # Usamos Bcrypt para hashear la contraseña
94      passPlana = datos.get('contraseña')
95      passHasheada = bcrypt.generate_password_hash(passPlana).decode('utf-8')
96
97      nuevoUsuario = {
98          "nombre_usuario": nombre_usuario,
99          "contraseña": passHasheada, # Se almacenará la pass hasheada y no plana
100         "nombre": datos.get('nombre'),
101         "apellidos": datos.get('apellidos'),
102         "dni": dni,
103         "telefono": datos.get('telefono'),
104         "email": email,
105         "rol": "cliente", # cliente/administrador, cliente por defecto al registrarse
106         "fecha_alta": datetime.now(),
107         "estado_suscripcion": True
108     }
109
110     idGeneradoNuevoUsuario = coleccion.insert_one(nuevoUsuario).inserted_id
111
112     return jsonify({
113         "mensaje": "Usuario registrado correctamente",
114         "id": str(idGeneradoNuevoUsuario)
115     }), 201
```

Figura 3: Parte 2

En la línea 94 podemos observar cómo asignamos a una variable la contraseña obtenida en la cabecera de la petición a la hora de hacer un registro. Posteriormente, en la línea 95, aplicamos el hasheo a la constraseña y la almacenamos en una nueva variable que, posteriormente en la línea 99, la incluiremos como parte del objeto *'nuevoUsuario'* que almacenaremos en la base de datos en la línea 110.

El trabajo en el endpoint de registro ya está hecho, pasemos ahora al endpoint de login.

```
118  ## LOGIN USUARIO EN APP
119  @app.route('/auth/login', methods=['POST'])
120  def loginUsuario():
121      coleccion = db['usuarios']
122      datos = request.json
123
124      # VALIDACIONES
125      try:
126          # Intentamos validar los datos obtenidos del registro
127          datos = validacionLogin.validate(datos)
128      except Exception as e:
129          # Si hay un error se especificará de forma detallada
130          return jsonify({"ERROR": "Validación fallida", "detalle": str(e)}), 400
131
132      nombre_usuario = datos.get('nombre_usuario')
133      passPlana = datos.get('contraseña')
134
135      usuario = coleccion.find_one({"nombre_usuario": nombre_usuario})
136
137      # Verificamos si el usuario existe y si la contraseña coincide
138      if usuario and bcrypt.check_password_hash(usuario['contraseña'], passPlana):
139
140          # Creamos token guardando el id del usuario y su rol
141          token = create_access_token(
142              identity = str(usuario['_id']),
143              additional_claims = {"rol": usuario.get('rol')}
144          )
145
146          return jsonify({
147              "token": token,
148              "usuario": {
149                  "nombre_usuario": usuario.get('nombre_usuario'),
150                  "rol": usuario.get('rol')
151              }
152          }), 200
153
154      return jsonify({"ERROR": "Nombre de usuario o contraseña incorrectos"}), 401
```

En la línea 138 podemos comprobar que, si existe en la base de datos el nombre de usuario introducido y, si la contraseña hasheada almacenada de ese usuario se compara con la *"contraseña plana"* obtenida del login y devuelve verdadero el login será satisfactorio y crearemos un token. De lo contrario devolverá un error 401.

Los métodos clave son *check_password_hash* y *create_access_token* de las librerías bcrypt y jwt respectivamente.

Vamos a logearnos en la aplicación con un usuario ya creado en MongoDB.


```
_id: ObjectId('699892e499197ad91114b333')
nombre_usuario : "manullo"
contraseña : "$2b$12$ZZ8/hRZNFux/VRJ8q5IdXuKodyHwqNRAD/xjNSU9o681uE5liVP6m"
nombre : "Manolo"
apellidos : "Fernández Ulloa"
dni : "12345678U"
telefono : ""
email : "manullo@gmail.com"
rol : "cliente"
fecha_alta : 2026-02-20T17:59:16.897+00:00
estado_suscripcion : true
```


Figura 4: Como se puede apreciar, la contraseña está perfectamente hasheada

Veamos el login del frontend de escritorio.

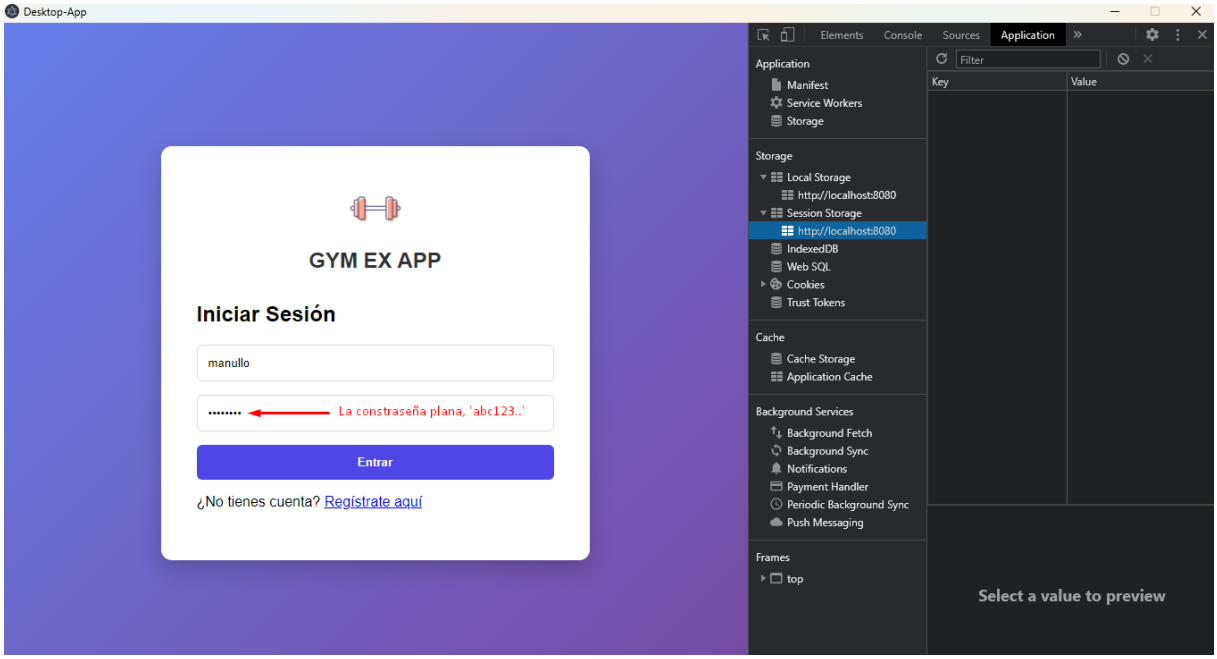

Figura 5: Vista en desarrollo

Si el logeo es satisfactorio, entraremos en la aplicación.

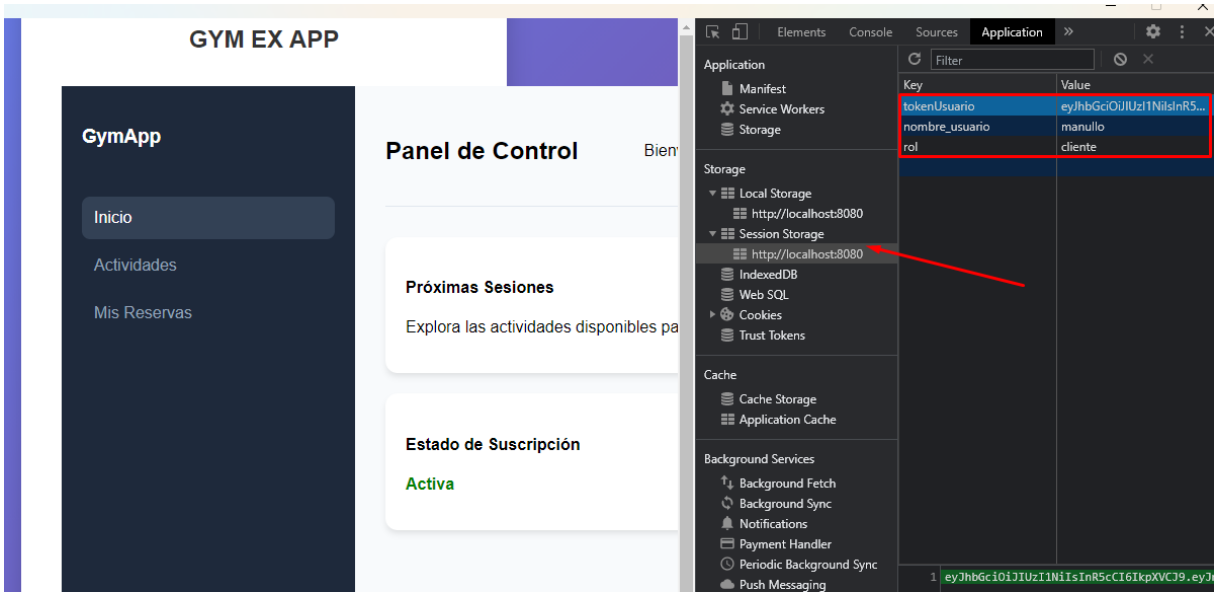

Figura 6: Vista en desarrollo

Como se puede apreciar, en el Session Storage queda almacenado el token del usuario, su nombre y su rol.